Institute for Visualization and Interactive Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Acceleration of P-k-d tree traversal using probabilistic occlusion

Pascal Walloner (3494480)

Course of Study:	Informatik
Examiner:	Prof. Dr. Thomas Ertl
Supervisors:	Dr. Guido Reina, Patrick Gralka, M.Sc., Prof. Markus Hadwiger
Commenced:	November 20, 2021
Completed:	May 20, 2022
CR-Classification:	I.7.2

Abstract

The growing amount and complexity of particle data in various fields of study (e.g. molecular dynamics) poses an increasing demand of high-quality and high-performance visualizations of such data. Ray tracing is known for the high image quality it can produce and its well-suitedness to global illumination effects, e.g. shadows or ambient occlusion. Effects like these can serve as helpful cues for understanding three-dimensional structures within the data. As an optimization for the rendering of particle data, Wald et al. [WKJ+15] suggest the Particle-k-d (P-k-d) tree, a ray tracing acceleration structure for particles which does not require any memory overhead. Ibrahim et al. [IRR+21] propose a novel, probabilistic method to accelerate particle rendering by culling particles which are likely to be occluded. The goal of this thesis is to examine the viability of probabilistic occlusion culling in the context of a P-k-d-based ray tracer. I contribute by adapting the method proposed by Ibrahim et al. for the use within a ray tracing application and implementing the adapted method as an extension of a P-k-d ray tracer by Gralka et al. [GWG+20]. I go on to evaluate my approach with respect to visual quality and performance. My findings show that probabilistic occlusion culling can provide a performance improvement during super sampling. However, the magnitude of the improvement and the effect on visual quality are greatly dependent on the size of and features within the data set among other factors.

Kurzfassung

Die wachsende Menge und Komplexität von Partikeldaten in zahlreichen Forschungsgebieten (z.B. Molekulardynamik) erhöht die Nachfrage an hochqualitativen und hochperformanten Visualisierungen solcher Daten. Raytracing ist allgemein für seine hohe Bildqualität bekannt, sowie für seine gute Eignung gegenüber globaler Beleuchtungseffekte (z.B. Schatten oder Ambient Occlusion). Solche Effekte können als hilfreiche Referenz für das Verständnis drei-dimensionaler Strukturen innerhalb der Daten dienen. Um Raytracing für das Rendering von Partikeldaten zu optimieren, schlagen Wald et al. [WKJ+15] den Partikel-k-d-Baum (P-k-d-Baum) vor, eine Beschleunigungsstruktur für Partikeldaten, welche ohne zusätzlichen Speicherbedarf auskommt. Ibrahim et al. [IRR+21] präsentieren eine neuartige, probabilistische Methode zur Beschleunigung des Partikelrenderings durch Cullen mit großer Wahrscheinlichkeit verdeckter Partikel. Das Ziel dieser Arbeit ist, die Praktikabilität des Cullings basierend auf probabilistischer Verdeckung im Kontext eines P-k-d-basierten Raytracers zu untersuchen. Hierzu trage ich Anpassungen der Methode von Ibrahim et al. bei und implementiere die angepasste Methode als Erweiterung eines P-k-d-Raytracers von Gralka et al. [GWG+20]. Des

Weiteren evaluiere ich den Ansatz hinsichtlich visueller Qualität und Leistungsfähigkeit. Meine Ergebnisse zeigen, dass Culling durch probabilistische Verdeckung eine Leistungssteigerung während des Supersamplings erzielen kann. Diese, sowie der Effekt auf die visuelle Qualität, hängen jedoch u.A. stark von der Größe und den Eigenschaften der gerenderten Daten ab.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

AS Acceleration Structure. 7, 15, 18, 20, 25, 26, 27, 29, 31, 47

BVH Bounding Volume Hierarchy. 7, 18, 19, 20, 25

LOD Level-of-Detail. 15, 25

1 Introduction

Particle data sets are becoming larger and more common in various fields of study, ranging from molecular dynamics and atomistic simulations to astrophysics and cosmology. With the increasing size and complexity of particle data, the demand for high-quality and high-performance visualizations of such data is also growing. Modern particle simulations yield data sets containing millions or even billions of particles. At such scales traditional rasterization approaches have to rely on Level-of-Detail (LOD) methods, filtering data to fit the current scale of exploration, in order to keep performance acceptable. This may, in the worst case, result in a biased visualization hiding key features within the data at certain scales. Furthermore, with particle data of such sizes, individual particles usually only take up much less than a single pixel when projected to the screen. Thus extensive super sampling becomes necessary to prevent artifacts from arising when only one or a few samples per pixel are taken. Ibrahim et al. [IRR+21] suggest a novel approach to accelerate the super sampling process by incrementally building a probabilistic depth map. It serves as the basis to cull occluded particles from the scene, so they do not have to be traversed for future samples, increasing performance and doing away with the need for LOD-based methods. They implement their method into a Vulkan rasterization pipeline, achieving impressive results.

When it comes to understanding features inside particle data, cues like shadows or ambient occlusion can serve as a useful tool, making visual communication of these features more intuitive. Rasterization-based rendering methods, however, do not inherently offer a way to implement these global illumination techniques and often times only allow for an approximation. Ray tracing on the other hand, assumes a model of light that is quite close to streams of photons in the physical world, enabling the rendering of ambient occlusion and shadows by casting secondary rays. The computational cost of ray tracing though, is quite high compared to rasterization methods. Even with recent hardware advances in the field bringing interactive ray tracing capabilities to consumer GPUs, the need for efficient use of memory and compute power by ray tracing applications remains. Wald et al. [WKJ+15] propose the Particle-k-d (P-k-d) tree, a novel Acceleration Structure (AS) for ray tracing of particle data, reducing memory overhead down to zero. Gralka et al. [GWG+20] optimize their method in terms of traversal time, by proposing a hybrid AS.

The goal of this thesis is to examine the viability of probabilistic culling of occluded particles as proposed by Ibrahim et al. in the context of a ray tracing application using P-k-d trees. I make adaptions to the probabilistic occlusion culling approach in order for it to operate on particle data structured in P-k-d trees and decrease their traversal time to improve performance and to take full advantage of the ray tracing model. I extend the prototype built by Gralka et al. [GWG+20] to feature the adapted probabilistic occlusion culling method and evaluate it in terms of visual quality and performance.

Structure

I structure this thesis as follows:

Chapter 2 – Fundamentals: I begin by introducing fundamental concepts this thesis builds on. I briefly introduce the concept of ray tracing and acceleration structures, focusing on P-k-d trees and their advantages over traditional structures. Furthermore, I give a high-level overview of the probabilistic occlusion culling approach proposed by Ibrahim et al. [IRR+21].

Chapter 3 – Related Work: Following the fundamentals, I place this thesis within the context of existing work employing different approaches for particle rendering. Additionally, I present OptiX [PBD+10], a common framework for ray tracing applications, which I use to implement my approach.

Chapter 4 – Methodology: This chapter outlines the key conceptual changes I make to the probabilistic method to adapt it for the application in a P-k-d-based ray tracer.

Chapter 5 – Implementation: Following the conceptual overview, I go into detail about how I implement the modified particle culling technique. The prototype that results from this is the test setup I use to examine visual quality and performance of the adapted approach.

Chapter 6 – Results: In this chapter, I present the results of the evaluation. I show and explain common artifacts and how different parameters affect them. Analogously, I analyze the computational performance of the implementation. Finally, I discuss the results and put them into context with existing techniques.

Chapter 7 – Summary and Discussion: I conclude by giving a summary of the contributions I make as part of this thesis as well as providing an outlook on how future work might improve the limitations of my method.

2 Fundamentals

2.1 Ray Tracing

In computer graphics we distinguish between object-order and image-order approaches for image synthesis. As opposed to iterating over the objects in a scene and calculating their contribution to the rendered image, like rasterization algorithms [FVV+96] do, ray tracing follows the image-order approach. Ray tracing [Gla89] models light as rays within the scene whose trajectories are followed backwards, i.e. we cast one or more rays through each of the image's pixels on the viewing plane and intersect them with the scene's geometry to determine the pixel's color. Each ray can be interpreted as a sample of the stream of photons inside the pixel's frustum. Correctly modeling this stream would require solving the integral over each ray that can be cast through the pixel's footprint (i.e. the area on the viewing plane mapping to a given pixel on the screen) which we can only estimate in practice. We perform this estimation by treating each ray as a sample. If only a single ray is cast, we typically let it intersect the pixel's footprint at its center. Accumulating the result of multiple ray casts through different points on the pixel's footprint (super sampling), however, improves the rendering quality significantly. Additionally, we may cast secondary rays from the intersection point to model reflection or to check whether the mapped point is shaded or not.

Traditionally, rays are cast from a single origin through the corresponding pixel's footprint on the viewing plane modeling a pinhole camera. Thus the rays diverge leading to a perspective projection. For this work, I use an orthographic projection, i.e. all rays are parallel and their intersection with the viewing plane is always orthogonal and merely defined by their origin (see figure 2.1).

Since the ray tracing model follows the real world physics of light quite closely, the visual appearance of ray traced images is usually rated very well compared to the results of rasterization algorithms. On the other hand, this comes at a high cost as computing a large amount of ray-geometry intersections per frame is a very computationally intensive task. For this reason, rasterization is still widely used in interactive applications like computer games. In off-line renderings, where the focus lies on visual quality rather than rendering time, ray tracing can produce results that are very close to physical correctness.

