# Ray Queries On Raw Point Clouds

Balthasar Teuscher[1], Paul Walther[1], Kwasi Nyarko Poku-Agyemang[1], Martin Werner[1]

[1] Technical University of Munich, Germany; TUM School of Engineering and Design, Department of Aerospace and Geodesy, Professorship of Big Geospatial Data Management - (balthasar.teuscher, paul.walther, kwasi.poku, martin.werner)@tum.de

**Keywords:** Point Cloud, Ray Tracing, Query Semantics.

### Abstract

Retrieving information from point clouds for analysis and visualization has gained ever-increasing interest. A growing niche in this regard is ray queries, commonly used for image synthesis. Ray tracing is widely used in computer graphics, with a multitude of solutions based on bounding volume hierarchies. However, these solutions are rarely straightforward to integrate with raw point cloud data and geospatial analytical workflows. To overcome this, we present a novel approach to ray tracing in raw point clouds that builds upon and extends existing geospatial indices. The solution is exemplified by a fast octree implementation that supports versatile query semantics, such as neighborhood queries with constraints on $k$ and radius for both points and rays, while offering configurable data organization schemes, including layered, fixed, and adaptive depth. The evaluation demonstrates satisfactory speed and capabilities for many scientific use cases, while simultaneously exhibiting low implementation costs, high flexibility, and simplicity in integrating ray tracing into analytical point cloud workflows.

## 1. Introduction

Ray tracing and ray marching approaches have a long history in computer graphics for image synthesis (Glassner, 1989). Recently, such traditional visualization methods have gained renewed interest by combining and integrating them with neural approaches (Mildenhall et al., 2022, Kerbl et al., 2023, Chang et al., 2023). Apart from image synthesis, numerous analytical approaches from photogrammetry, remote sensing, and adjacent fields rely on ray queries. For example, generating visibility maps (Cho and Forsyth, 1999), evaluating spatial radio signal coverage (Vitucci et al., 2015), detecting changes in spatiotemporal surveying (Anders et al., 2020), skyline extraction (Stojanovic and Stojanovic, 2014), modeling solar energy for urban planning (Kosmopoulos et al., 2024), or geometric point in polygon tests (Laass, 2021) among others.

While a multitude of established ray tracing solutions exist in the field of computer graphics, such as Embree (Wald et al., 2014), transferring these solutions to the geospatial context is challenging. Issues arise from subpar scalability to larger geographic contexts, interoperability with geospatial tooling, and their focus on triangles and meshes, thereby neglecting raw point clouds. On the other hand, tooling from the geospatial domain often has limited support for ray tracing. For example, the Point Cloud Library (PCL) only supports querying the nearest voxel hit by a ray (Rusu and Cousins, 2011), whereas the Open3D library only supports ray casting on meshes that can be loaded into memory (Zhou et al., 2018). However, converting raw point clouds to meshes is not always desirable or even possible.

Raytracing can be accelerated with hierarchical indexing structures, such as the Bounding Volume Hierarchy (BVH) (Meister et al., 2021). Unfortunately, implementations from the field of computer graphics often fail to efficiently handle raw point clouds and lack support for neighbourhood queries. Another widely used index structure for analyzing and visualizing point clouds is the Octree (Meagher, 1982, Gobbetti and Marton, 2004). While they provide range and neighborhood query functionality, they often lack supporting ray queries. A more specialized indexing solution, specifically designed for ray tracing raw point clouds, is *HashPoint*, which hashes the points to pixels in the image plane of a given camera position (Ma et al., 2024). However, this requires transforming the entire dataset into the camera coordinate system. Similarly, the approach from *PointerSect* employs a brute-force method using a grid index and custom CUDA kernels (Chang et al., 2023). Unfortunately, these approaches do not scale well to larger datasets, multiple views, and more generic ray queries.

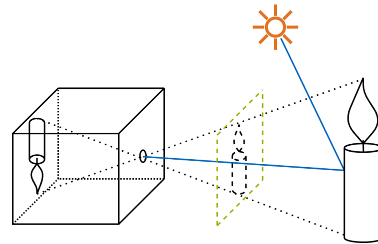


Figure 1. Ray tracing illustration with a pinhole camera (left), the image plane (green), and a ray (blue) starting from the pinhole, reflected on the object (candle) towards a light source (sun).