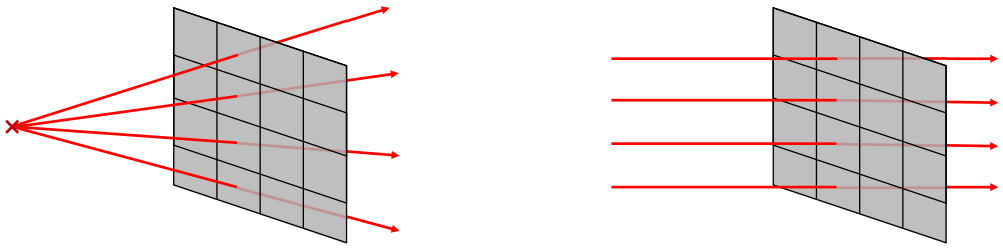

Figure 2.1: Rays intersecting the viewing plane. Diverging rays with common origin modeling a pinhole camera leading to a perspective projection (left) and parallel rays leading to an orthographic projection (right).

2.2 Acceleration Structures

In order to reduce the computational demands of ray tracing, it is helpful to reduce the amount of geometry a ray has to be checked against for intersections. Storing geometry naively without any spatial connotation to the scene (i.e. keeping an unordered list of objects) does not allow for this, as no information about an object's placement in the scene can be derived from its place within the data structure. To improve this we define data structures that allow for this kind of spatial connotation. This way we can disregard certain parts of the data structure, which we are sure the ray does not intersect. This speeds up traversal as intersection checks have to be performed on less geometry primitives. We call a data structure like this an Acceleration Structure (AS). While non-hierarchical structures (e.g. uniform grids) provide a step in the right direction, hierarchical ASs, especially for very large data, provide a significant performance benefit. In the following I introduce the most important ASs when it comes to rendering large particle data.

- **Bounding Volume Hierarchies** (BVHs) allow for an even larger computational advantage than non-hierarchical ASs [TS+05]. They are one of the most commonly used types of ASs. As the name implies, BVHs provide a tree structure with nodes of bounding volumes of primitives. The root node is a bounding volume containing all the primitives within the scene. Inner nodes contain a subset of objects within the parent's volume, while leaf nodes correspond to single primitives. If the hierarchy is balanced, computational complexity of traversal becomes logarithmic, which for large amounts of geometry provides a significant performance increase. Usually, BVHs use axis-aligned bounding boxes as bounding volumes since intersections between them and common geometry primitives (e.g. rays, triangles, spheres)

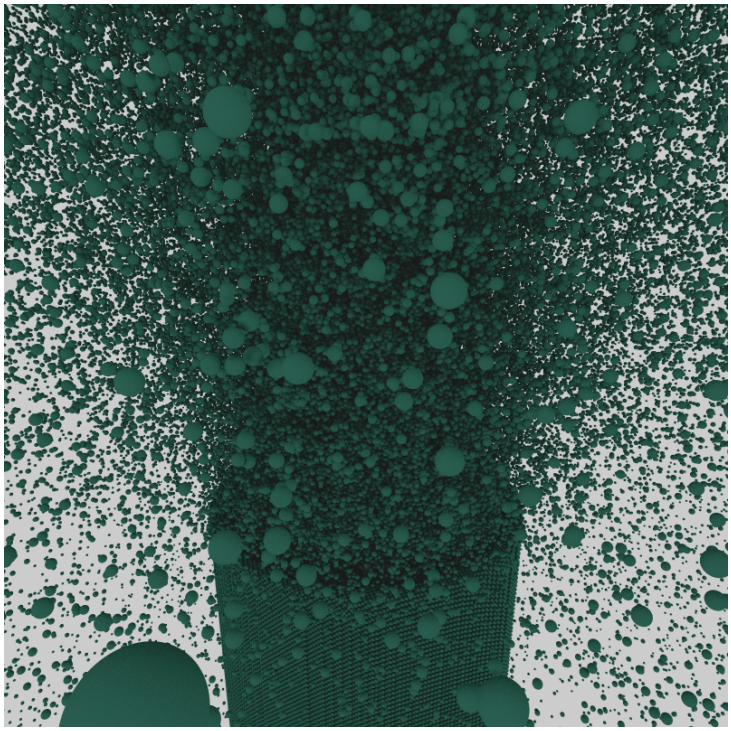

Figure 2.2: A ray traced image of the *Nozzle*-dataset (see chapter 6) rendered using the particle ray tracer by Gralka et al. [GWG+20].

are easy and fast to compute. Yet, their speed comes at a cost: As the bounding volumes are not implicitly defined by their contained geometry, their extents have to be stored separately, leading to a significant memory overhead. This can become a limiting factor when rendering large sets of data as GPU memory is still a fairly expensive resource.

- **K-D trees** [HH11] are a special case of BSP-trees. They hierarchically subdivide space along axis-parallel planes. K-d trees have a significantly smaller memory overhead than BVHs. For scenes solely comprised of particles (i.e. spheres with a center and radius), Wald et al. [WKJ+15] take this concept even further, defining Particle-k-d trees (P-k-d trees) which do not carry any memory overhead. They choose space partitioning planes to subdivide each node. Their goal is to define these without the need for storing any additional data. The position of splitting planes is defined by a particle within the node to save memory as particle positions are already part of the stored data and therefore no overhead is necessary. The chosen particle thus lies directly on the splitting plane and is contained in neither of the nodes children, i.e. a P-k-d tree's inner nodes actually store particles whereas inner nodes of a BVH do not. To complete the definition it is necessary to specify the planes' orientations. Wald et al. have found the implicit definition through

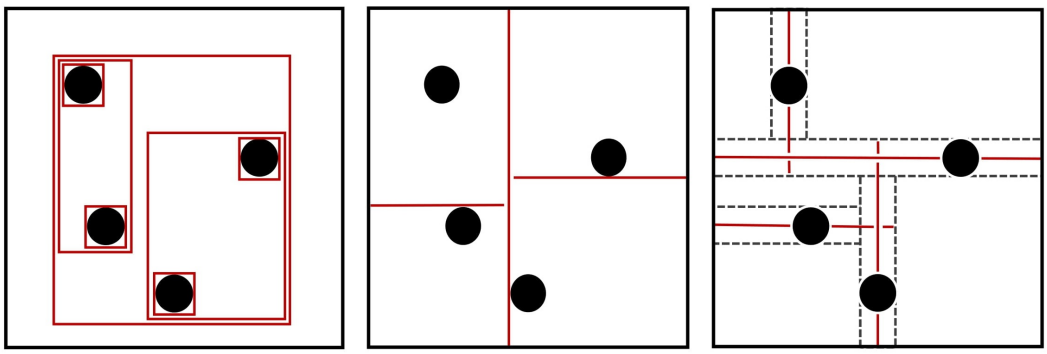

Figure 2.3: 2D-schematic of Acceleration Structures. The same set of particles is represented by a BVH (left), a traditional K-d tree (center) and a P-k-d tree as proposed by Wald et al. [WKJ+15] (right) using the maximum extent strategy to define the orientation of splitting planes (red). Dashed lines represent the extruded bounds of child nodes.

the maximum extent of the node's bounding volume to work best. This means that a splitting plane's orientation is chosen so that it subdivides a node through the dimension in which its extent is largest. They do this to prevent nodes from deforming into long, non cube-like shapes with increasing depth in the tree. The particle chosen to define the splitting plane is typically the median particle w.r.t. its position along the dimension chosen for subdivision. This is done to yield a balanced tree minimizing its depth. To ensure however, that particles intersecting the partitioning plane are bound by any of the child nodes, the authors extrude the child nodes' volumes by the maximum particle radius behind the partitioning plane, which leads to intersecting bounding boxes of child nodes.

As a result of the implicit definition of subdivisions, storing particle positions in a heap structure (i.e. reordering the raw particle data) is sufficient to fully define a P-k-d tree, making the method very memory-efficient since no overhead is necessary.

Figure 2.3 shows how the same set of particles would be structured by a BVH, a traditional K-d tree and a P-k-d tree.

2.3 Probabilistic Occlusion Culling

Ibrahim et al. [IRR+21] suggest a novel, probabilistic method to accelerate the rendering of large particle data. This section aims to outline their approach.

They employ a probabilistic model which accumulates per-pixel information about sample depths and based on this information makes decisions to cull groups of particles from the scene that are likely to be occluded by others.

Essential to their approach is the **depth-confidence map**. It maps each pixel to a tuple (d, C) representing a depth d and a confidence C with C being the estimated probability that a new sample, uniformly chosen within the pixel's footprint, intersects a particle at a depth $\leq d$. This map is built progressively during the super sampling process. After the first sample, C denotes the projected area of the sampled particle onto the pixel's footprint. During subsequent samples of particles closer than d, they increase C (according to equation 2.1), which has it approach a value $\in [0, 1]$. They call this process confidence accumulation. Once C reaches a chosen threshold confidence C_{occ} (which is typically chosen quite high, e.g. $C_{occ} := 0.95$), they deem particles inside that pixel's frustum with a depth $> d$ to be occluded and cull them from the scene so they do not have to be traversed for future samples.

The authors describe confidence accumulation by defining the confidence after k samples $C(k)$ by the fraction of the pixel's footprint's area jointly covered by the sampled particles $A_{n(k)}$, where $n(k)$ denotes the number of *unique* particles hit by k samples. Assuming constant particle size a, they approximate $A_{n(k)}$ as follows:

$$C(k) := A_{n(k)} \approx 1 - (1 - a)^n, \text{ with } n := E(n(k)) \tag{2.1}$$

Since particles might overlap, they assume that each new particle contributes the area it shares with the pixel's footprint by the fraction of sub-pixel area that has not yet been covered by other particles. As the number of unique particles sampled $n(k)$ is not efficiently retrievable, the authors use its expected value $E(n(k))$ instead. Computing $E(n(k))$ requires knowledge about the number of particles N the k samples are chosen from, i.e. the total number of particles inside the pixel's frustum that have not yet been culled. They approximate N using a pre-computed **particle density field** of the scene, discretized on a uniform grid. Via volume ray casting, they traverse this grid along the pixel's frustum to obtain not only an approximation of the total number of particles N inside the pixel's frustum, but also a per-pixel density histogram mapping discrete depth intervals to the estimated number of particles within them and the pixel's frustum. By this histogram the authors define a per-pixel function $D(d_1, d_2)$ yielding the estimated number of particles inside the pixel's frustum between depths d_1 and d_2.

Using the aforementioned concepts as a foundation, Ibrahim et al. go on to propose a double-buffered algorithm (algorithm 2.1) to accumulate depth and confidence updates. The reason for the use of two depth-confidence maps is the fact that a balance between depth and confidence has to be maintained: On the conservative side, large depth values d lead to high confidence values C as then there are many particles at smaller depths eligible to contribute to the confidence accumulation, but as the confidence C only

Algorithm 2.1 Double-Buffered Depth Confidence Updates

if $d_s \leq d_{cull}$ **then**
  $C_{cull} \leftarrow$ accumulateConfidence(C_{cull}, a, k_{cull}); k_{cull}++
end if
if $d_s \leq d$ **then**
  $P(d_s) \leftarrow$ acceptanceProbability(d_s); $u \leftarrow$ rand(0,1);
  if $d_s < d \wedge u \leq P(d_s)$ **then**
    $(d, C) \leftarrow (d_s, a)$; $k \leftarrow 1$
  else
    $C \leftarrow$ accumulateConfidence(C, a, k); k++
  end if
end if
if OP(d_{cull}, C_{cull}) < OP(d, C) **then**
  $(d_{cull}, C_{cull}) \leftarrow (d, C)$; $k_{cull} \leftarrow k$
end if

carries meaning for particles with a depth larger than d, there are only relatively few particles left to potentially cull. On the other hand, small depths d will leave many particles eligible to cull, but C might not ever accumulate large enough so that any culling takes place. To measure this, the authors define a metric of potentially occluded particles OP(d, C) by the product of the estimated amount of particles eligible to cull $D(d, d_{max})$ and the confidence C:

$$\text{OP}(d, C) = \text{D}(d, d_{max})C. \tag{2.2}$$

To facilitate a double-buffered approach, in addition to the accumulation buffer, the authors keep an additional depth-confidence buffer (d_{cull}, C_{cull}) (which they call the *cull*-buffer) that aims to represent the conservative side of the spectrum, prioritizing a high confidence. While they reset the accumulation buffer with new samples closer than d if they deem it likely enough (*acceptance probability*) that restarting the accumulation from the sampled closer depth d_s will accumulate sufficient confidence fast enough, they only overwrite the *cull*-buffer if it actually holds that the accumulation buffer potentially allows for the culling of more particles than the *cull*-buffer currently does, i.e. OP(d_{cull}, C_{cull}) < OP(d, C). If C_{cull} surpasses the threshold value C_{occ} at some point during confidence accumulation, particles inside the corresponding pixel's frustum that have a depth $> d_{cull}$ are considered to be occluded and thus culled from both the scene and the density field for future iterations. However, Ibrahim et al. do not decide about culling for individual particles, but rather group them into *meshlets* and further into *nodes* and make culling decisions for these larger groups of particles based on multiple pixels' depth-confidence values. They determine the algorithm's convergence by monitoring the

amount of iterations passed since the last culling of a group of particles has occurred. If for long enough no more particles were culled from the scene, they assume the algorithm to have converged.

It is the goal of this thesis to examine the viability of the probabilistic approach by Ibrahim et al. [IRR+21] applied to a ray tracer structuring particles into P-k-d trees as proposed by Wald et al. [WKJ+15].

3 Related Work

3.1 Particle Rendering

An early approach to rendering large particle data is volume rendering. Cha et al. [CSI09] use the GPU to transform particle data to volume data which can subsequently be visualized using volume rendering algorithms [KM05].

Other than that, rasterization [ZTH03] has been widely used for particle rendering. Examples for this include VMD [HDS96], MegaMol [GKM+15] and VisIt [Chi12]. Rasterization-based methods usually come at a low performance cost. On the other hand however, extending them to feature global illumination techniques like shadows or ambient occlusion, which can significantly enhance the visual communication of structures in the data, is not trivial. Though there are methods to implement such techniques in rasterization-based approaches [KRZ+17] [SGG15], the ray tracing model naturally lends itself well to the implementation of global illumination techniques by the use of secondary rays. Through the recent advancement of ray tracing hardware to even consumer level GPUs the technique has become much more available, even at interactive rates, making it a viable approach for visualization problems.

A variety of methods have been proposed to improve both visual and performance results of ray traced particle renderings. Le Muzic et al. [LPSV14] use an LOD-based approach to render particle data consisting of billions of atoms. Wald et al. [WKJ+15] suggest the P-k-d tree, a novel AS specifically tailored to memory-efficient particle rendering. Gralka et al. [GWG+20] build on their approach to find that a hybrid structure, built from an outer BVH containing several small P-k-d treelets at its leafs, serves as a good compromise between low memory consumption and fast traversal times.

3.2 Ray Tracing Frameworks

There are numerous frameworks for building ray tracing applications. For this thesis, I build on an implementation by Gralka et al. [GWG+20] which uses NVIDIA OptiX [PBD+10]. As opposed to frameworks like OSPRay [WJA+17], which runs on the CPU,

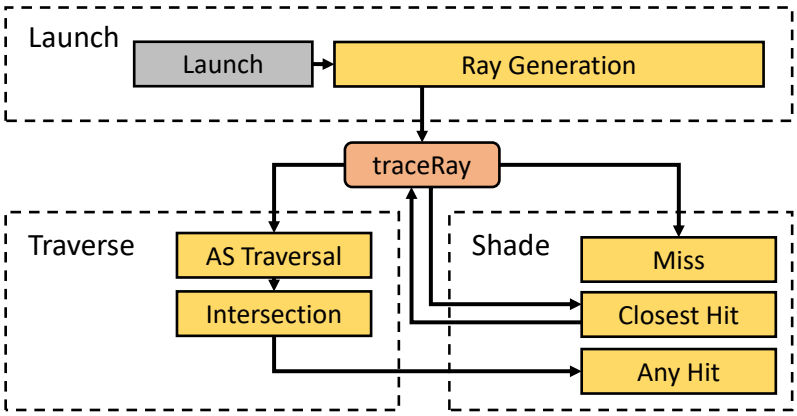

Figure 3.1: Call graph representing a simplified version of the OptiX ray tracing pipeline [PBD+10]. `traceRay` is a built-in function first called by the `RayGen` program. `AS traversal` calls the user-specified `Intersection` program which in turn calls the `Any Hit` program if an intersection occurs. For the hit closest to the ray's origin, the `Closest Hit` program is called, which is typically responsible for secondary ray casts, thus it may also call the `traceRay` function. If no intersection occurs, the `Miss` program is called.

OptiX takes advantage of NVIDIA graphics hardware and thus provides high performance capabilities. OptiX is a general purpose ray tracing framework which allows the user to customize every part of the ray tracing pipeline.

The OptiX ray tracing pipeline consists of several stages, which are user-specifiable using programs written in CUDA C, an OptiX-specific version of C used to program NVIDIA graphics hardware. The pipeline (figure 3.1) is essentially structured as follows: **Ray generation (RayGen)** defines rays to be cast into the scene. Camera models, such as perspective or orthographic projection, as well as techniques like super sampling or accumulation of color samples [HA90] are implemented here. For each ray cast, this program calls the built-in `traceRay` function. **Intersection** checks for geometry eligible after AS traversal if an intersection actually occurs and if so, calls the **Any Hit** program. For the intersection closest to the ray's origin, the **Closest Hit** program usually handles shading and secondary ray casts. If no intersection occurs, the **Miss** program defines a background color or implements environment maps.

4 Methodology

The goal of this thesis is to combine the probabilistic particle culling approach by Ibrahim et al. [IRR+21], culling particles from the scene that are confidently classified as occluded with the memory-efficient ray tracing acceleration structure of P-k-d trees as proposed by Wald et al. [WKJ+15]. Ibrahim et al. implement their method using the Vulkan rasterization pipeline grouping particles into *meshlets* and *nodes*. As ray tracing and P-k-d trees provide entirely different preconditions, some conceptual adjustments become necessary. This section describes and justifies the adjustments I choose to make in order to combine the approaches.