Our main contribution is a generic approach that extends common spatial indexing structures to support ray queries on raw point clouds. This is exemplified by a concrete implementation based on an Octree, evaluated on real-world photogrammetric point cloud data. Furthermore, we present an approach for extending a database system with user-defined functions to support out-of-core ray queries on large geospatial point cloud datasets.

## 2. Ray Tracing

Ray tracing is an image synthesis method that models the light, respectively, rays captured by a camera (Figure 1). Since it is based on the real-world physical process of image generation, it

is predestined for high-quality, realistic rendering, including the accurate representation of lighting, shadows, lenses, and material properties. This section provides a brief introduction to the core concepts of camera, ray, and hit, highlighting key aspects relevant to the remainder of this work.

**Camera.** The pinhole camera is a very basic concept of how a camera works, where the pinhole is the focal point through which the rays are captured on the image plane. It is often formalized through an extrinsic and intrinsic matrix. The extrinsic matrix defines the camera's position and orientation in the real world, whereas the intrinsic matrix defines its internal properties, such as focal length, which determine how objects are projected onto the image plane. This information is used to generate the rays to trace.

**Ray.** The concept of a ray models the path of light in a scene that gets captured by a camera. For image synthesis with ray tracing, the path of light is reversed, traveling from the camera towards the scene to be captured. In its basic form, the rays feature an *origin* $\mathbf{o}$ represented as a point in space and a *direction* represented by a (unit) vector $\hat{\mathbf{d}}$ in space (Figure 2). The time $t$ in the context of a ray measures the distance of a point on the ray to $\mathbf{o}$. Consequently, a segment $s$ of the ray can be specified by using a time interval $s = [t_{min}, t_{max}]$. This allows us to translate the ray segment into a line $L_s = (\mathbf{p_1}, \mathbf{p_2})$ by calculating the start and end point as $(\mathbf{o} + t_{min}\hat{\mathbf{d}}, \mathbf{o} + t_{max}\hat{\mathbf{d}})$. Given that the direction is normalized to the unit length, for example, in the Euclidean space, the time values conveniently correspond to the distance along the ray.

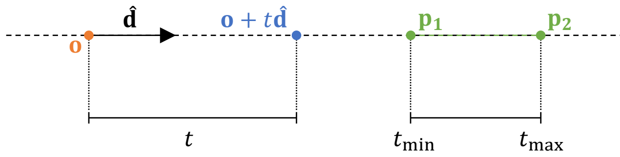



Figure 2. Conceptualization of a ray, its origin $\mathbf{o}$, direction $\hat{\mathbf{d}}$, expression of points along the ray with time $t$, and segments with a time interval $[t_{min}, t_{max}]$.

**Hit.** When tracing a ray for image synthesis, the main point of interest is where the ray hits, respectively, interacts with an object. The goal here is to retrieve an intersection point and normal, along with other features such as material properties, including color. This is used to derive the outward direction of the ray based on reflection or refraction of the material and normal, and to aggregate color, light, and shadow information. While extracting an intersection point from objects represented as meshes with surfaces is theoretically well-defined, numerical issues can arise. On the other hand, extracting a surface and an intersection point from a set of raw points is not well-defined, though it is more robust (Adamson and Alexa, 2003). Moreover, it has been demonstrated that neural approaches can infer it from raw points; however, they still rely on meshes for training (Chang et al., 2023).

## 3. Ray Queries

Retrieving points from raw point clouds presents certain challenges. The most obvious is that points have zero volume, and thus the probability of a random ray actually hitting a point is close to zero. Hence, the points often represent a tiny, but non-empty volume of homogeneous material, often a sphere, cube, or splat. However, this poses a scalability problem as the volumes are fixed, and adapting it requires a complete reconstruction of the indexing structure from scratch. Alternative approaches include buffering the ray predicate or using a nearest-neighbor search. In this work, we build on the latter by relying on distance metrics and half-planes to assign volumes to queries.

**Algorithm 1** Nearest Neighbor Ray Query.

```
procedure RAYQUERY(R, t_min, t_max, k, r)
    k ← inf                              ▷ Number of nearest neighbors if not set
    r ← inf                              ▷ Serach radius if not set
    t_min ← − inf                        ▷ Start of time interval if not set
    t_max ← inf                          ▷ End of time interval if not set

    N ← []                               ▷ Neighboring points (ordered)
    C ← [root]                           ▷ Candidate index nodes

    while C is not empty do              ▷ Process candidates
        node ← pop first element of C
        d_node ← node min distance to ray (orthogonal) R
        [t_near, t_far] ← time interval of projected bounds

        if d_node > r ∨ t_near > t_max ∨ t_far < t_min then
            continue
        end if

        for all p ∈ node do              ▷ Evaluate node points
            d_point ← point min distance to ray (orthogonal) R
            t_p ← time from origin to projected point
            if d_point > r ∨ t_p < t_min ∨ t_p > t_max then
                continue
            end if
            insert p into N              ▷ Ordered by distance or time ①
            if Length(N) > k then
                remove last element of N
                update r or t_max        ▷ From max in N ②
            end if
        end for

        for all child ∈ node do          ▷ Descend to child nodes
            add child to C
        end for
    end while
    return N
end procedure
```

In the Algorithm 1, we present an approach for ray queries on raw point clouds. It works on top of any bounding volume indexing structure, such as R-Tree, Kd-tree, or Octree, that allows traversing from the root node to the leaves. As input, a ray is required, and optionally, it can be further parameterized with the number of neighbors $k$ to find, a search radius $r$, and a time interval $[t_{min}, t_{max}]$ for the ray. When traversing the nodes, each node is first checked to see whether it can be pruned using its bounds. Then, the node's points are evaluated to determine if they can become neighboring points before its child nodes are included as candidates for further traversal.

### 3.1 Query Modalities

The presented nearest neighbor ray query algorithm supports several query modalities that can be induced by specifying the parameters $k$, its distance measure (distance to ray vs. distance to origin), radius $r$, time interval $[t_{min}, t_{max}]$, and combinations thereof, forming a logical conjunction.

**$k$ Nearest Neighbors (kNN).** Queries solely parameterized by $k$ are expected to return at most $k$ nearest points with respect to the chosen distance measure. The distance of a point to the ray can be measured as the distance of the point to its orthogonal projection on the ray, or the distance of the projection to the ray origin, respectively, its time value $t_p$. The latter is especially interesting when the goal of retrieving points is to extract the primary point surface. The kNN query is visualized in Figure 3 (left).

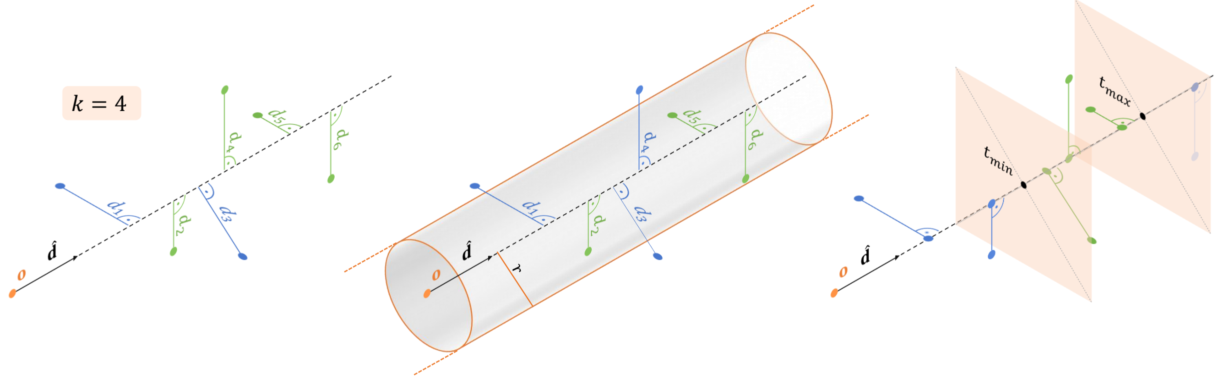



Figure 3. Illustration of various ray query modalities with selected points highlighted in green: $k$ Nearest Neighbors query for $k = 4$ (left), radius query with radius $r$ (middle), and time range query between $t_{min}$ and $t_{max}$ (right).