4.1 Sampling the Scene

Ibrahim et al. interpret each fragment with its depth as a sample towards the accumulation algorithm, i.e. causing an iteration of the accumulation algorithm. The ray tracing equivalent of fragments rasterized onto a pixel are ray geometry intersections. It is important to note that a sample in ray tracing traditionally refers to the closest intersection reported during AS traversal. Since fragments in a rasterization pipeline are processed before depth sorting however, I have to take into account any ray particle intersection, not just the closest one. Just as it is the case for fragment processing, ray particle intersections found during P-k-d traversal do generally not occur ordered by depth. When mentioning the word "sample" I refer to samples toward the accumulation algorithm and not to the closest ray particle intersection, which is what a sample is typically interpreted as in a ray tracing context.

In alignment with the implementation by Ibrahim et al. and several other works from the field of particle rendering, I render the scene using an orthographic projection (i.e. all rays are parallel). This comes with the advantage that the projected area of particles onto the viewing plane (particle footprint) a does not depend on the depth of the particle, but only on the particle's radius. For full correctness, it is necessary to determine whether the particle footprint intersects the edge of the pixel's footprint and if so, calculate the amount of area they both share. However, I assume a use-case where

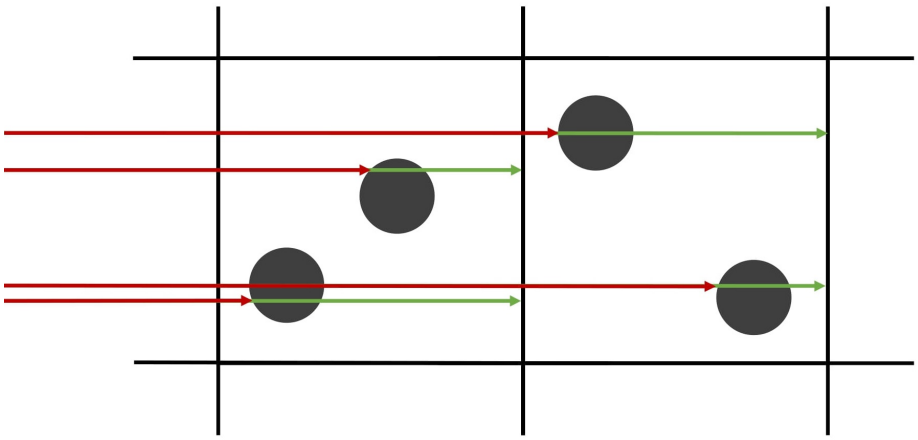

Figure 4.1: 2D schematic of depth quantisation. Four example samples of a simple scene consisting of four particles (grey) within two voxels of the particle density grid (black). Note that a sample is not the closest hit, as it is in the traditional ray tracing sense, but rather any ray particle intersection. The actual depth at which the intersection occurs (red) is quantised to the back-face of the current density grid voxel (green).

particles are significantly smaller than pixels in terms of their footprints. Thus this case appears only rarely and has a negligible impact on final results.

A conservative modification I make to depth samples reported during traversal is the following: I trace the ray further until it intersects the back-face of the density grid voxel. This way depth samples from within the same voxel will be quantised to the same value. I call this measure *depth quantisation*. The motivation for this is the fact that if a sample closer than the current depth d is found, the *acceptanceProbability* function (see algorithm 2.1) decides whether to restart accumulation from the new sample or not. This function merely depends on readings from the density grid at depths d and d_s, which if within the same voxel, query information at a resolution the density grid is too coarse for. Therefore I argue that a well-grounded decision cannot be made in this case and that the depth samples should be treated as the same. Figure 4.1 shows a 2D schematic of depth quantisation.

4.2 Particle Culling

The original authors' approach groups particles into *meshlets* and further into *nodes*. The algorithm classifies *meshlets* into one of four classes ranging from *occluded* to *potentially*

visible based on depth-confidence values of the pixels the particles contained project to. Once confidence accumulates sufficiently and a culling decision is made, they remove the particles inside the culled *meshlet* from the scene for future samples. Additionally, in order for the density volume to stay consistent with the updated scene, it has to be updated as well.

Ray tracing architectures offer the ability to define a depth d_{max} per ray indicating the maximum depth until which intersections should be reported, which allows for an elegant and efficient approach to culling particles on a fine scale. All intersections occurring deeper than d_{max} will be ignored. Efficient AS traversal implementations take this into account and do not descend into subtrees whose bounding box only intersects the queried ray at depths $> d_{max}$. This enables me to use depth-confidence values directly for ray generation without the need for an image pyramid of depth-confidence maps, which Ibrahim et al. use to efficiently find a tuple (d, C) representative for the entire meshlet. Their approach however comes with an advantage: The probabilistic nature of the entire method yields erroneous overestimations of confidence in certain cases for single pixels, which may cause visual artifacts by culling visible parts of geometry. Ibrahim et al. construct the depth-confidence image pyramid bottom-up by averaging confidence values and using the maximum depth over a 2×2 group of pixels per mipmap level, preventing these artifacts by basing culling decisions on multiple pixels' depth-confidence values, averaging out misjudgements for single pixels. While I still make culling decisions on a per-pixel level, I base these on depth-confidence values in a small neighborhood kernel of user-specified size. In some cases the use of a kernel is not necessary. If it is, a small 3×3 neighborhood kernel is usually sufficient (see chapter 6.1).

In summary: I make culling decisions based on the average confidence in a small neighborhood around the pixel in question. If the average confidence is sufficient, I cull geometry behind the largest depth d_{cull} of any pixel in the neighborhood by casting rays of subsequent iterations with that depth as d_{max}. Since I clip rays cast through the particle density volume the same way, updating it also becomes unnecessary.

The consequences of this approach extend to determining accumulation convergence. Since culling is no longer a global decision (i.e. actually manipulating the scene), multiple pixels do not influence each others' accumulations. Therefore it makes sense to determine convergence for each pixel separately, i.e. to track the amount of samples since d_{max} for rays cast through the pixel was last decreased and to stop accumulation once a threshold of frames is reached. I define global convergence to occur once accumulation has converged for every pixel.

5 Implementation

This chapter describes the details of the prototype I build to combine probabilistic occlusion culling of particles [IRR+21] with the memory-efficient Acceleration Structure for particle ray tracing of P-k-d trees.

5.1 Project Structure

I build this prototype as an extension of the implementation by Gralka et al. [GWG+20]. They implement a particle ray tracer using several ASs, among which is the P-k-d tree as proposed by Wald et al. [WKJ+15]. Gralka et al. build the prototype using NVIDIA OptiX [PBD+10] (see chapter 3.2) and the *Optix Wrapper Library* by Wald et al. [WMH20] for better ease of use. I extend this prototype by implementing the probabilistic occlusion culling architecture by Ibrahim et al. [IRR+21] using the modifications described (see chapter 4) to optimize it for a ray tracer. My goal is to examine the performance impact of probabilistic occlusion culling on a P-k-d-based particle ray tracer for large particle data.

The program is split into a host- and device side. The host side runs on the CPU and is mainly responsible for preparatory tasks, while the device side (GPU) handles computationally demanding rendering tasks in parallel. The host side consists of the following classes:

ModelViewer is part of the original implementation by Gralka et al. and handles rendering and user interaction with the program like camera manipulation or keyboard input.

Model encapsulates the raw particle data loaded into the program. It holds all particles in a list and provides functionality to retrieve their individual or collective bounds.

OptixParticles handles all buffers necessary for the device side of the program. It is the super class of all AS implementing classes, e.g.

AllPKDParticles which provides the functionality for the construction of the P-k-d tree. This is the only AS I examine as part of this work.

DensityVolume is a class I introduce to hold all the functionality necessary to construct the density field which many of the probabilistic estimations are based on.

FrameState encapsulates information about the current frame, e.g. the amount of frames since accumulation started as well as properties for the probabilistic accumulation algorithm.

The device side consists of the following files:

raygen.cu contains the *RayGen* and *Miss* programs of the OptiX pipeline. The latter of which is empty for this project. *RayGen* implements the orthographic generation of rays using a randomized super sampling pattern and calls the *traceRay* function, which returns a color value per pixel which *RayGen* writes to the frame buffer.

allPKD.cu holds the *Intersection* kernel of the OptiX pipeline. This is where Gralka et al. implement P-k-d traversal and report the closest hit back to *traceRay*. When a ray particle intersection is found, I interrupt P-k-d traversal to perform an iteration of the depth-confidence accumulation. Since an iteration needs to occur for each ray particle intersection, this is the stage at which I implement the accumulation algorithm.

5.2 Density Field

To estimate particle counts inside a pixel's frustum at certain depths, Ibrahim et al. [IRR+21] suggest using a scalar field representing particle density at each point in space. They discretize the field on a coarse uniform grid.

5.2.1 Construction

The particle density grid is constructed offline on the CPU on program startup. The user can specify the number of grid cells via the `-voxel_count` launch parameter.

I construct the density grid by subdividing the model's bounds (i.e. its axis-aligned bounding box). Along the shortest axis I subdivide it into the amount of slices specified by `-voxel_count`. On the remaining axes I choose the number of subdivisions so that the resulting grid cells are kept as cube-like as possible.