**Radius Queries.** Queries solely parameterized by the radius return all points intersecting the infinite cylinder with the ray as its centerline. As such, it is equal to buffering the ray and testing for intersection. Considering multiple connected ray segments, for example, following the ray along several reflections and refractions, it becomes clear that this resembles profiling a trajectory with a given buffer. This approach is commonly used in change-detection and analysis workflows for 4D spatiotemporal lidar data (Anders et al., 2020). The radius query is visualized in Figure 3 (middle).

**Time Range.** Querying with a time interval restricts the search space to the points between the two half planes going through the points on the ray at $t_{min}$ and/or $t_{max}$ with the ray direction being its normal. This allows slicing the point cloud with arbitrary parallel planes. Furthermore, the capability to express planes and the signed distance of points to them also unlocks the expression of view-frustum queries. The time range query is visualized in Figure 3 (right).

**Ray Marching.** In ray marching, one tries to find the closest point to the origin within a given distance to the ray iteratively by advancing the center along the query ray by the distance to the closest point. While this is not directly supported by the proposed ray query algorithm, it can be emulated with kNN queries, where $k = 1$, and the distance of the closest point to the ray origin serves as the distance measure. However, it is more efficient to constrain the query with a radius to directly find the closest point.

### 3.2 Distance Predicates

To evaluate whether a point belongs to the ray query result set basically involves testing two distance predicates, namely, whether the distance to the line is within the given radius $r$, and whether the time interval $[t_{min}, t_{max}]$ contains the time value of its projection on the ray $t_p$. An alternative approach to calculating $t_p$ is to use the distance to the plane defined by the ray's origin and direction.

**Point to Line.** Given two points on the ray, the distance $d_l$ of a point orthogonal to the ray can be calculated with

$$d_l = \frac{\|(\mathbf{p} - \mathbf{a}) \times (\mathbf{p} - \mathbf{b})\|}{\|\mathbf{b} - \mathbf{a}\|} \tag{1}$$

where $\mathbf{p}$ is the point. For $\mathbf{a}$, we can use the ray origin, and for $\mathbf{b}$, the ray origin plus a factor times the ray direction (e.g., $\mathbf{b} = \mathbf{o} + 100.0\mathbf{d}$).

**Point to Plane.** Given a plane defined by a point on the plane and a normal vector, the signed distance of a point $p$ to the plane can be calculated by

$$d_p = \hat{\mathbf{n}} \cdot (\mathbf{p} - \mathbf{p_o}) \tag{2}$$

where $\mathbf{p_o}$ is a point on the plane and $\hat{\mathbf{n}}$ the normalized plane normal. When substituting the ray origin $o$ for $\mathbf{p_o}$ and the ray direction $\hat{\mathbf{d}}$ for $\hat{\mathbf{n}}$, the resulting distance is equivalent to the time value $t_p$ of the point's projection on the ray.

### 3.3 Pruning Mechanisms

Several interconnected variables that parameterize ray queries are suitable for pruning nodes and, by extension, subgraphs of the tree. These are the number of nearest neighbors $k$, the time interval of the ray $[t_{min}, t_{max}]$, and the radius $r$, in case they are specified. The radius is most comprehensible and is calculated as the minimum distance between the node bounds and the ray. The approaches are visualized in Figure 4.

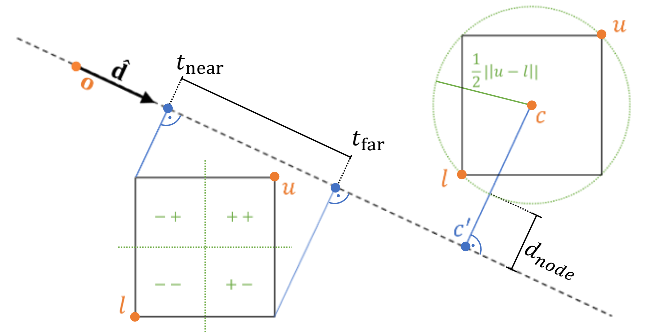



Figure 4. Node bounds pruning mechanisms: projected time interval (left) and ray minimum distance (right).

**Ray Minimum Distance.** For the node bounds, an upper limit of the shortest distance to the ray can be calculated by subtracting half the length of the diagonal from the length of the distance of the center $\mathbf{c}$ to its projection on the ray $\mathbf{c}'$ with

$$d_{node} = \|(\mathbf{c}' - \mathbf{c})\| - \frac{1}{2} \cdot \|\mathbf{u} - \mathbf{l}\| \tag{3}$$

where $\mathbf{u}$ are the upper and $\mathbf{l}$ the lower bounds of the node. An accurate measure of the shortest distance between an axis-

aligned cube and a ray is the smallest path between the nearest edge and the ray, which is more involved.

**Projected Time Interval.** The time interval of a node is calculated as the projection of its nearest $\mathbf{p_n}$ and farthest $\mathbf{p_f}$ corners in the direction of the ray. These corner points can be constructed component-wise based on the signum of the direction components, selecting the lower component when positive, and the upper component otherwise, for the nearest corner. Conversely, the farthest corner is selected by the upper component when it is positive, and by the lower component otherwise. The interval is finally calculated with

$$[t_{near}, t_{far}] = [\|\mathbf{p'_n} - \mathbf{o}\|, \|\mathbf{p'_f} - \mathbf{o}\|] \tag{4}$$

where $\mathbf{o}$ is the ray origin and $\mathbf{p'_n}$ and $\mathbf{p'_f}$ are the projections of $\mathbf{p_n}$ and $\mathbf{p_f}$ on the ray. If the projected interval of a node is disjoint from the query time interval, the node and all its descendants can be pruned. An alternative approach to calculating the time intervals is to evaluate the minimum and maximum distances of the bounds to the plane defined by the ray's origin and direction using Equation 2.

**Dynamic $k$ Distance Filter.** The number of neighbours $k$ determines the upper bounds on the search space from the point at which the specified number of potential candidates has been found. Here we assume keeping track of the evaluated candidates by inserting them into an ordered list with a max size of $k$ (see ① in Algorithm 1). Then the maximum distance of the members, equal to the last one, can be used to set or update the radius or upper time bounds (see ② in Algorithm 1). Which value depends on the distance measure used for evaluating the nearest members, either the distance to the ray or the distance to the origin. This gives additional flexibility for certain query modalities.

## 4. Implementation

This section provides a brief overview of the implementation used to exemplify the proposed query approach described in the previous section, as well as the evaluation presented in the following section.

### 4.1 Octree

The ray query approach outlined in the previous section has been implemented on top of a modified octree based on (Warren and Salmon, 1993). Customizations made are to use a lazy-initialized static index map per tree layer for nodes, instead of hashing, and explicitly encoding the node depth in the ID next to the z-order. Furthermore, various data organization modes are supported, including storing elements in each node (layered), with a fixed depth (grid), or with dynamic node splitting based on the number of elements (adaptive).

### 4.2 Database

One possible approach to integrating the queries into a database management system and its query engine is to create scalar user-defined functions for the point-to-ray and point-to-plane distances. We exemplified this based on DataFusion, an extensible query engine (Lamb et al., 2024). The user-defined functions were further extended to support evaluating input bounds against the statistics for pruning chunks.

**User Defined Functions.** The two scalar distance functions we implemented are the point to ray distance (Equation 5) and the signed point to plane distance function (Equation 6). The scalar ray distance function is defined as

$$d_l = dist_ray(o_x, o_y, o_z, dir_x, dir_y, dir_z, \mathrm{x}, \mathrm{y}, \mathrm{z}) \tag{5}$$

where $(o_x, o_y, o_z)$ and $(dir_x, dir_y, dir_z)$ are the ray origin and direction and $(\mathrm{x}, \mathrm{y}, \mathrm{z})$ the point coordinates. The signed plane distance function is defined analogously as

$$d_p = dist_plane(p_x, p_y, p_z, n_x, n_y, n_z, \mathrm{x}, \mathrm{y}, \mathrm{z}) \tag{6}$$

where $(p_x, p_y, p_z)$ is a point on the plane and $(n_x, n_y, n_z)$ its normal. These two functions are sufficient to express all the presented ray query modalities in Section 3.1 and hence, can accommodate out-of-core ray tracing.