Subsequently, I iterate through the model's particles and determine the grid cells each particle intersects. If a particle is completely contained within only one grid cell, I add $\frac{1}{V_{cell}}$ (where V_{cell} is the cell's volume) to the cell's density estimate. If a particle intersects more than one, I splat its density contribution according to the volumetric

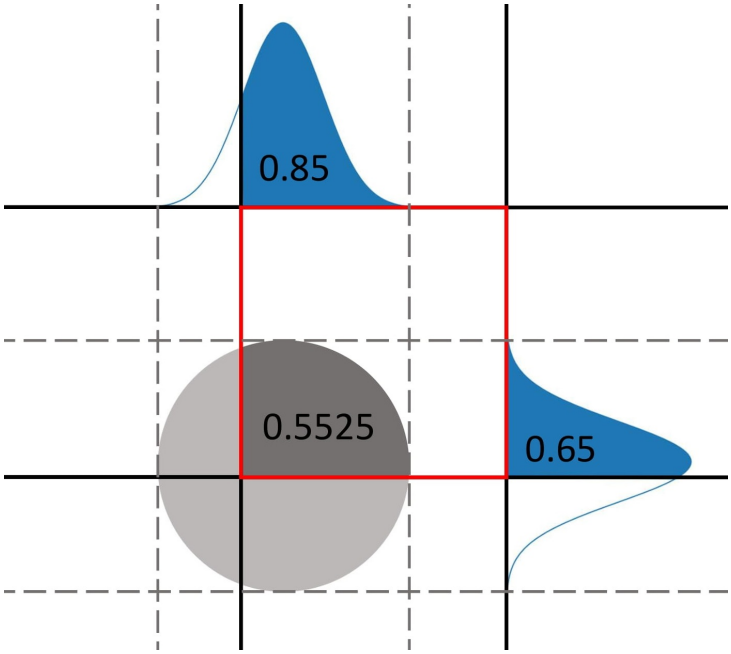

Figure 5.1: 2D schematic of particle splatting: To determine the fraction of the grey particle's volume (area in 2D) inside the current cell (red), I approximate the volume inside the cell's bounds by looking up a pre-computed Gaussian integral (area under blue curves, not to scale) w.r.t. each axis separately. The product of these serves as a good estimate for the fraction of its volume the particle shares with the grid cell. At the vertical overlap, the Gaussian integral reads 0.65 and at the horizontal overlap 0.85. The product $0.65 \cdot 0.85 = 0.5525$ serves as the estimate for the fraction of particle area inside the cell's bounds.

overlap with each of the intersecting grid cells. Per intersecting cell, I approximate cell particle overlap on each axis separately using a pre-computed Gaussian integral and multiply them to retrieve the final contribution of the particle for that cell (see figure 5.1). Since I assume particles to be smaller than grid cells, the number of cells a single particle can intersect cannot exceed eight.

After construction I linearize the 3D density grid and store it in a buffer to supply to the device side of the program, where I traverse it via volume ray casting.

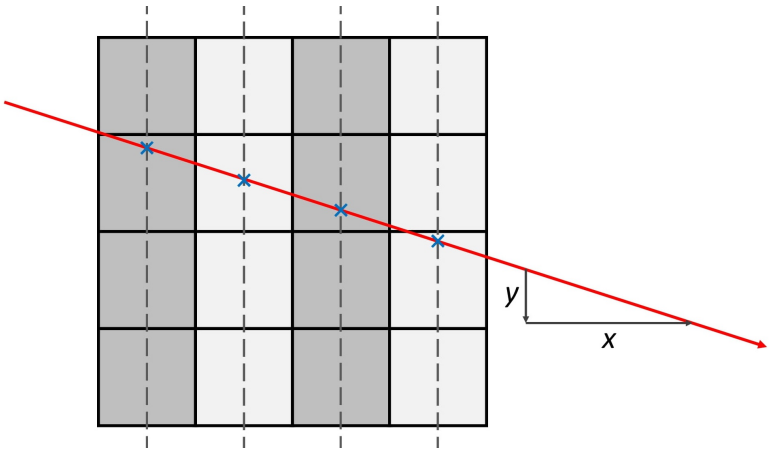

Figure 5.2: 2D schematic of a center ray (red) cast through the particle density grid. As the x-component of the ray's direction vector is largest in absolute terms, I choose sampling points (blue) to be centered on voxel slabs orthogonal to the x-axis. This way, with one sample per slab, the information from the density grid is captured with all its detail without oversampling it.

5.2.2 Traversal

To traverse the particle density field, I follow the original authors' approach and employ a volume ray cast [07]. However, there are some choices to be made when implementing the algorithm:

The first choice is how to define the ray to be cast through the field. It is impractical to naively use the ray I trace through the scenes geometry, as it differs for every iteration of the super sampling process. I want the density histogram obtained through traversal to be representative for the pixel's frustum, which remains constant. Thus I choose the ray to be cast through the density field to always intersect the pixel's footprint on the viewing plane at its center, independent of the sample ray traced through the P-k-d tree. I call this ray the *center ray*.

To obtain the density histogram, I perform a traditional volume ray cast, marching along the center ray and sampling the density volume at equidistant points. Choosing the distance between samples is the second choice to be made. Choosing it too small and oversampling the grid does not lead to any increase in precision in the resulting histogram but costs more memory accesses than necessary, causing a negative impact on performance. On the other hand, undersampling should also be avoided as the density field will then not be captured in full detail, leading to bigger errors in approximations and possibly amplifying visual artifacts. To strike a balance, I choose a sampling distance so that each slab of grid cells along the dimension in which the center ray's direction

vector's component is largest (w.r.t its absolute value) is sampled once at its center (see figure 5.2).

Traditionally in volume ray casting, sampled values are determined by trilinearly interpolating between neighboring cells' entries. As this comes at a cost of additional memory access operations, it might be worth only taking into account the density value of the cell the sample point is contained in. I call these strategies *linear interpolation* and *nearest neighbor* respectively.

In practice, I do not actually store the density histogram. Instead, I compute all evaluations of $D(d_1, d_2)$ necessary for confidence accumulation and computing acceptance probability during the volume ray cast and return them directly.

5.3 Depth-Confidence Accumulation

I implement the depth-confidence accumulation algorithm as part of the OptiX *Intersection* kernel. This is where Gralka et al. implement P-k-d traversal. Their program however, only reports back the closest particle intersection with the ray. To stay as close as possible to the original approach by Ibrahim et al., it is necessary for the accumulation to treat every ray particle intersection as a sample as they take into account every fragment before depth sorting. I achieve this by implementing depth-confidence accumulation functionality directly into P-k-d traversal. If convergence has not yet been reached, I interrupt P-k-d traversal at every ray particle intersection found to perform an iteration of the depth-confidence accumulation algorithm (algorithm 2.1).

5.3.1 Accumulating Confidence

The following section describes the details of how I implement confidence accumulation. Meaning, how do I increase confidence C whenever a sample with sampled depth $d_s < d$ is processed?

In practice, I implement the depth-confidence buffers to not only hold depth d and confidence C but also two additional values k and $E(n(k-1))$. k denotes the sample count including the current one and $E(n(k-1))$ the expected amount of *unique* particles sampled until (but excluding) the current one. I initialize both buffers by $(d = d_{max}, C = 0, k = 1, E(n(k-1)) = 0)$. During accumulation, I use k to compute $E(n(k))$ via the closed form of the recurrence relation suggested by Ibrahim et al. [IRR+21] (eq. 5.1),

$$E(n(k)) := \frac{1 - M^k}{1 - M} \text{ , with } M := 1 - \frac{1}{N} \tag{5.1}$$

where N denotes the estimated number of particles sampled from $D(d_{min}, d_{max})$, which I retrieve from traversing the center-ray through the density field. Using the projected particle footprint a, the expected value of unique particles sampled since the previous iteration $n_k := \Delta E(n(k)) = E(n(k)) - E(n(k-1))$ and the confidence from the previous sample $C(k-1)$, I accumulate this sample's confidence using the second form of the accumulation equation

$$C(k) \approx 1 - \prod_{i=1}^{k}(1-a)^{n_i} \tag{5.2}$$

$$C(k) \approx 1 - (\prod_{i=1}^{k-1}(1-a)^{n_i})(1-a)^{n_k} \tag{5.3}$$

$$C(k) \approx 1 - (1 - C(k-1))(1-a)^{n_k} \tag{5.4}$$

as proposed by Ibrahm et al. As shown, this form comes with the benefit of enabling me to split up the product across multiple samples.

5.3.2 Improving Depth-Confidence Estimates

Merely accumulating confidence is not sufficient. To facilitate effective culling, the representative depth d needs to be as low as possible while still accumulating sufficient confidence. This is where Ibrahim et al. introduce a double-buffered algorithm (algorithm 2.1). The ray tracing context does not necessitate any major changes to the algorithm. Thus my prototype follows their method. They propose the accumulation buffer to be reset (i.e. overwritten with $(d = d_s, C = a_s, k = 1, E(n(k-1)) = 0)$) by new samples with depth $d_s < d$ if it is deemed likely enough that restarting the accumulation from the closer depth d_s will reach sufficient confidence C_s (i.e. $\text{OP}(C_s, d_s) > \text{OP}(C, d)$) within a certain amount of samples N_{budget}. They determine this *acceptance probability* $P(d_s)$ by finding the expected number of closer samples required $k(d_s)$. Thus, $P(d_s)$ denotes the probability that at least $k(d_s)$ of N_{budget} samples have a depth $\leq d_s$. Since this fulfils the definition of a Bernoulli trial, $P(d_s)$ follows a binomial distribution, with $p(d_s)$ being the probability that a randomly sampled particle has a depth $< d_s$:

$$P(d_s) := P(k \geq k(d_s)) = \sum_{k=k(ds)}^{N_{budget}} \binom{N_{budget}}{k} p^k (1-p)^{N_{budget}-k} \tag{5.5}$$

$$\text{with } p = p(d_s) := \frac{D(d_{min}, d_s)}{D(d_{min}, d_{max})} \tag{5.6}$$

Since binomial distributions are very inefficient to compute, they approximate it using a pre-computed discretization of a normal distribution, turning *acceptanceProbability*(d_s) into a simple look-up operation.

5.3.3 Culling via Ray Clipping

Ibrahim et al. originally employ an object-order approach. Therefore, for culling to have any effect on rendering performance, it is necessary to actually remove culled particles from the scene and update the density volume to represent the new scene correctly. With an image-order approach like ray tracing this is not necessary. Once the average confidence $\frac{1}{|K|}\sum_{p\in K} C_{cull}(p)$ from the pixels inside neighborhood kernel K (usally a 3×3 neighborhood kernel) around the pixel in question accumulates to a value greater than or equal to the culling threshold C_{occ}, I deem the maximum depth value inside the kernel $\max_{p\in K}(d_{cull}(p))$ to be a sufficient estimate for the depth behind which particles are occluded. Thus I perform subsequent ray casts with $d_{max} = \max_{p\in K}(d_{cull}(p))$, with d_{max} being the maximum depth at which intersections with the ray are reported during P-k-d traversal. The P-k-d implementation by Gralka et al. [GWG+20] that I use takes d_{max} into account and does not descend into subtrees whose bounding boxes intersect the ray only at depths $> d_{max}$. This way I achieve a performance improvement.

Furthermore, I apply d_{max} to the center ray cast through the density volume. This way, there is no need to update the entries of the density grid after culling to keep it consistent with the updated scene, as the scene does not actually change, but merely the depth until which I traverse it. Thus traversing the density volume to the same depth d_{max} is sufficient. After convergence, d_{max} is assumed constant and new culling decisions (i.e. further decreasing d_{max}) are no longer considered. I examine visual and performance effects of this technique as part of chapter 6.

6 Results

I evaluate my implementation in terms of visual quality of rendered images and rendering performance. I discuss the results and compare them to those of previous work. Table 6.1 lists the particle data sets I use for evaluation. All measurements and renderings are performend on an Intel® Core™ i7 4790 3.6 GHz and an NVIDIA GeForce GTX 970 4GB at a resolution of 800 $\times$ 800 pixels. Unless stated differently, I perform renderings and measurements using parameters as stated in table 6.2. I choose these parameters so that renderings maintain an acceptable visual quality, while keeping results comparable across data sets.

Data set	#particles
Nozzle	1,550,333
Laser Ablation	48,000,000
Fluid	29,999,997
Covid-19	14,566,669

Table 6.1: Data sets used for evaluation and their respective particle count.

Data set	vc	C_{occ}	ci	spp	dq	di	ks
Nozzle	128	0.95	128	4	yes	no	1×1
Laser Ablation	256	0.95	128	4	yes	no	1×1
Fluid	256	0.95	128	4	yes	no	1×1
Covid-19	512	0.95	128	4	yes	no	3×3

Table 6.2: Parameters used for evaluation per data set. *vc*: density grid voxel count along shortest axis, C_{occ}: occlusion confidence threshold, *ci*: iterations since last culling decision until accumulation is assumed to be converged, *spp*: samples (rays cast) per pixel per frame, *dq*: depth quantisation, *di*: density interpolation, *ks*: kernel size.

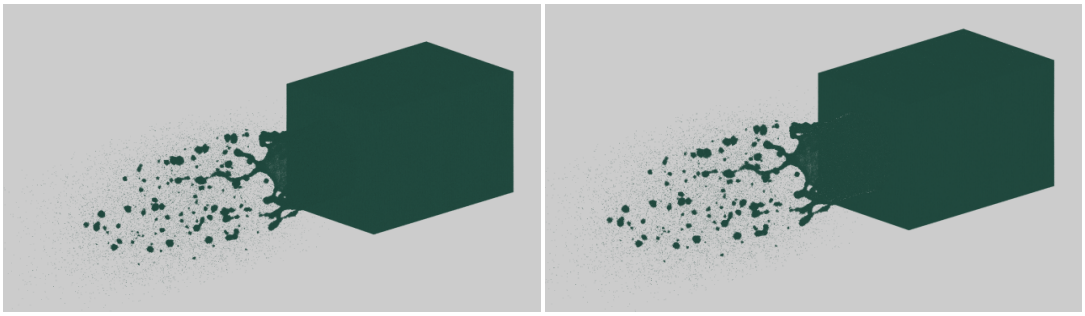

Figure 6.1: Comparison of renderings of the *Laser Ablation* data set (View 2): without culling (Left), after culling convergence using default parameters as stated in table 6.2 (Right). Slight artifacts are visible around the edge of the crown.

6.1 Visual Results

Ideally, culling does not alter the visual appearance of the rendered image, as only those particles should get culled that are occluded by others and thus do not contribute to the pixel's color displayed on screen. Figure 6.1 shows a comparison between renderings of the *Laser Ablation* data set both with and without probabilistic culling enabled. For this data set, most culling takes place on the right hand side of the image, on the solid aluminium block made of several tens of millions of particles, while for pixels that map to the crown, which consists of few, spread out particles, confidence does not accumulate high enough. My prototype renders data sets like this, with a low amount of high-frequent detail where culling takes place, quite well without the need for any conservative measures.

Due to the probabilistic nature of the method however, it is possible for some of the per-pixel approximations to be erroneous leading to visual artifacts. It turns out that data sets containing visible high-frequent details are most susceptible to such artifacts. In the following, I describe where these artifacts appear within renderings produced by my implementation, explain their origin, how various parameters affect them and what measures I take to avoid them.

Visual artifacts originate from an overestimation of confidence for depths, behind which lie visible particles that significantly affect the pixel's color. If the accumulated confidence surpasses the threshold value C_{occ}, subsequent rays get clipped at depth d_{cull} and intersections with particles behind d_{cull} are no longer reported. Pixels, whose frustums contain visible particles at largely differing depths are especially susceptible to this phenomenon, e.g. pixels whose frustums intersect a dense structure's surface at a small angle or contain a dense structure partially occluded by another one. These artifacts become even more exaggerated when details within the data set are too high-frequent to be sufficiently captured by the density grid. Probability estimates highly depend on

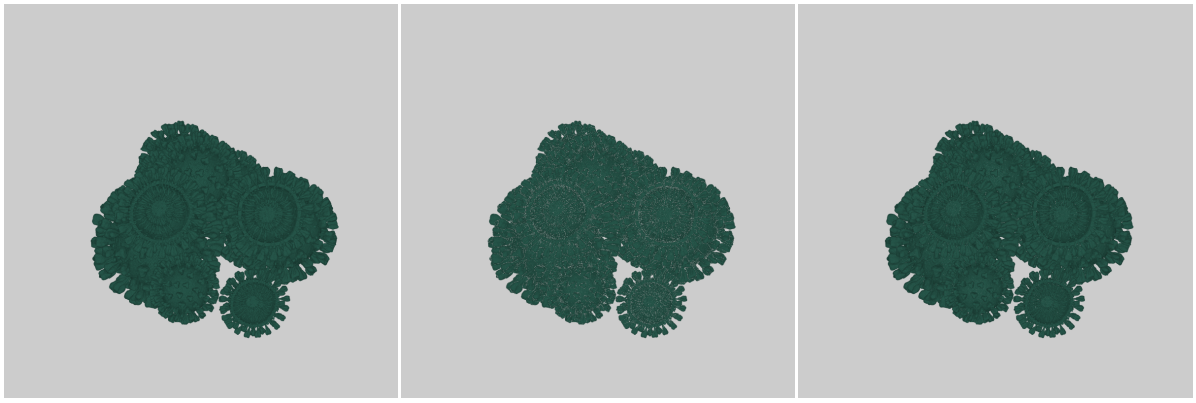

Figure 6.2: Comparison of renderings of the *Covid-19* data set (View 1): without culling (Left); after culling convergence, without applying a kernel for culling decisions (Center), after culling convergence, taking surrounding pixels into account within a 3×3 kernel (Right). Strong visual artifacts are visible without neighborhood-based culling decisions. A 3×3 kernel improves visual quality significantly.

density samples from the grid and if they do not properly capture structures within the data, these structures are also poorly represented by depth-confidence estimates, possibly leading to an overestimation of confidence, erroneous culling and thus visual artifacts. This is especially prevalent in renderings of the *Covid-19* data set (see figure 6.2), where spikes on the surface of the virus are not captured well enough by the discretized density field. Artifacts of the background color are visible around the spikes' edges, since rays get clipped too closely that would intersect the surface of the virus at a greater depth. The strength of artifacts visibly decreases with an increase in the density grid's resolution, but even at high resolutions ($\geq 512^3$ voxels), it does not sufficiently depict these high-frequent details. Using trilinear interpolation during density grid traversal can reduce the magnitude of some artifacts, but if the detail frequency is too high, artifacts will be visible nonetheless.