**Bounds Evaluation.** The user-defined functions can be extended with a bounds evaluation method. The inputs are the intervals for each function argument, from which an output interval is calculated. Given that the ray and plane arguments are expected to be scalars, the upper and lower values of their input interval are equal. For coordinates, the interval is determined by an indexing structure such as chunk statistics or node bounds. The resulting distance interval can therefore be calculated as described in Section 3.3. The output is finally evaluated against the comparison predicate for pruning. For example, to query all points of a ray within the radius $r = 1$, the corresponding SQL query is:

```
SELECT *
FROM points
WHERE dist_ray(...) <= 1
```

## 5. Experiments

In this section, we highlight some initial findings regarding the evaluation of the proposed approach. The following experiments are conducted on a mobile computer with 8 cores (16 threads), 32 GB of main memory, and 1 TB of SSD storage.

### 5.1 Index Validation

To validate our Octree implementation, we compare it against an R*-tree with $10^5$ randomly generated points in the unit cube (see Table 1). For the construction time, we use bulk loading for the R*-tree and insertion for the Octree. For the range queries, we run $10^3$ iterations, each consisting of two randomly generated points. For the neighborhood queries, we choose $10^3$ points to query with $k = 8$.

| Index | Construction | Range Queries | kNN Query |
|---|---|---|---|
| R*-tree | 17.2 ms | 1.1 ms | 7.1 ms |
| Octree | 7.6 ms | 0.9 ms | 4.5 ms |

Table 1. Mean runtime of tree construction, range, and kNN queries compared against R*-tree.

The results in Table 1 display competitive characteristics for loading and querying compared to state-of-the-art bulk loading R*-trees.

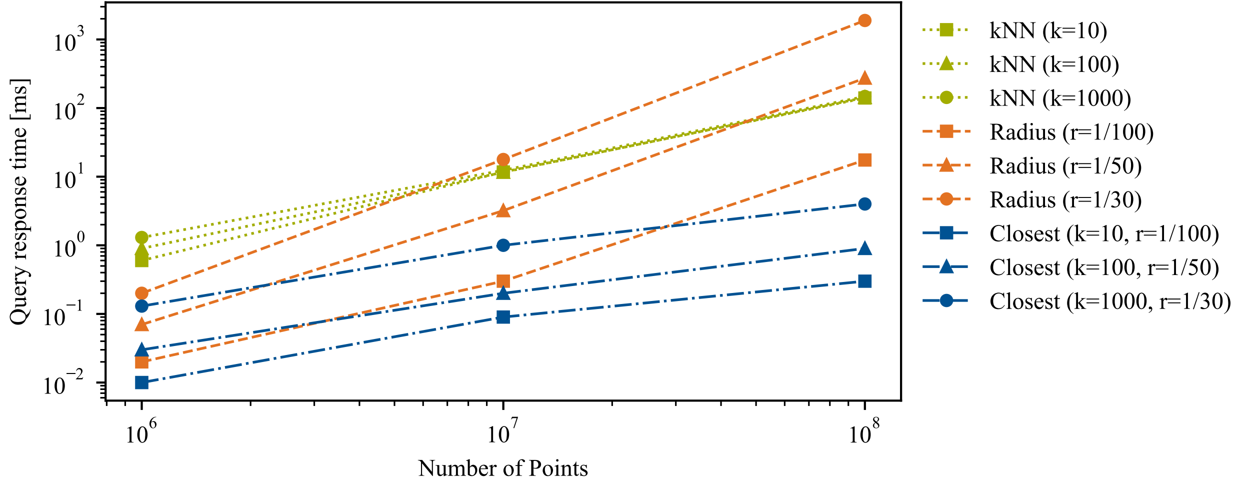



Figure 5. Ray query performance of kNN, radius, and closest (kNN with distance to origin within radius) modalities parameterized with various $k$, radius, $r$, and number of points.

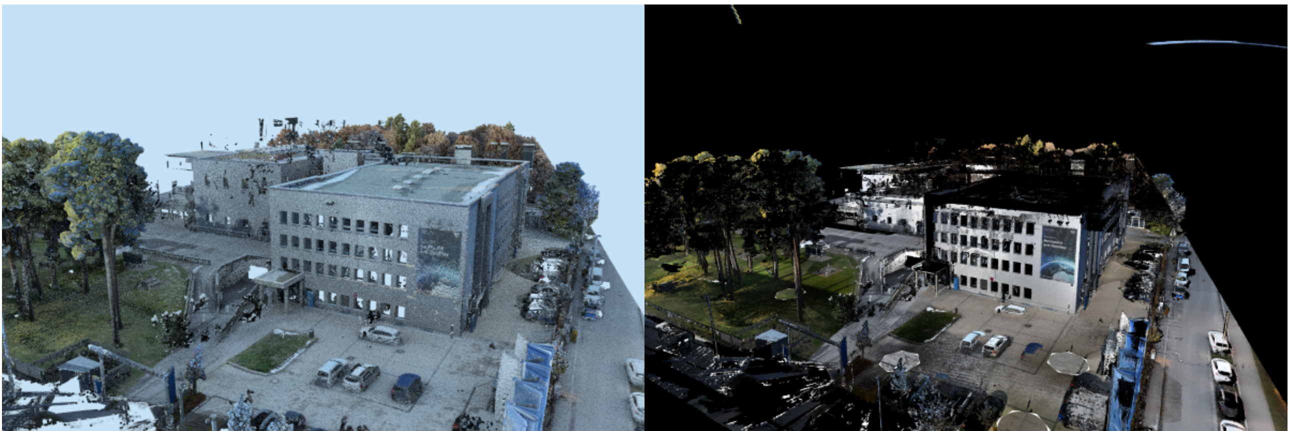
### 5.2 Query Performance

To evaluate the query performance characteristics, we run kNN queries with different $k$ values, radius queries, and combinations thereof, using distance to the ray origin as a measure on synthetic data comprising 1, 10, and 100 million points (Figure 5). The results show that the queries scale relatively well, especially those constrained by both $k$ and radius. Furthermore, queries can be executed in parallel, with performance scaling linearly with the number of available cores. In a real-world scenario, as presented below, this resulted in approximately 30,000 ray queries per second.

### 5.3 Image Synthesis

While the presented ray queries do not target image synthesis per se, and therefore, do not require a ray tracer, they resemble a predestined example application. To facilitate this, we implemented a simple ray tracer based on *Ray Tracing in One Weekend* (Shirley, 2018). The parts directly used from this to query are the ray data structure and the function that tests whether a ray hits (intersects) a bounding volume, as well as handling materials, including reflectance and color. The main adaptations we made are using our octree instead of the BVH approach described in the original and supporting real-world coordinate values using double-precision floating-point numbers. In this experiment, we employ a naive approach to calculate the normal and hit position, as well as color, based on a sphere with the closest point as its center and a search radius equal to the radius, using a Lambertian material. As a real-world dataset, we use the laser scan of the TUM Campus Ottobrun (Anders et al., 2024). It consists of about 200 million points from both terrestrial and UAV sensors, colorized from images.

Figure 6 shows a rendering of the TUM Campus Ottobrunn using Pointersect (Chang et al., 2023) and our simple ray tracing method using spheres. Both images have been rendered at 600 by 400 pixels, with 50 sampled rays per pixel. The Pointersect model was run on an A100 and completed within several hours, with most of the time spent executing the ray queries. Although it relies on custom CUDA shaders and a grid index for ray tracing, scalability is subpar for large point clouds. The simple ray tracing approach, on the other hand, took only several minutes on the CPU.



(a) Simple ray tracing (b) Pointersect

Figure 6. Visualization of TUM Campus Ottorbrun with (a) simple ray tracing on spheres and (b) Pointersect.

## 6. Conclusion

In this work, we demonstrated a simple approach to retrieve points from raw point clouds by means of ray queries that can be applied to most existing indexing structures used in the geospatial domain, such as R-trees, Octrees, Kd-trees, Grids, and Space Filling Curves (SFC), alike. Although the performance is not comparable to optimized ray tracers geared towards real-time computer graphics, it offers the benefits of scalability and great flexibility in tackling and prototyping research questions involving ray queries and geospatial data with low investment and familiar tooling. While focused on raw point clouds, extending to generic geometries is straightforward and considered for future work, including the integration of more elaborate surface intersection methods for increased visual fidelity and cone queries.

## Data and Software Availability

The code used to get the results in Chapter 5 can be made available upon request. The used data is publicly available (Anders et al., 2024).

## Acknowledgements

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 507196470.

## References

Adamson, A., Alexa, M., 2003. Approximating and Intersecting Surfaces from Points. *Proceedings of the 2003 Eurographics/ACM SIGGRAPH Symposium on Geometry Processing*, Eurographics Association.

Anders, K., Wang, J., Chang, M., Letard, M., Schulte, F., Winiwarter, L., 2024. Terrestrial and UAV Laser Scanning Point Clouds of TUM Campus Ottobrunn.

Anders, K., Winiwarter, L., Lindenbergh, R., Williams, J. G., Vos, S. E., Höfle, B., 2020. 4D objects-by-change: Spatiotemporal segmentation of geomorphic surface change from LiDAR time series. *ISPRS Journal of Photogrammetry and Remote Sensing*, 159, 352–363.

Chang, J.-H. R., Chen, W.-Y., Ranjan, A., Yi, K. M., Tuzel, O., 2023. Pointersect: Neural Rendering with Cloud-Ray Intersection. *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Vancouver, BC, Canada, 8359–8369.

Cho, F. S., Forsyth, D., 1999. Interactive ray tracing with the visibility complex. *Computers & Graphics*, 23(5), 703-717. https://doi.org/10.1016/S0097-8493(99)00093-X.

Glassner, A. S., 1989. *An introduction to ray tracing*. Morgan Kaufmann.

Gobbetti, E., Marton, F., 2004. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6), 815–826.

Kerbl, B., Kopanas, G., Leimkuehler, T., Drettakis, G., 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics*, 42(4), 1–14. https://dl.acm.org/doi/10.1145/3592433.

Kosmopoulos, P., Dhake, H., Kartoudi, D., Tsavalos, A., Koutsantoni, P., Katranitsas, A., Lavdakis, N., Mengou, E., Kashyap, Y., 2024. Ray-Tracing modeling for urban photovoltaic energy planning and management. *Applied Energy*, 369, 123516.

Laass, M., 2021. Point in polygon tests using hardware accelerated ray tracing. *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '21, Association for Computing Machinery, New York, NY, USA, 666–667.

Lamb, A., Shen, Y., Heres, D., Chakraborty, J., Kabak, M. O., Hsieh, L.-C., Sun, C., 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. *Companion of the 2024 International Conference on Management of Data*, ACM, Santiago AA Chile, 5–17.

Ma, J., Liu, M., Ahmedt-Aristizabal, D., Nguyen, C., 2024. HashPoint: Accelerated Point Searching and Sampling for Neural Rendering. *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Seattle, WA, USA, 4462–4472.

Meagher, D., 1982. Geometric modeling using octree encoding. *Computer Graphics and Image Processing*, 19(2), 129–147.

Meister, D., Ogaki, S., Benthin, C., Doyle, M. J., Guthe, M., Bittner, J., 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum*, 40(2), 683–712.

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., Ng, R., 2022. NeRF: representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1), 99–106. https://dl.acm.org/doi/10.1145/3503250.

Rusu, R. B., Cousins, S., 2011. 3D is here: Point Cloud Library (PCL). *IEEE International Conference on Robotics and Automation*, IEEE, Shanghai, China, 1–4.

Shirley, P., 2018. Ray tracing in one weekend. *Amazon Digital Services LLC*, 1(4).

Stojanovic, N., Stojanovic, D., 2014. High performance processing and analysis of geospatial data using CUDA on GPU. *Advances in Electrical and Computer Engineering*, 14(4), 109–115.

Vitucci, E., Degli-Esposti, V., Fuschini, F., Lu, J., Barbiroli, M., Wu, J., Zoli, M., Zhu, J., Bertoni, H., 2015. Ray tracing RF field prediction: An unforgiving validation. *International Journal of Antennas and Propagation*, 2015(1), 184608.

Wald, I., Woop, S., Benthin, C., Johnson, G. S., Ernst, M., 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4), 1–8.

Warren, M. S., Salmon, J. K., 1993. A parallel hashed Oct-Tree N-body algorithm. *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Association for Computing Machinery, Portland, Oregon, United States, 12–21.

Zhou, Q.-Y., Park, J., Koltun, V., 2018. Open3D: A modern library for 3D data processing. *arXiv preprint arXiv:1801.09847*.