I neither observe such artifacts in images rendered by Gralka et al.'s implementation without culling nor in those I generated by using the prototype by Ibrahim et al. Thus I conclude that the artifacts visible originate from the per-pixel nature of my adapted approach and the fact that, without neighborhood-based culling decisions, erroneous approximations arising from the probabilistic nature of the method will not be mitigated.

Therefore, I employ two more methods, other than increasing the resolution of the particle density grid, to keep depth estimates more conservative and reduce artifact visibility. Namely, these are depth quantisation (chapter 4.2) and making culling decisions based on additional depth-confidence values from neighboring pixels (chapter 5.3.3). While the effect of depth quantisation is subtle, applying even a small 3×3 kernel to average

Data Set	*Nozzle*				*Laser Ablation*			
View	1	2	3	4	1	2	3	4
Avg. FPS								
nc	27.2	15.8	24.8	21.7	18.9	21.4	18.5	22.4
bc	9.9	10.2	10.0	11.4	14.1	9.6	6.8	16.8
ac	27.8	16.4	25.5	21.9	26.5	21.9	26.4	30.9

Data Set	*Fluid*				*Covid-19*			
View	1	2	3	4	1	2	3	4
Avg. FPS								
nc	15.4	23.1	11.0	10.8	6.4	8.8	6.4	9.6
bc	6.6	10.0	4.7	4.6	2.3	2.9	2.6	4.4
ac	16.1	28.2	11.1	11.1	6.4	9.2	6.4	9.8

Table 6.3: Performance results across all examined data sets at different views. Values given in average frames per second over the respective period (*nc*: no culling, *bc*: before convergence, *ac*: after convergence). Marked views are axis-aligned.

out confidence misjudgements and keep the representative depth conservative makes a very noticeable difference and removes artifacts almost completely (see figure 6.2), even though this measure compromises performance which the following section describes in detail.

6.2 Performance Results

Table 6.3 shows performance measurements of my implementation across all data sets at different viewing angles.

In most cases, it is clear that after accumulation convergence, culling particles using clipped rays yields a performance increase compared to rendering data sets without any culling. As Ibrahim et al. do, I observe a drop in performance during accumulation (before convergence). Whether or not the additional computational cost during accumulation is worth the benefit after convergence, heavily depends on the data set, the total number of samples and the viewing angle. I observe that large data sets (e.g. *Laser Ablation*) yield performance increases after accumulation convergence of up to 30%-40% compared to the rendering performance without culling, making up for the accumulation performance cost within a relatively low number of samples. The benefit for small data

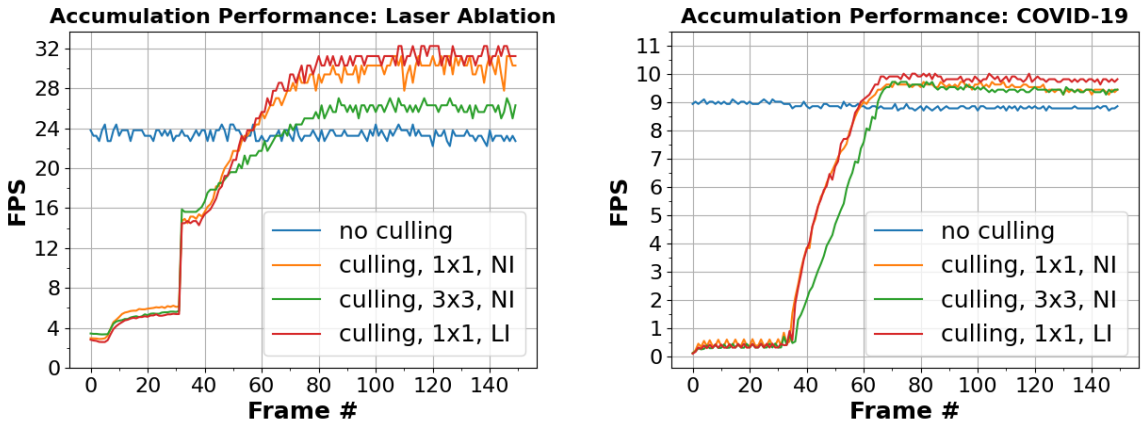

Figure 6.3: Accumulation performance plots for *Laser Ablation*, View 4 (Left) and *Covid-19*, View 2 (Right). I compare performance without culling (blue) to performance with culling using single-pixel-based (orange) and 3×3-neighborhood-based (green) culling decisions, further differentiated between using trilinear density volume interpolation (LI) and using the nearest-neighbor strategy (NI). Convergence occurs around frame 60.

sets (e.g. *Nozzle*) however, is significantly lower. P-k-d traversal is already quite fast in these cases and the small performance improvement reached after convergence likely does not make up the overhead cost during accumulation. Figure 6.3 plots the prototype's performance for two exemplary data sets (*Laser Ablation* and *Covid-19*) across the entire accumulation process and compares it to the baseline: rendering the data set without culling. Against my initial hypothesis, the negative impact of trilinear interpolation during the accumulation process is negligible. However, the resulting depth-confidence estimates seem to allow for the culling of more particles than using the nearest-neighbor strategy, leading to a slight performance improvement after convergence when compared to the post-convergence performance without interpolation.

As mentioned above (chapter 6.1), for data sets with a large amount of high-frequent detail (e.g. *Covid-19*), conservative measures may become necessary to reduce visual artifacts to an acceptable level. The most effective of these, namely increasing the density grid's resolution and applying a neighborhood kernel for culling decisions, come at a significant performance or memory cost. Higher density grid resolutions not only require more GPU memory, but also have to be sampled more often (see chapter 5.2.2) to make use of the additional detail, causing more memory accesses and thus a negative impact on performance. Neighborhood-based culling decisions also come at a performance cost during accumulation (since more memory accesses are necessary) and may result in more conservatively clipped rays and thus less effective culling, potentially having a negative impact on performance after convergence (see figure 6.3).

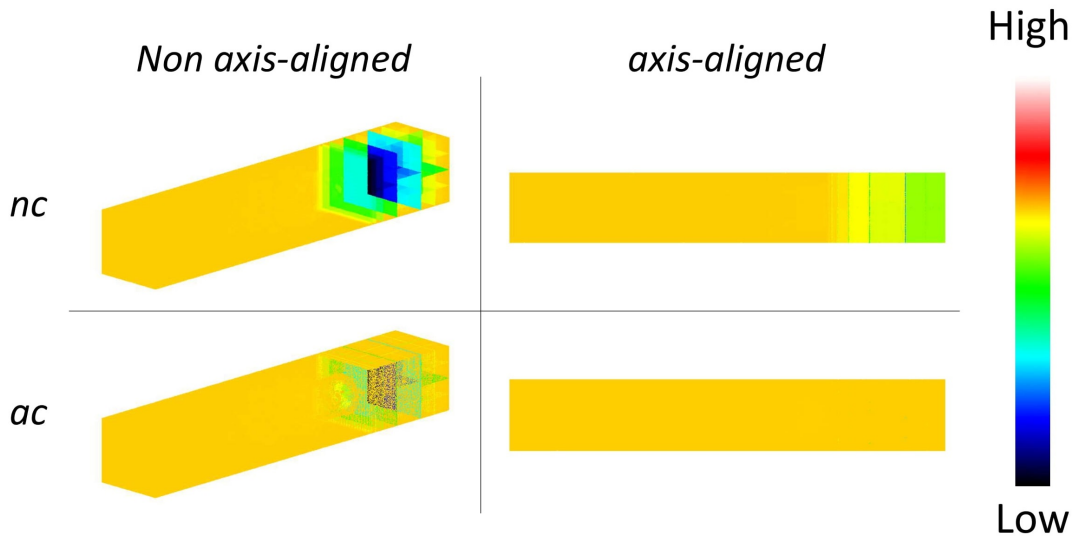

Figure 6.4: Amount of particles in subtrees skipped during P-k-d traversal per pixel for an axis-aligned and a non axis-aligned view of the *Laser Ablation* data set, both without culling (*nc*) and after culling convergence (*ac*). Colors indicate the amount of particles skipped during P-k-d traversal of a ray cast through the respective pixel.

Furthermore, I observe a significantly larger performance increase after accumulation for views which are axis aligned (i.e. all rays cast parallel to one of the three axes). This is behaviour is quite noteworthy as Ibrahim et al. do not observe this with their prototype. I attribute this observation to the nature of P-k-d trees and their axis-parallel splitting planes. The traversal of a ray through the P-k-d tree occurs in a top down fashion. The algorithm thereby only descends into those subtrees whose bounding volumes intersect the ray. Skipping (i.e. not descending into) subtrees containing a large amount of particles significantly reduces traversal time as less intersection checks have to be performed. Figure 6.4 shows how many particles lie within subtrees which do not have to be traversed during P-k-d traversal of a ray cast through the respective pixel, i.e. how many particles can be skipped during P-k-d traversal. Yellow and reddish hues indicate a high amount of particles skipped during P-k-d traversal, while cyan and blueish hues signify that large parts of the P-k-d tree still have to be traversed for rays cast from that pixel. What becomes visible are splitting planes separating high-level (i.e. close to the root) nodes from one another. For rays intersecting these, neither side of the splitting plane can be disregarded during traversal, leading to a lower amount of skipped particles (and thus longer traversal times) than for neighboring rays, that do not intersect the splitting plane. Low-level splitting planes are not visible as the nodes they separate only contain a small number of particles. With axis-aligned views into the scene, rays are cast parallel to one of the three axes. As P-k-d splitting planes are also axis

aligned, many of them lie parallel to the rays cast, significantly decreasing the amount of pixels whose rays intersect a large number of them (i.e. for which P-k-d traversal has a large negative performance impact). Clipping of rays of course, has a similar effect. It turns out however, that if confidence does not accumulate high enough and some pixels remain whose rays cause high traversal times, this has a significant impact on performance after convergence. I attribute this to the parallel processing nature of GPUs. Since P-k-d traversal is executed concurrently for all pixels, each frame's processing time is bounded below by the last pixel's thread to finish. Thus performance might not significantly improve if there are even just few pixels left whose rays require substantially more time to traverse the P-k-d tree. I observe this with most non axis-aligned views.

7 Summary and Discussion

I contribute by building on the work by Ibrahim et al. [IRR+21], suggesting adaptions to their probabilistic occlusion culling approach that prepare it for the application within a P-k-d tree-based [WKJ+15] particle ray tracer. My suggested adaptions aim to make use of the ray tracing model and the P-k-d tree's structure to facilitate a performance improvement of the super sampling process by culling occluded particles from the scene, disregarding them during P-k-d traversal. To achieve this, I clip rays at depths behind which the probabilistic method by Ibrahim et al. would confidently classify particles as occluded. In contrast to their approach, I suggest to perform culling on a per-pixel, instead of a per-particle level, adapting it to the image-order nature of a ray tracer and enabling pixel-precise culling. I go on to examine the consequences of my adaptions and how to mitigate them. I implement the adapted method into a particle ray tracer by Gralka et al. [GWG+20] and evaluate its effect on visual quality and performance. Results show that in certain cases my approach yields a significant performance improvement compared to super sampling particle data without culling. The benefit measured, however, largely depends on the size of the data set and the alignment of the view with the orientation of the P-k-d tree. There are also cases in which visual artifacts arise from wrongful culling of particles, especially on particle data with high-frequent detail, unable to be captured by the discretization of the density field. These are factors that pose a significant restriction on the method's use case, requiring future work or alternative approaches to be lifted.

The conservative measures I introduce to mitigate visual artifacts do not come without their drawbacks in terms of performance. Future work might explore different solutions with a lower computational demand than e.g. neighborhood-based culling decisions which increase the amount of required memory accesses. One such solution might be to perform depth-confidence accumulation on a coarser scale altogether, or to explore the use of a depth-confidence pyramid to base culling decisions on.

Alternatively, it might also be viable to build the particle density histogram not based on a discretized density field as Ibrahim et al. do, but solely on reported samples from AS traversal.

With this work, I adapt probabilistic occlusion culling as proposed by Ibrahim et al. [IRR+21] to the application within a P-k-d-based ray tracer. Part of the motivation for

using the ray tracing model is its aptness to global illumination effects, like shadows or ambient occlusion, through the use of secondary rays. These effects can greatly improve the visual quality of rendered images as well as the intuitive understanding of depicted three-dimensional structures. It is out of the scope of this work to investigate how the traversal of secondary rays could benefit from probabilistic occlusion culling. Future work in this direction might examine approaches like light source-based depth-confidence maps or such that are oriented along the three main axes.

Bibliography

[07] "Visualization of 3D Scalar Fields." In: *GPU-Based Interactive Visualization Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 11–79. ISBN: 978-3-540-33263-3. DOI: 10.1007/978-3-540-33263-3_2. URL: https://doi.org/10.1007/978-3-540-33263-3_2 (cit. on p. 34).

[Chi12] H. Childs. "VisIt: An end-user tool for visualizing and analyzing very large data." In: (2012) (cit. on p. 25).

[CSI09] D. Cha, S. Son, I. Ihm. "GPU-assisted high quality particle rendering." In: *Computer Graphics Forum*. Vol. 28. 4. Wiley Online Library. 2009, pp. 1247–1255 (cit. on p. 25).

[FVV+96] J. D. Foley, F. D. Van, A. Van Dam, S. K. Feiner, J. F. Hughes, J. Hughes. *Computer graphics: principles and practice*. Vol. 12110. Addison-Wesley Professional, 1996 (cit. on p. 17).

[GKM+15] S. Grottel, M. Krone, C. Müller, G. Reina, T. Ertl. "MegaMol—A Prototyping Framework for Particle-Based Visualization." In: *IEEE Transactions on Visualization and Computer Graphics* 21.2 (2015), pp. 201–214. DOI: 10.1109/TVCG.2014.2350479 (cit. on p. 25).

[Gla89] A. S. Glassner. *An introduction to ray tracing*. Morgan Kaufmann, 1989 (cit. on p. 17).

[GWG+20] P. Gralka, I. Wald, S. Geringer, G. Reina, T. Ertl. "Spatial partitioning strategies for memory-efficient ray tracing of particles." In: *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE. 2020, pp. 42–52 (cit. on pp. 3, 15, 16, 19, 25, 31, 37, 47).

[HA90] P. Haeberli, K. Akeley. "The accumulation buffer: Hardware support for high-quality rendering." In: *ACM SIGGRAPH computer graphics* 24.4 (1990), pp. 309–318 (cit. on p. 26).

[HDS96] W. Humphrey, A. Dalke, K. Schulten. "VMD: visual molecular dynamics." In: *Journal of molecular graphics* 14.1 (1996), pp. 33–38 (cit. on p. 25).

[HH11] M. Hapala, V. Havran. "Review: Kd-tree Traversal Algorithms for Ray Tracing." In: *Computer Graphics Forum* 30.1 (2011), pp. 199–213. DOI: https://doi.org/10.1111/j.1467-8659.2010.01844.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2010.01844.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2010.01844.x (cit. on p. 19).

[IRR+21] M. Ibrahim, P. Rautek, G. Reina, M. Agus, M. Hadwiger. "Probabilistic Occlusion Culling using Confidence Maps for High-Quality Rendering of Large Particle Data." In: *IEEE Transactions on Visualization and Computer Graphics* 28.1 (2021), pp. 573–582 (cit. on pp. 3, 15, 16, 20, 23, 27, 31, 32, 35, 47).

[KM05] A. E. Kaufman, K. Mueller. "Overview of volume rendering." In: *The visualization handbook* 7 (2005), pp. 127–174 (cit. on p. 25).

[KRZ+17] M. Krone, G. Reina, S. Zahn, T. Tremel, C. Bahnmüller, T. Ertl. "Implicit sphere shadow maps." In: *2017 IEEE Pacific Visualization Symposium (PacificVis)*. IEEE. 2017, pp. 275–279 (cit. on p. 25).

[LPSV14] M. Le Muzic, J. Parulek, A.-K. Stavrum, I. Viola. "Illustrative visualization of molecular reactions using omniscient intelligence and passive agents." In: *Computer Graphics Forum*. Vol. 33. 3. Wiley Online Library. 2014, pp. 141–150 (cit. on p. 25).

[PBD+10] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, M. Stich. "OptiX: A General Purpose Ray Tracing Engine." In: *ACM Trans. Graph.* 29.4 (July 2010). ISSN: 0730-0301. DOI: 10.1145/1778765.1778803. URL: https://doi.org/10.1145/1778765.1778803 (cit. on pp. 16, 25, 26, 31).

[SGG15] J. Staib, S. Grottel, S. Gumhold. "Visualization of particle-based data with transparency and ambient occlusion." In: *Computer Graphics Forum*. Vol. 34. 3. Wiley Online Library. 2015, pp. 151–160 (cit. on p. 25).

[TS+05] N. Thrane, L. O. Simonsen, et al. "A comparison of acceleration structures for GPU assisted ray tracing." In: (2005) (cit. on p. 18).

[WJA+17] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, P. Navratil. "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization." In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (2017), pp. 931–940. DOI: 10.1109/TVCG.2016.2599041 (cit. on p. 25).

[WKJ+15] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, M. E. Papka. “CPU ray tracing large particle data with balanced Pkd trees.” In: *2015 IEEE Scientific Visualization Conference (SciVis)*. IEEE. 2015, pp. 57–64 (cit. on pp. 3, 15, 19, 20, 23, 25, 27, 31, 47).

[WMH20] I. Wald, N. Morrical, E. Haines. *OWL: A Node Graph "Wrapper" Library for OptiX 7*. 2020. URL: https://owl-project.github.io/ (cit. on p. 31).

[ZTH03] P. Zemcik, P. Tisnovsky, A. Herout. “Particle Rendering Pipeline.” In: *Proceedings of the 19th Spring Conference on Computer Graphics*. SCCG ’03. Budmerice, Slovakia: Association for Computing Machinery, 2003, pp. 165–170. ISBN: 158113861X. DOI: 10.1145/984952.984979. URL: https://doi.org/10.1145/984952.984979 (cit. on p. 25).

All links were last followed on May 18, 2022.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature