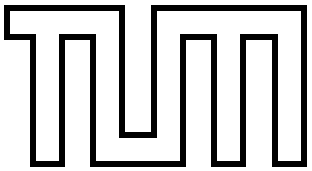


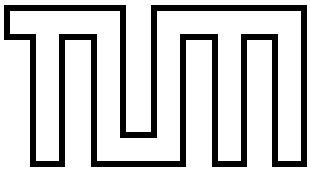
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Construction of KD-Trees for Spatial Partitioning of Polyhedral Models

Ben Sebastian Frauenknecht



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Efficient Construction of KD-Trees for Spatial Partitioning of Polyhedral Models

Effiziente Konstruktion von KD-Bäumen zur räumlichen Unterteilung von Polyhedron Modellen

Author:	Ben Sebastian Frauenknecht
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Jonas Schuhmacher, M.Sc.
Date:	01.02.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 01.02.2025 Ben Sebastian Frauenknecht

Acknowledgements

I'm deeply grateful to my advisor, Jonas, for always giving invaluable feedback, even in sickness. You always gave it your all for me, enabling me to do the same.

To my brother, who spent three hours sitting in my bed helping me refine the math sections.

To my family and loving girlfriend, who never stopped showering me with love and endless patience. You are the reason I made it this far.

Abstract

Ray tracing is a versatile technique used across various fields, most notably in image rendering, but it also plays a critical role in scientific computation. Although ray tracing in its basic form requires extensive computational power, several strategies exist to improve efficiency. One approach is leveraging specialized hardware like GPUs, while optimized data structures, such as BSP-trees, KD-trees, and Octrees, also offer performance improvements. In this thesis, we will improve the ray tracing-based mesh check algorithm deployed in an implementation of a polyhedral gravity model by Schuhmacher et al. The best-suited data structure for this application is the KD-tree because it focuses on optimal space subdivision and guarantees fast build times in $O(n \cdot \log(n))$ by adhering to the descriptions of Wald et al. Currently, there is no easy-to-use, high-quality implementation promising fast build times. Thus, we create a new KD-tree library written in modern C++17, employing the "lazy loading" pattern. It follows best software practices and is optimized to reduce runtime. We deploy multithreading techniques, achieving full CPU core utilization for 80% of total runtime. The mesh-check for the Eros asteroid mesh comprising 70150 vertices and 140296 faces takes 4.7 seconds using the highest optimization level of the library. Execution without the integrated KD-tree takes 54.4 seconds. We validate our implementation through extensive input fuzzing and regression testing, comparing differently optimized implementations of KD-tree construction algorithms.

Zusammenfassung

Raytracing ist eine vielseitige Technik, die in verschiedenen Bereichen eingesetzt wird. Sie kommt vor allem bei der Bildsynthese zum Einsatz, aber auch in wissenschaftlichen Anwendungen spielt sie eine wichtige Rolle. Obwohl Raytracing in seiner Grundform eine hohe Rechenleistung erfordert, gibt es zahlreiche Strategien zur Verbesserung der Effizienz. Ein Ansatz ist die Nutzung von Spezialhardware wie GPUs, während optimierte Datenstrukturen wie BSP-Bäume, KD-Bäume und Octrees ebenfalls Leistungsverbesserungen bieten. In dieser Arbeit wird der auf Raytracing basierende Mesh-Check-Algorithmus verbessert, der in einer Implementierung eines polyedrischen Gravitationsmodells von Schuhmacher et al. eingesetzt wird. Die am besten geeignete Datenstruktur für diese Anwendung ist der KD-Baum, da er sich auf eine optimale Unterteilung konzentriert und schnelle Erstellungszeiten in $O(n \cdot \log(n))$ garantiert, wie Wald et al. zeigen. Derzeit gibt es keine einfach zu verwendende, qualitativ hochwertige Implementierung, die schnelle Bauzeiten verspricht. Daher erstellen wir eine neue KD-Baum Software Library, geschrieben in modernem C++17, die das "Lazy Loading" Software Pattern verwendet. Sie folgt etablierten Software-Praktiken und ist auf Laufzeitreduzierung optimiert. Wir setzen Multithreading-Techniken ein und erreichen so eine volle Auslastung der CPU-Kerne für 80% der Gesamtlaufzeit. Der Mesh-Check für das Eros-Asteroidenmesh mit 70150 Knoten und 140296 Flächen dauert 4,7 Sekunden bei Verwendung der höchsten Optimierungsstufe der Bibliothek. Die Ausführung ohne den integrierten KD-Baum dauert 54,4 Sekunden. Wir validieren unsere Implementierung durch umfangreiches Input-Fuzzing und Regressionstests, die unterschiedlich optimierte Implementierungen von KD-Baum-Konstruktionsalgorithmen vergleichen.

Contents

Part I.

Introduction and Background

1. Introduction

In 2018, NVIDIA revolutionized the gaming industry by introducing a new series of graphics processors designed specifically for real-time ray tracing [1]. This groundbreaking technology enabled the rendering of photo-realistic lighting effects, such as accurate reflections and shadows, in real time, significantly enhancing visual immersion. Although ray tracing has existed for decades, it has traditionally been associated with high computational costs and slow processing times, limiting its widespread adoption in interactive applications.

One approach to mitigate these computational challenges, as demonstrated by NVIDIA, involves leveraging specialized hardware to accelerate ray tracing computations. Another complementary strategy is incorporating optimized data structures, which aim to reduce the overall number of computations required during rendering.

This thesis aims to enhance the ray tracing runtime performance of the mesh-check algorithm in the polyhedral gravity model application by Schuhmacher et al. [2]. We do so by integrating an efficient implementation of a KD-tree data structure. The proposed solution will be developed using modern C++, focusing on optimizing construction and traversal processes. The resulting implementation will be evaluated through performance testing to demonstrate its effectiveness in achieving a measurable speedup.

2. Theoretical Background

This section depicts algorithms and mathematical concepts to understand the inner workings of a KD-tree. For this purpose, we will dive into intersection algorithms first and then apply them to build the tree efficiently.

2.1. Ray Tracing

Besides rasterization, ray tracing is one of two prominent traditional approaches to rendering 3D scenes [3]. It uses rays shot from an origin point, most often the camera position, to determine intersection points with objects in the scene. The results are then used to determine the pixel colors shown to the user. For intersection tests with arbitrarily shaped objects to work, objects need to be represented as a mesh. The mesh is a collection of multiple smaller interconnected 2D shapes that roughly approximate the surface of the original object. With large quantities of smaller shapes, the approximation of the object becomes better, which can be seen in Figure 2.1. In theory, arbitrary shapes can be used in a mesh. However, triangles are often used due to their simplicity [4, p.19].

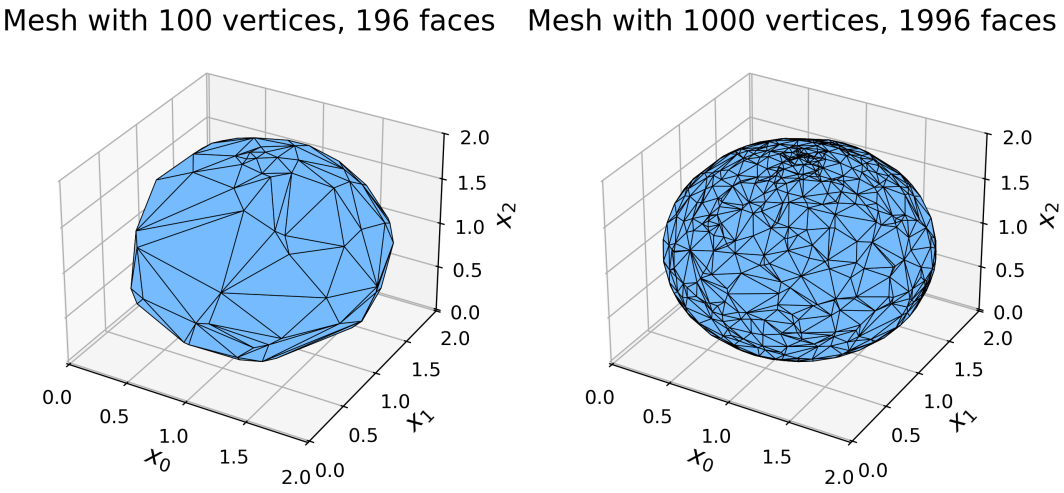


Figure 2.1.: Example sphere triangle meshes with different amount of detail

To perform an intersection test for objects represented by meshes, one only needs to perform tests for each face of the mesh. By using this divide-and-conquer approach, we reduce the complexity of the intersection algorithm because only ray-triangle intersections are performed.

2.2. Ray-Triangle Intersection Test

In this thesis, the algorithm used for performing intersection tests of rays with triangles is the Möller-Trumbore algorithm [5]. In order to be able to explain the algorithm in detail, a few key mathematical concepts have to be elaborated on.

2.2.1. Barycentric Coordinates

When encoding the position of a fixed point in space, the norm is to use a linear combination of the unit vectors of the space we operate in [6, pp.14-16]. Let's assume we place the point $\overrightarrow{P} := (1, 1, 2)^T$ in $\mathbb{R}^3$. The linear combination would then be:

$$\overrightarrow{P} = 1 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 1 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + 2 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} \tag{2.1}$$

The scalars multiplied by the linear combination's unit vectors are called coordinates. When we look at barycentric coordinates, we do not use the unit vectors as the base of the vector space in the linear combination. Instead, we utilize the position vectors of a triangle's vertices, assuming that the triangle's area is nonzero. This was first introduced by Möbius [7, p. 26]. We extend example Equation 2.1 by calculating the barycentric coordinates of $\overrightarrow{P}$ based on the triangle $\triangle ABC := (\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 3 \end{pmatrix}, \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix})$:

$$0 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + \frac{2}{3} \cdot \begin{pmatrix} 1 \\ 0 \\ 3 \end{pmatrix} + \frac{1}{3} \cdot \begin{pmatrix} 1 \\ 3 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix} \tag{2.2}$$

From Equation 2.2 we can infer the barycentric coordinates $(0, \frac{2}{3}, \frac{1}{3})$ of $\overrightarrow{P}$. If a point is located on the triangle, then the barycentric coordinates sum up to one [7, p. 35][8]. In that case, we can express the third barycentric coordinate w by using the complement $1 - u - v$, with u, v being the first and second barycentric coordinates. For the Möller-Trumbore algorithm, it is not important how to calculate barycentric coordinates in detail. Thus, we will not explore this topic any further. The interested reader is advised to Skala [8].

2.2.2. Möller-Trumbore Algorithm

A ray can be defined by a point of origin $\overrightarrow{O}$ and a direction vector $\overrightarrow{d}$. The points lying on a fixed ray can be expressed by inserting arbitrary ray argument $t \in \mathbb{R}$ into:

$$\overrightarrow{r}(t) = \overrightarrow{O} + t \cdot \overrightarrow{d} \tag{2.3}$$

Finding an intersection point of a ray $(\overrightarrow{O}, \overrightarrow{d})$ and triangle $\triangle ABC$ is equivalent to solving the following equation for t.

$$\overrightarrow{O} + t \cdot \overrightarrow{d} = (1 - u - v) \cdot \overrightarrow{A} + u \cdot \overrightarrow{B} + v \cdot \overrightarrow{C} \tag{2.4}$$

This means to find a point represented by ray argument t that lies on the ray and inside the triangle by finding its barycentric coordinates u, v. Rearranging Equation 2.4 yields

$$\overrightarrow{AB} \cdot u + \overrightarrow{AC} \cdot v - \overrightarrow{d} \cdot t = \overrightarrow{AO} \tag{2.5}$$

This is effectively a linear system of equations. Solving the system can result in three cases:

Case 1: There is no solution for the system
→ The ray is parallel to the triangle, so the plane the triangle defines is not hit.

Case 2: There exists a solution, but $u, v \notin [0, 1]$ or $u + v > 1$
→ The ray hits the plane defined by the triangle, but the intersection point lies outside the triangle.

Case 3: There exists an intersection point inside the triangle ($u, v \in [0, 1]$ and $u + v \leq 1$) that can be calculated by inserting the solved value for t in Equation 2.3.

2.3. Axis Aligned Bounding Box

In the following chapters, we need a concept called axis aligned bounding box (hereafter referred to as AABB) elaborated by Ericson [9, p.78]. It describes a box whose edges are each constrained to be parallel to one of the coordinate axes of the space $\mathbb{R}^n$ it is located in. It is generally used to encapsulate other objects of arbitrary shape, providing a more

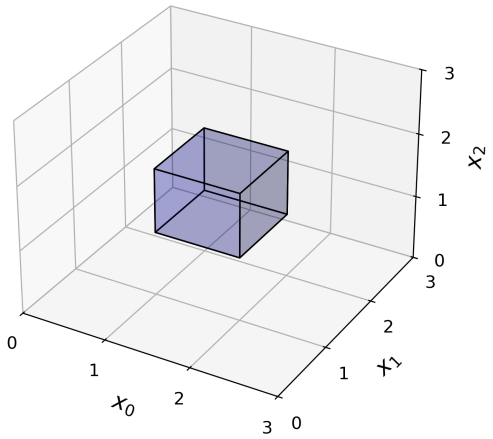


Figure 2.2.: Example of an AABB in three-dimensional space

straightforward box shape of the object through this approximation. Overlap or intersection tests benefit significantly from this practice. For the best approximation, an AABB needs to have minimal volume while still containing the object entirely, meaning the object is not allowed to have non-zero volume outside the AABB. An AABB can be defined by specifying two diagonally opposite corner points, $minPoint = (c_{min,0}, \ldots, c_{min,n-1})$ and $maxPoint = (c_{max,0}, \ldots, c_{max,n-1})$, of the AABB. We further demand that

$$c_{min,i} \leq c_{max,i} \mid i \in [0, n-1] \tag{2.6}$$

holds. This assures that equal AABBs are not defined ambiguously and facilitates constructing them around objects. We iterate over all object vertices and store minimal and maximal values coordinate-wise. By constructing points out of the minimal and maximal coordinate values each, we receive $minPoint$ and $maxPoint$.

2.3.1. Ray-AABB Intersection Test

In order to intersect a ray with an AABB, it is possible to naively intersect each face of the bounding box with the ray. There are certainly more efficient ways to accomplish this task. In our implementation, we use Smits' algorithm [10] modified by Wiliams et al. [11]. As this algorithm utilizes the intersection of rays and axis-aligned planes, we will depict this topic first.

Ray and Axis Aligned Plane Intersection Test

Intersecting an axis-aligned plane with a ray is more straightforward than doing a typical ray-plane intersection. When operating in coordinate systems of arbitrary amounts of dimensions, we only need to solve an equation for the single dimension D with index d for which the plane's normal vector has a nonzero coordinate. Let $origin_d$ be the origin point coordinate in D, $ray_d \neq 0$ the intersection ray in D, and $anchor_d$ the plane anchor point in D. Then we can calculate a parameter t by solving:

$$t = \frac{anchor_d - origin_d}{ray_d} \tag{2.7}$$

If t is negative, then there is no intersection point. Otherwise, plugging t into Equation 2.3 calculates the intersection point. Special attention is needed for the case $ray_d = 0$, previously excluded. This means that the ray and the plane are parallel. If $origin_d \neq ray_d$ holds, no intersection points are present then. Otherwise, infinitely many intersection points exist as the ray lies in the plane, and so does every point on it.

Smits' Algorithm

Smits' algorithm [10] adapted by Wiliams et al. [11] relies on defining the AABB as the volume enclosed by a set of intersecting planes called slabs. They can be calculated using the minimal ($minPoint$) and maximal corner ($maxPoint$) vertices as anchor points for the slab. As for the slabs' normal vectors, we use each unit vector of the tree's vector space $\mathbb{R}^n, n \in \mathbb{N}$ once for each anchor point. This results in six planes or slabs if the KD-tree is built in $\mathbb{R}^3$.

The intersection algorithm iterates over all n unit vectors of $\mathbb{R}^n$ and intersects the ray with the two slabs defined by $minPoint$ and $maxPoint$, using the unit vector as the normal vector. Ray-plane intersection, as defined by Subsubsection 2.3.1, is utilized for calculating intersections. We will also use the naming scheme defined there in the following.

Intersection results in two values, one per slab, $t_{min,d}$ and $t_{max,d}$ for the normal vector with nonzero coordinate at index d. Now, we need to discern which value describes the point on the slab that is hit first ($t_{enter,d}$) and which slab is hit afterward ($t_{exit,d}$). This can easily

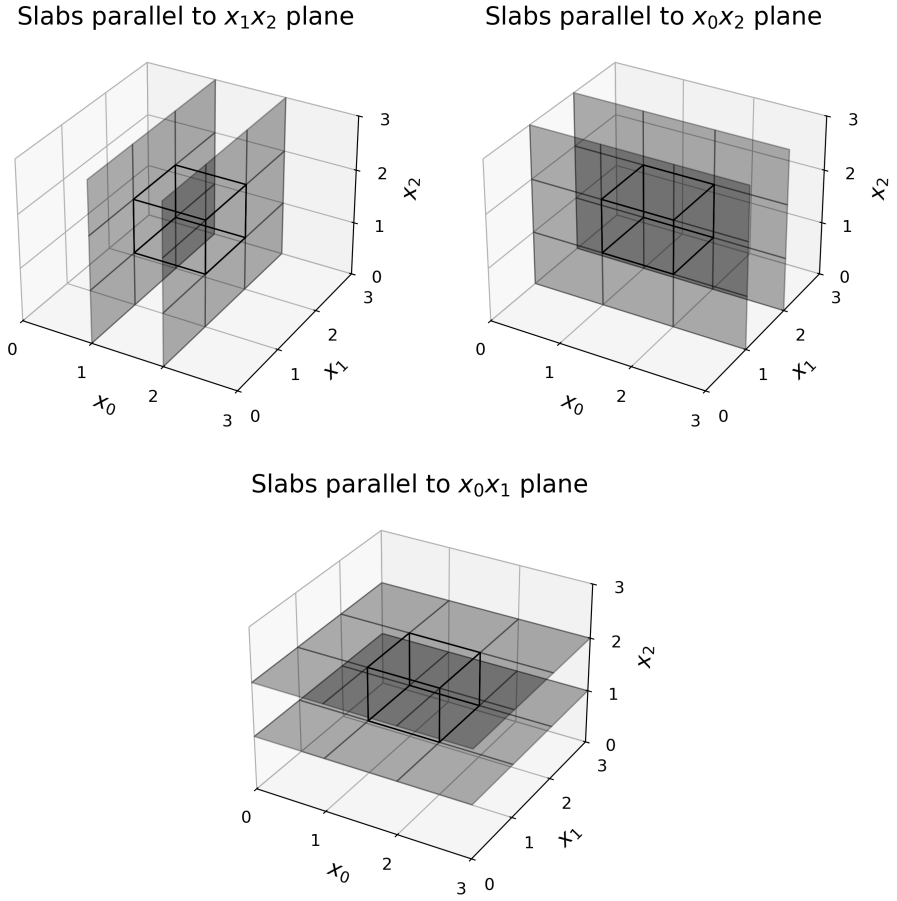


Figure 2.3.: Slabs visualized for an AABB in $\mathbb{R}^3$

be done by evaluating $sign(ray_d)$:

$$ray_d \geq 0 \implies t_{enter} = t_{min,d} \ \wedge \ t_{exit} = t_{max,d}$$
$$ray_d < 0 \implies t_{enter} = t_{max,d} \ \wedge \ t_{exit} = t_{min,d}$$

$t_{enter,d}$ and $t_{exit,d}$ act as interval limits that t_P, t-value for a point P, may take in order for P to lie in the AABB when both AABB and P are projected onto the same coordinate axis. P is only inside of the AABB if the constraints for all axes are fulfilled:

$$P \in AABB \iff t_{enter,d} \leq t_P \leq t_{exit,d} \ , \forall d \in [dim(\mathbb{R}^3)] \tag{2.8}$$
$$\iff \max_{d \in [dim(\mathbb{R}^3)]} t_{enter,d} \leq t_P \leq \min_{d \in [dim(\mathbb{R}^3)]} t_{exit,d} \tag{2.9}$$

The entry point can be calculated by inserting the smallest t-value still inside the box and inserting it into the ray equation Equation 2.3. The smallest t-value is specified by Equation 2.9, by calculating the max-expression. The same can be done for the exit point analog with the min-expression.

2.4. KD-Trees

As stated by Havran et al. [12, p. 3], the naive ray intersection algorithm by simply brute forcing is not efficient for scenes with many objects. The cost of intersecting a single ray amounts to $O(N)$, with N being the amount of objects to test against in the scene. This is problematic since most problems often require multiple ray intersections to be computed. For larger, more complex meshes, this becomes computationally infeasible relatively fast. To resolve this issue, they describe hierarchical data structures to spatially divide the scene into smaller subspaces called cells. They then test against these cells first to reduce the amount of total intersections. The most efficient of the reviewed structures is the KD-Tree [12, pp. 11-14]. The KD-tree is a multidimensional binary tree structure that divides a K-dimensional space into cells. Each inner node n is assigned an axis-aligned bounding box (AABB), here called v. The nodes of the tree correspond to the cells mentioned above. Along with v, an inner node also defines a split plane p. The split plane splits v into two subspaces, assigned to the left and right children of n. They can again be inner nodes and further split each assigned space. In the tree's leaf nodes, no further splitting is performed. Instead, they reference the objects contained in the AABB for which they are responsible. Should no objects exist in the AABB, then the leaf is called empty and does not contain any object references.

One special characteristic of the KD-tree is that each splitting plane needs to be parallel to one of the axes of the space the tree is built in. This is analogous to the orientation of an AABB's edges. This is the main difference to the BSP-Tree, mentioned in Section 3.2, where arbitrarily oriented bounding box edges and splitting planes are allowed. According to Wald et al. [13], there exist two main approaches to determining the split plane position:

- **Spatial median splitting:** This approach places the splitting plane at the mid-point of the AABB. The orientation is then determined by iterating over the k dimensions increasing with tree node depth. The iteration starts over at the beginning when the kth dimension has been used. Formally expressed, the plane orientation of a node at depth t of dimensions $X_0 ... X_{k-1}$ is:

  $$X(t) = X_{t \bmod k}$$

  This approach is easy to compute, but it builds suboptimal KD-trees as it heavily simplifies the underlying scene geometry.
- **Evaluating the splitting plane using a cost function:** The other group of approaches tries to evaluate the plane's effectiveness by using cost functions and then choosing the optimal plane. The most sophisticated variants use the surface area heuristic.

2.4.1. Surface Area Heuristic

The SAH estimates the probability of an arbitrary ray with arbitrary origin hitting a convex object A contained in another convex object R under the following assumptions [14]:

- The ray origin is sufficiently far away from objects A and R.
- Every ray origin and direction is equally likely to appear during intersection testing.

If the conditions are met, then the probability of a ray r intersecting A, given that R is intersected, can be roughly calculated as follows:

$$hit(X, r) = \begin{cases} true, & \text{if ray } r \text{ hits object } X \\ false, & \text{otherwise} \end{cases} \tag{2.10}$$

$$P(A|R) := P(hit(A, r) = true \,|\, hit(R, r) = true) = \frac{SA(A)}{SA(R)} \tag{2.11}$$

$SA(A)$ and $SA(R)$ are the surface area of A and R, respectively. Combine the probability of intersection with the amount of work that has to be performed should a hit occur, and construction of a cost function for split evaluation becomes possible [13]. For that, we introduce constants κ_T and κ_I. κ_T is the cost of traversing one node in the tree, and κ_I describes the cost of a ray-triangle intersection. The original work by Wald et al. [13] uses variables V_l and V_r for the halfspaces left and right of the plane. However, we find that this naming scheme does not scale well to higher dimensions. Thus, we call the halfspace "lesser" if it is closer to the origin and "greater" if it is further away. Using κ_T, the cost-effectiveness of a splitting plane p dividing a voxel V into V_l and V_g is then expressed by:

$$C(p) = \kappa_T + P(V_l|V) \cdot C(V_l) + P(V_g|V) \cdot C(V_g) \tag{2.12}$$

Expanding Equation 2.12 using Equation 2.10 results in:

$$C(p) = \kappa_T + \frac{SA(V_l)}{SA(V)} \cdot C(V_l) + \frac{SA(V_g)}{SA(V)} \cdot C(V_g) \tag{2.13}$$

One could recursively apply this equation to calculate the globally optimal split planes for the perfect KD tree. This becomes expensive and impractical for non-trivial scenes due to the number of possible plane position combinations exploding. We assume that only leaves follow after the current tree node to solve this issue. With $|T|$ being the number of triangles contained in the leaf node, its cost can be expressed by:

$$\kappa_I \cdot |T| \tag{2.14}$$

We end up with a locally greedy variant of Equation 2.13:

$$C(p) = \kappa_T + \frac{SA(V_l)}{SA(V)} \cdot \kappa_I |T_l| + \frac{SA(V_g)}{SA(V)} \cdot \kappa_I |T_g| \tag{2.15}$$

Even though Equation 2.15 overestimates the actual cost by ruling out any cost reductions that further node splits could achieve, this variant remains the best approximation.

Modifications

Wald et al. [13] propose an additional weighting to the cost function. If a split results in one of the halfspaces containing no triangles, the split's cost is further reduced by 20%. They intend to cut off empty space early and avoid passing it to child nodes.

Termination Criterion

Using the SAH, an elegant way to check whether subdividing space is still profitable presents itself. When the optimal split plane is found, we record its cost, $cost_{split}$. Then, we treat the current node as a leaf node and compute $cost_{leaf}$ using Equation 2.14. Afterward, we compare $cost_{split}$ with $cost_{leaf}$ and, based on that, decide whether to continue splitting the node or terminate the recursion.

2.4.2. Split Plane Candidates

We have discussed how to evaluate a split plane for constructing an inner node in the KD-tree. This still leaves the question of which plane candidates should be considered for evaluation in the first place. Wald et al. [13] construct "split candidates" for that purpose. In theory, there are infinitely many plane positions due to operating in a field of $\mathbb{R}^n, n \in \mathbb{N}$. Thus, the filtering of split planes is in order before moving on to evaluation. The locally greedy surface area heuristic cost function, defined in Equation 2.15, is linear for fixed amounts of T_l and T_g. This also means that the function's local minimum in between intervals, where the amount of triangles on each side does not change, can only lie at the borders of the interval. Because of that, the only relevant candidates are planes that count a specific triangle as part of T_l and then as part of T_g. Expressed differently, the triangle has to lie after the plane once and once before it when the plane's orientation is fixed. The slabs defined by the triangle's AABB show this exact behavior (Figure 2.4). To summarize, the split planes that

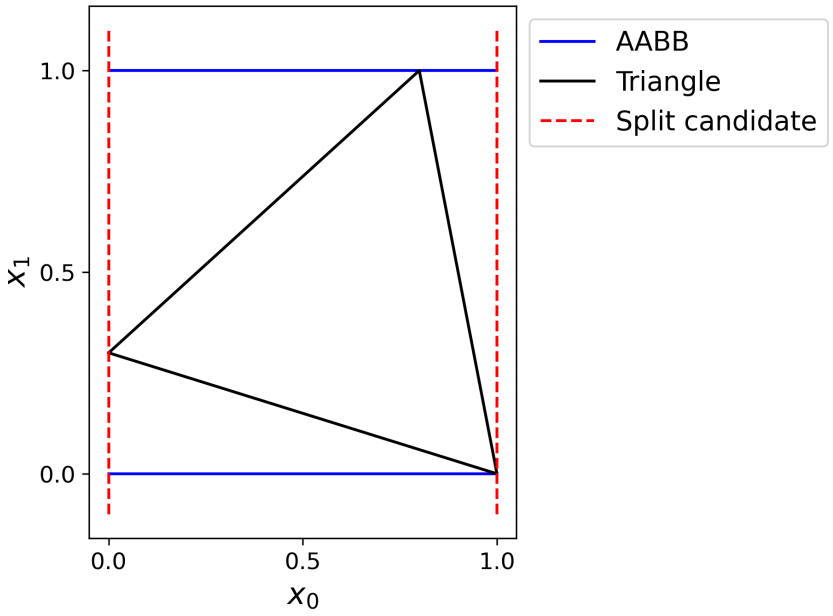


Figure 2.4.: Split candidates in x_0 direction visualized for a triangle in 2D by calculating the AABB

need to be evaluated are called split candidates. They are defined by the AABBs of the triangles of the mesh object we want to build the KD-tree for.

A triangle may not be fully contained in the inner node's AABB when traversing deeper down the tree. A case of this happening is depicted in Figure 2.5. A naive algorithm would

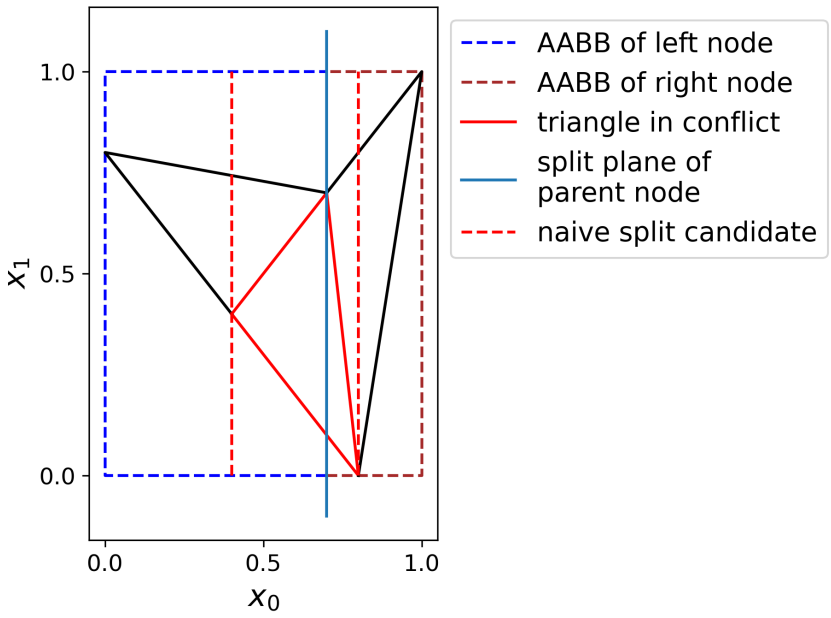


Figure 2.5.: A triangle's AABB not fully enclosed by inner node's AABB

then evaluate possible split planes, including those that lie outside the node to be split. This can result in faulty splits. So before we assess the split planes introduced by a triangle T, we clip the triangle to the AABB of the tree node we are trying to split. By clipping T and creating a new polyhedron T^*, we ensure that every split proposed by T^* is valid.

Clipping Triangles to AABBs

For clipping triangles to the AABBs, the Sutherland-Hodgman algorithm is used [15]. By clipping, the resulting object may not be a triangle anymore but a polyhedron of arbitrary shape. The clipping process iterates through the slabs of the AABB and subsequently clips to them instead. The result of the previous iteration is passed to the next iteration of clipping as input.

Clipping Triangles to Planes

Clipping an object O to a plane/slab is done by determining the object's vertex positions with respect to the plane. For that we define a distance measure for a vertex $v \in V$ and a plane P with anchor point p_a and normal vector $\overrightarrow{n}$:

$$distance(v) := (v - p_a) \circ \overrightarrow{n} \tag{2.16}$$

v lies inside the plane, if and only if $distance(v) \geq 0$. Constructing n to point to the inside of the AABB is important. Equation 2.16 can calculate distances between points and planes of

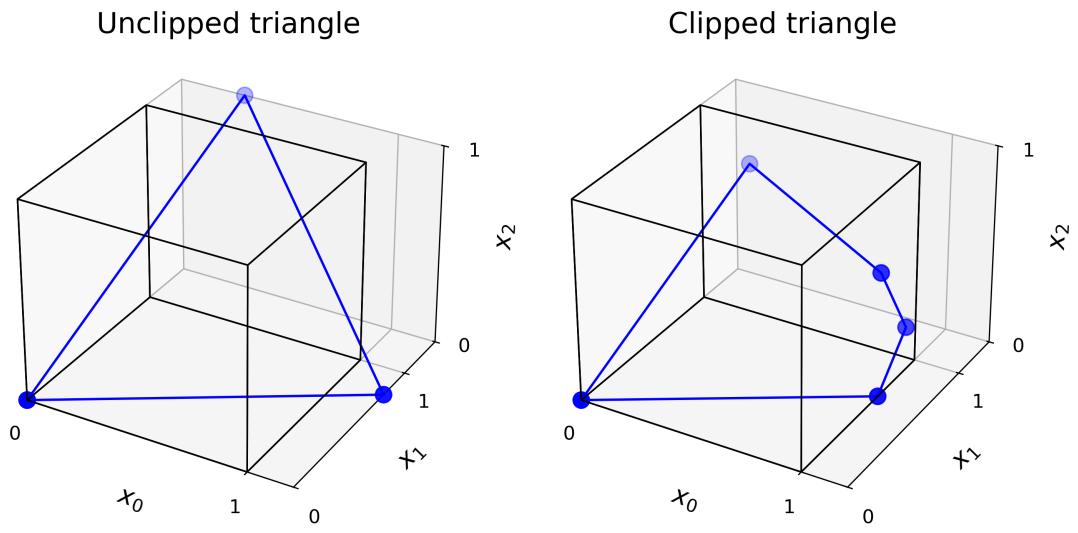


Figure 2.6.: Example of a triangle clipped to an AABB

arbitrary orientation. Nevertheless, since we enforce all split planes to be axis aligned inside the KD-tree, its normal vector only consists of one instance of ± 1 and zeroes otherwise. Because of that, the program can replace the dot product with an index access and a sign adjustment, making it faster. For the algorithm to work, we demand that the ordering of V is that so $v_i \in V$ and $v_{i+1 \bmod |V|} \in V$ (abbreviated with v_{in} in the following) are connected by an edge in O. We then iterate over V and determine the distance measures for each (v_i, v_{in}) pair. A total of four cases can occur:

Case 1: Both v_i and v_{in} lie inside the plane:
→ Add v_{in} to the list of new vertices.

Case 2: v_i is inside but v_{in} is outside the plane:
→ Add the intersection point of the edge defined by $[v_i, v_{in}]$ and the plane to the list of new vertices.

Case 3: v_i is outside and v_{in} is inside the plane:
→ Add the intersection point of the edge defined by $[v_i, v_{in}]$ and the plane, and v_{in} to the list of new vertices.

Case 4: Both v_i and v_{in} lie outside the plane:
→ Do nothing and continue the iteration.

Calculating intersection points of edges and planes the way Subsubsection 2.3.1 describes is possible. However, since we calculated the distance measures of v_i and v_{in}, we can reuse that information to linearly interpolate the intersection point directly instead [15]. We first calculate a parameter α:

$$\alpha = \frac{|distance(v_i)|}{|distance(v_i)| + |distance(v_{in})|} \overset{1}{=} \frac{distance(v_i)}{distance(v_i) - distance(v_{in})} \tag{2.17}$$

Afterwards, we can calculate the intersection point I with:

$$I = v_i + \alpha \cdot (v_{in} - v_i) = (1 - \alpha) \cdot v_i + \alpha \cdot v_{in} \tag{2.18}$$

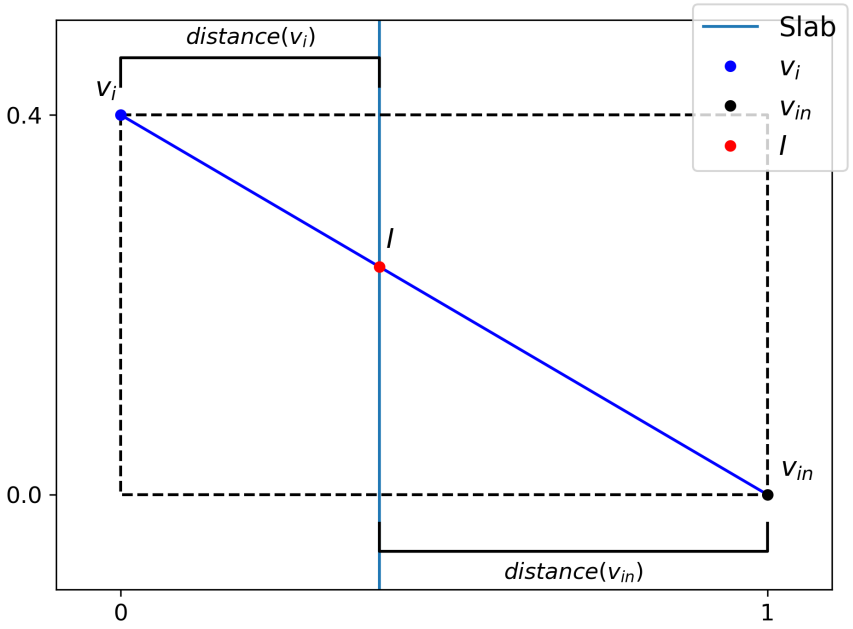


Figure 2.7.: Calculate intersection point using distance measures

2.4.3. Evaluating Split Plane Candidates

To be able to evaluate split planes, $|T_l|$ and $|T_g|$, introduced by the SAH in Equation 2.15, need to be calculated [13]. Depending on how these values are calculated, different runtime complexities are achieved. It is important to note that should a triangle straddle the split plane, effectively lying on both sides, it counts to both halfspace's triangle counters. Triangles that are planar and inside the plane are included in a separate set T_p. After all triangles have been classified, the cost determined by SAH is calculated once with T_p added to T_l and once to T_g. The minimal cost will be used to compare the split plane with other planes. Additionally, the combination that produced this cost will be recorded for later use.

$O(n^2)$ Algorithm

This is the simplest method to find the optimal split plane. We iterate over all possible split planes, calculate their cost, and track the plane with minimal cost. Each triangle generates

[1] This equivalence holds because we assume that this equation is only evaluated if and only if there exists an intersection point. Thus, we can infer that $distance(v_i)$ and $distance(v_{in})$ have different signs.

a linear number of split plane candidates, resulting in the number of iterations k_1 being in $O(N)$ (N corresponds to the number of triangle faces). We again iterate over all triangles for each cost calculation and classify their position relative to the plane. This amounts to another $k_2 \in O(N)$ iterations. Thus, the total complexity of this approach is quadratic.

$O(n \cdot \log(n)^2)$ Algorithm

In order to avoid classifying triangles multiple times, the next best variant calculates the relative triangle positions in an iterative scheme [13]. We printed the pseudocode of the procedure in Algorithm 1 for better understanding. Every dimension is evaluated separately. All split plane candidates are initially generated by iterating over all faces T bound by the current voxel V. We differentiate between split planes that lie "before" or "after" the face that generated it and record this information along with the plane. Formally, let $\overrightarrow{n}$ be the normal vector of the split plane P, whose anchor point is $\overrightarrow{p}$, and $T_\triangle \in T$ a triangle in V. $T_\triangle$'s locality relative to P is then determined by the function:

$$loc(T_\triangle) = \begin{cases} \text{"before"}, & \text{if } \forall \overrightarrow{t} \in \text{SurfacePointsOf}(T_\triangle), \forall n_i \in \overrightarrow{n} : n_i \neq 0 \implies t_i \leq p_i, \\ \text{"after"}, & \text{if } \forall \overrightarrow{t} \in \text{SurfacePointsOf}(T_\triangle), \forall n_i \in \overrightarrow{n} : n_i \neq 0 \implies t_i \geq p_i \end{cases} \tag{2.19}$$

We call the tuple of a split plane, the triangle face that generated the plane, and the locality of the plane to the face a *plane event*. There exist three different types of plane events depending on the locality they encode:

- Starting Plane Event: The triangle lies after the plane. Thus, the triangle "starts" in the plane.
- Planar Plane Event: The triangle lies on the plane, meaning it is before and after the plane. Analogously, the triangle also "starts" and "ends" in the plane.
- Ending Plane Event: The triangle lies before the plane, so it "ends" in the plane.

For the algorithm to work, the plane events must be sorted once before execution. The primary sorting criterion is the plane position in ascending order. Should equality occur, then the plane event type is used as a secondary measure with sorting order $Ending < Planar < Starting$. After sorting, Algorithm 1 sets the triangle counters required by the SAH.

Let A be a split plane that lies directly before a split plane B. Then, all triangles that lie before A also lie before B. To get the whole number of triangles before B, one needs to add the planar triangles in B itself and those that start at A. A similar approach can be made for triangles that lie after the planes: Subtract the ones planar and ending in B and from the triangles that lie after A. By iteratively updating T_l and T_g accordingly, we do not need to start anew every time we evaluate a new split plane.

After the optimal split plane has been found, all that is left to do is build the triangle sets for each halfspace and pass them to the respective child nodes. We can achieve that by iterating over the plane event list again and comparing the suggested split planes to the optimal plane we found earlier. If a triangle ends in a plane before the optimal plane, it is in T_l. The same works analogously for T_g and T_p. If none of these axioms hold, the triangle straddles the optimal plane and lies in both T_l and T_g.

Algorithm 1: $O(n \cdot \log^2(n))$ plane event parsing algorithm

Input : **AABB**: the axis-aligned bounding box of the current node to be split
triangles: triangles in the current AABB

$planeEvents \leftarrow generatePlaneEvents(triangles).sort()$ $i \leftarrow 0$
// there are no triangles before the first split plane candidate
$|T_l| \leftarrow |T_p| \leftarrow 0$
// all triangles lie after the first split plane candidate
$|T_g| \leftarrow |triangles|$
for $i = 0;\ i < |planeEvents|$ **do**
  $t_{start} \leftarrow t_{planar} \leftarrow t_{end} \leftarrow 0$
  $plane \leftarrow planeEvents[i]$
  while $i < |planeEvents| \wedge planePositionEqual(plane, planeEvents[i]) \wedge eventType(planeEvents[i]) = ENDING$ **do**
    $t_{end} \leftarrow t_{end} + 1$
    $i \leftarrow i + 1$
  end
  while $i < |planeEvents| \wedge planePositionEqual(plane, planeEvents[i]) \wedge eventType(planeEvents[i]) = PLANAR$ **do**
    $t_{planar} \leftarrow t_{planar} + 1$
    $i \leftarrow i + 1$
  end
  while $i < |planeEvents| \wedge planePositionEqual(plane, planeEvents[i]) \wedge eventType(planeEvents[i]) = STARTING$ **do**
    $t_{start} \leftarrow t_{start} + 1$
    $i \leftarrow i + 1$
  end
  // faces that end in the current plane lie only before the plane. As planar faces have their own counter, they are removed from $|T_g|$
  $|T_g| \leftarrow |T_g| - t_{planar} - t_{end}$
  $|T_p| \leftarrow t_{planar}$
  // tracking the plane with minimal cost is omitted for readability reasons here
  $cost = evaluateSAH(|T_l|, |T_p|, |T_g|, AABB, plane)$
  // triangles that start at the current plane will lie before the next plane to be parsed, update for next iteration
  $|T_l| \leftarrow |T_l| + t_{start} + t_{planar}$
end
return *split plane with minimal cost found by evaluateSAH*

$O(n \cdot \log(n))$ Algorithm

The $O(n \cdot \log(n)^2)$ implementation [13] still requires generating the plane events and sorting them every time splits for a new AABB are evaluated. To omit this step, we can adapt Algorithm 1 to only generate the sorted plane event list once and then pass it down to child nodes to perform future splits with it. For that to work, we must incorporate plane events with arbitrary orientation into a single list. The modified algorithm is depicted in Algorithm 2.

Generating the triangle sets for each halfspace is no longer necessary because we reuse the plane event list in both child nodes. However, we must split the list to include only relevant events for each node. We first classify the triangles as we did in the $O(n \cdot \log(n)^2)$ implementation. We build two plane event lists $events_l$ and $events_g$ with plane events of faces that lie only in T_l and T_g, respectively. Their order must be maintained throughout this process. Events that stem from faces that lie in both halfspaces must be voided since their faces would require clipping first, as explained in Subsection 2.4.2. The affected faces are clipped to the child node's AABBs, generating new plane events. This results in two more plane event lists $events_{lc}$ and $events_{gc}$. These lists are sorted first and then merged into $events_l$ and $events_g$ via one mergesort iteration. The resulting plane event lists can now safely be passed to the child nodes.

Algorithm 2: $O(n \cdot \log(n))$ plane event parsing algorithm

Input : **AABB**: the axis-aligned bounding box of the current node to be split
triangleAmount: the number of triangles in the current AABB
planeEvents: sorted plane event list of planes with all dimensions
dimensions: dimensions of the current space

`// initialize counters for every dimension since non-parallel planes cannot be compared`

forall $d \in dimensions$ **do**
  `// there are no triangles before the first split plane candidate`
  $T_{lesser,d} \leftarrow T_{planar,d} \leftarrow 0$
  `// all triangles lie after the first split plane candidate`
  $T_{greater,d} \leftarrow |triangles|$
end

for $i = 0;\ i < |planeEvents|$ **do**
  $t_{start} \leftarrow t_{planar} \leftarrow t_{end} \leftarrow 0$
  $plane \leftarrow planeEvents[i]$
  while $i < |planeEvents| \land planeDimensionEqual(plane, planeEvents[i]) \land planePositionEqual(plane, planeEvents[i]) \land eventType(planeEvents[i]) = ENDING$ **do**
    $t_{end} \leftarrow t_{end} + 1$
    $i \leftarrow i + 1$
  end
  while $i < |planeEvents| \land planeDimensionEqual(plane, planeEvents[i]) \land planePositionEqual(plane, planeEvents[i]) \land eventType(planeEvents[i]) = PLANAR$ **do**
    $t_{planar} \leftarrow t_{planar} + 1$
    $i \leftarrow i + 1$
  end
  while $i < |planeEvents| \land planeDimensionEqual(plane, planeEvents[i]) \land planePositionEqual(plane, planeEvents[i]) \land eventType(planeEvents[i]) = STARTING$ **do**
    $t_{start} \leftarrow t_{start} + 1$
    $i \leftarrow i + 1$
  end
  `// faces that end in the current plane lie only before the plane. As planar faces have their own counter, they are removed from` $T_{greater,d}$
  $T_{greater,d} \leftarrow T_{greater,d} - t_{planar} - t_{end}$
  $T_{planar,d} \leftarrow t_{planar}$
  `// tracking the plane with minimal cost is omitted for readability reasons here`
  $cost = evaluateSAH(T_{lesser,d}, T_{planar,d}, T_{greater,d}, AABB, plane)$
  `// triangles that start at the current plane will lie before the next plane to be parsed, update for next iteration`
  $T_{lesser,d} \leftarrow T_{lesser,d} + t_{start} + t_{planar}$
end
return *split plane with minimal cost found by evaluateSAH*

3. Related Work

In this chapter, we explore alternative approaches instead of using KD-trees and elaborate on the core functionality of the software we try to improve. Additionally, we review existing KD-tree libraries and justify writing our own.

3.1. Polyhedral Gravity Model

3.1.1. Theory

While it is simple to calculate the gravitational force of a sphere, it becomes complicated when looking at objects of arbitrary shape. For that very purpose, Schuhmacher [2] presents a C++ 17 library implementing the polyhedral gravity model based on the line integral approach by Tsoulis et al. [16]. The program only requires the object's shape represented as a triangle mesh and the constant density of the material it is made of as inputs. However, it is imperative that the surface normal vectors, calculated using the cross product, point away from the object [17]. The orientation of a face's normal vector is dependent on the ordering of the face's vertices. Should all normal vectors point into the object instead, a simple sign flip can automatically correct the error during calculation. Mixed normal orientations lead to wrong results. Therefore, the application should be able to verify the mesh before the gravity model is evaluated. Schuhmacher utilizes the Möller-Trumbore algorithm, introduced earlier in Subsection 2.2.2. In order to determine the normal vector orientation of a surface part of the object O, we need the midpoint P of the surface. P is calculated by converting surface barycentric coordinates $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ into Cartesian coordinates (Subsection 2.2.1). Afterward, we shift P into the direction the normal is pointing by:

$$P^* = P + \epsilon \cdot n, \quad 0 < \epsilon \in \mathbb{R} \tag{3.1}$$

We then use P^* as ray origin and shoot the ray with direction n into the scene. Let k be the number of intersections that occurred with O, then n is pointing away from O if k is even. Otherwise, it is pointing into O. This is because the object's volume is finite, and the ray traverses an infinite distance throughout the scene. Thus, it can be inferred that should the ray enter into the object, then it also has to exit at some point. Both of these events are recorded as intersection points by performing Möller-Trumbore. Should the ray originate inside the object, the first intersection point will be the first occurrence of the ray exiting the object. We will not record an entry point since the ray has already been inside O. Afterward, the ray may enter O again, but then we will always record another intersection when the ray exits O, resulting in the total number of intersections remaining uneven. Subsequently, we can determine the relative position of P^* with respect to O and, based on that, the orientation of n itself. The original work by Schuhmacher parallelizes ray-triangle intersection but stays with the naive $O(n^2)$ approach. This thesis substitutes this

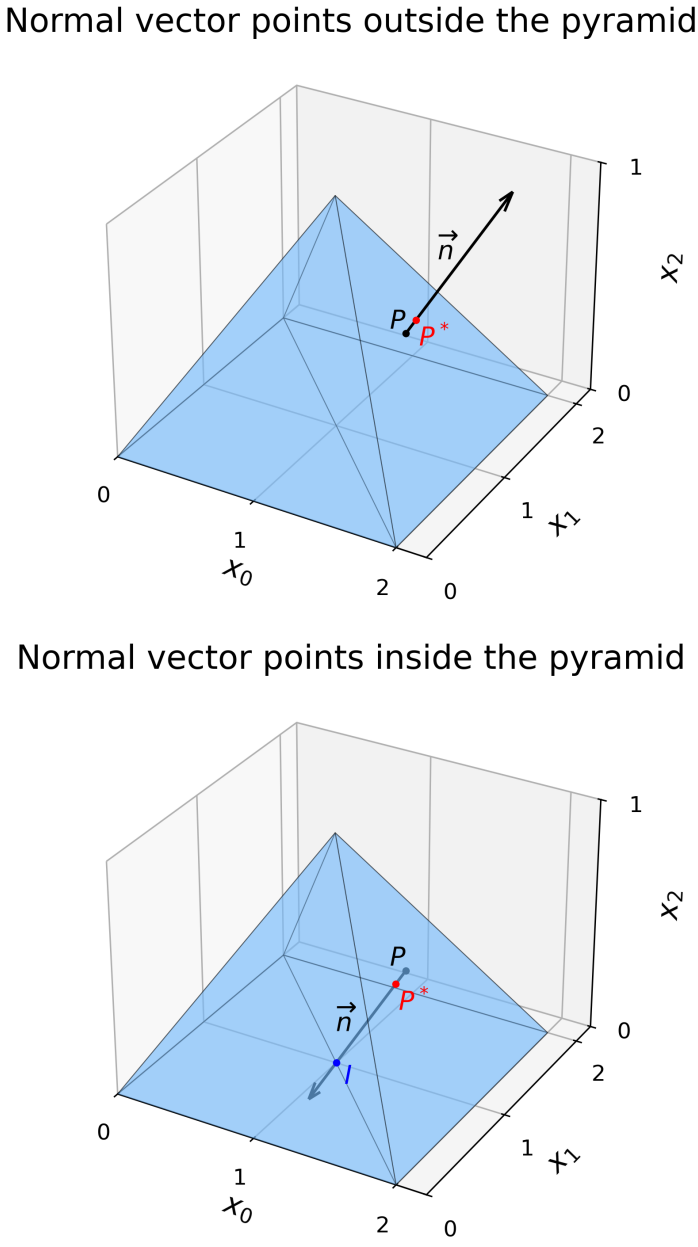


Figure 3.1.: Ray Intersection to check the normal orientation of a face in a pyramid mesh

with a more sophisticated ray-intersection algorithm using KD-trees, intending to increase performance.

3.1.2. Technology

The project uses a codebase written in C++17 and is available as a library built with CMake. It also includes CMake targets for testing and benchmarking using Google Test[1] and Google Benchmark[2]. Internally, the program is parallelized with NVIDIA's C++ Thrust[3] framework. Thrust is built on top of existing parallelization frameworks. There exist multiple implementations based on standard CPP serialized standard, OpenMP[4] or Intel TBB[5]. Various 3D mesh files are supported by including the Tetgen[6] library.

3.2. BSP-Trees

The BSP-tree, the binary space partition tree, is a modified binary search tree to support higher dimensional queries. Strictly speaking, the KD-tree, shown in Section 2.4, is a variant of the BSP-tree, in which split planes are constrained to be axis aligned. In that regard, BSP-trees can choose their split planes more freely. Ize et al. [18] present three runtime optimizations that are no longer trivially applicable because of the arbitrarily oriented split planes and bounding volumes: [18]

1. Measuring the distance between a point and a split plane as shown in Equation 2.16 can not be optimized to avoid the dot product.
2. Split plane positions aren't limited to $O(n)$ positions, but $O(n^3)$.
3. Surface area of bounding volumes, required by the SAH, defined in Subsection 2.4.1, can not be calculated as trivially.

However, the authors also point out the advantages and possibilities the KD-tree lacks. BSP-trees are highly adaptive to non-axis aligned scene objects and inhibit numerical inaccuracies better. The BSP-tree can be built like a KD-tree using axis-aligned planes. The construction algorithm only deviates should it be more beneficial to use non-axis aligned planes for a particular split. This, in theory, constructs a similarly or more optimal tree than a pure KD-tree. Ize et al. [18] showed that it is possible to construct BSP-trees that rival conventional KD-trees in intersection times by limiting split plane positioning and providing BSP-tree-specific optimizations. This comes at the cost of not building the most optimal BSP-tree theoretically possible. Despite these modifications, the authors stated that the KD-tree's $O(n \cdot \log(n))$ build process remains faster, making BSP-trees unsuitable for this project. The complexity of building BSP-trees stays beneath $O(n^2)$. [18]

[1] `https://github.com/google/googletest`, last accessed: 15.01.2025
[2] `https://github.com/google/benchmark`, last accessed: 15.01.2025
[3] `https://github.com/NVIDIA/thrust`, last accessed: 15.01.2025
[4] `https://www.openmp.org/resources/refguides/`, last accessed: 21.01.2025
[5] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html`, last accessed: 21.01.2025
[6] `https://github.com/libigl/tetgen`, last accessed: 15.01.2025

3.3. Octrees

The Octree is another variant of the BSP-tree. Each cell is an axis-aligned cube. Each parent cell in the tree is subdivided into eight smaller equally sized cube cells. This can be achieved by placing the split planes at the half points of the cube's edges. By not evaluating the underlying geometry, the construction process is faster than other approaches that aim for optimal solutions. This makes them desirable when dynamic data has to be modeled because reconstruction happens frequently then [19]. An optimal solution is ideal because the data used by Schuhmacher et al. [2] consists exclusively of static polyhedrons represented through triangle meshes. The advantages of an Octree do not come to fruition here. [12, p.14][9]

3.4. iKD-trees and cKD-trees

The implicit KD-tree (iKD-tree) is a static KD-tree optimized for minimum memory consumption [20]. The basic idea is to constrain each leaf node to have the same distance t to the root node. The tree is then called "balanced". By doing so, the tree can be stored in an array with the size of $2^t + 1$. This eliminates the need to store pointers required by a regular KD-tree. The compact KD-tree (cKD-tree) is built with a 2D iKD-tree as a base, compressing it further. The cell a point lies in is encoded using spiral encoding. Spiral encoding of a cell is the distance measured in cell sizes from the cell to a root cell in a spiral arrangement. For clarification, refer to Figure 3.2. By storing the encoded distance of the

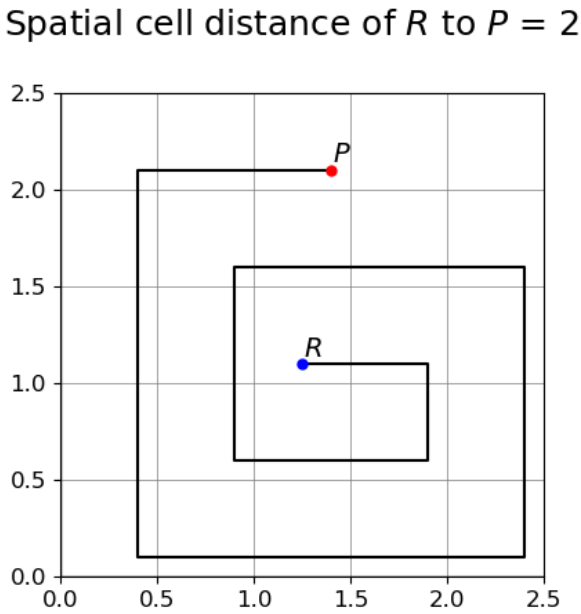


Figure 3.2.: Spiral encoding of a point P with root R

cell to its parent node, we can record the iKD-tree structure with less memory consumption in an array as a sequence of integers. Another encoding concept called direct access codes

(DAC) is deployed to enable direct readout of elements in the sequence. An array also provides this functionality. However, DACs allow the storage of elements of variable length without unnecessary memory overhead. While this improves the memory footprint, it takes a toll on runtime by introducing the extra complexity. The standard KD-tree algorithms are still viable but need to be adjusted. This thesis builds on top of the work by Schuhmacher et al. [2] to improve ray tracing runtime. The nature of their work does not guarantee balanced KD-trees, questioning whether memory improvements can be achieved through iKD-tress. The objects they operate on are three-dimensional polyhedrons, ruling out the usage of cKD-trees entirely. Even though the primary goal of this work is runtime improvements, considering a hybrid solution of a regular and iKD-tree might provide a better trade-off. We postpone this task to future work.

3.5. Existing KD-Tree Implementations

Many well-known KD-tree libraries are tailored to execute queries from nearest neighbors and point ranges. They inherently do not support ray tracing. It is possible to adapt the libraries accordingly. However, they will likely perform more poorly than a custom implementation. This is because they are not optimized for ray tracing. This eliminates common choices CGAL[7], nanoflann[8] and FLANN[9]. Intel's Embree[10] ray tracing suite supports building KD-trees for raytracing. However, it is quite complex and provides many other features leading to obfuscation. There are various implementations available on GitHub. However, to our best knowledge, they do not follow coding best practices and are not made by computer scientists. We decided to implement our custom KD-tree library because we demand compatibility with ray tracing while retaining simplicity and quality.

[7] `https://github.com/cgal/cgal`, last accessed: 26.01.2025
[8] `https://github.com/jlblancoc/nanoflann`, last accessed: 26.01.2025
[9] `https://github.com/flann-lib/flann`, last accessed: 26.01.2025
[10] `https://www.embree.org/`, last accessed: 29.01.2025

Part II.

Implementation, Verification and Results

4. Architecture

This chapter showcases the software architecture using a UML class diagram (Figure 4.1). We will also explore the contribution of each class to the program's functionality and explain design choices.

4.1. Polyhedron

This class represents the triangle mesh. It contains all vertices along with their coordinates stored in a list. A vertex triplet defines the faces. Instead of entirely copying the vertices from the vertex list, they are only referenced via list indexing. It is possible to supply the mesh in different file formats by utilizing the Tetgen library. The library is called internally by the `Polyhedron` constructor via an adapter. Additionally, the class contains the KD-tree to delegate intersections.

4.2. KDTree

The `KDTree` class acts as a root class for the KD-tree data structure by providing a reference to the root node. The program avoids unnecessary computations by building only necessary tree nodes. Thus, we implement a lazy loading approach that constructs tree nodes as needed during runtime. This results in the class storing an instance of `SplitParam`, described in Section 4.3. It is later used during node construction and removed afterward. This class also serves as the facade (refer to facade pattern [21, p.210]) for the underlying tree structure.

4.3. SplitParam

This class solely exists to enable lazy loading. It stores information about the current scene cell needed to construct new tree nodes. Stored data involves the vertices and faces of the polyhedron initially provided by the Polyhedron implementation, presented by Section 4.1. It also includes which split plane selection algorithm to use and a list of plane events or triangle faces bound in the current node's bounding box. Whenever a child node is built, the `SplitParam` object is copied and tailored for the node and then passed down. After all direct child nodes have been built, the parent's `SplitParam` instance is destroyed.

4.4. TreeNode

`TreeNode` is an abstract class and superclass to `SplitNode` (Section 4.5) and `LeafNode` (Section 4.6). That way, the tree hierarchy can be built without discerning the stored node's type. This is especially useful since the type of node to be built depends on the results of the split plane evaluation. Refer to Subsubsection 2.4.1 for more details.

4.5. SplitNode

`SplitNode` implements the functionality of the inner nodes of the tree. They have child nodes and are responsible for delegating intersection calls to them. When performing ray intersection tests dictated by Section 2.1, the split node is responsible for detecting hits with any of the two halfspaces it is responsible for. Detection is realized by a ray-box intersection test with the split node's AABB (Subsection 2.3.1) and a ray-plane intersection with the split node's split plane (Subsubsection 2.3.1). Comparison of the resulting three t-values yields the desired information. Before a split node's child is accessed, the parent node first checks whether the node has been built already. If not, the split node calls a factory method responsible for split plane building and constructing a new `TreeNode`. The necessary data is generated by copying and adjusting the node's `SplitParam` instance. Should the child node have already been built prior, its reference should be used for delegation.

4.6. LeafNode

`LeafNode` instances represent, as their name states, the leaf nodes of the KD-tree. They do not subdivide cells further but are directly responsible for a set of triangle faces. Intersection calls are executed on the triangle faces according to Section 2.2. The bound triangles are saved in a `SplitParam` object, which is persistent in contrast to the behavior in `SplitNode`.

4.7. Plane

The class defines an axis-aligned plane. The plane's orientation is not represented by a normal vector but by the coordinate axis into which the plane spans. Its internal representation is an enum class. Additionally, the anchor point of the plane is sufficiently specified by only providing the coordinate that is constant for all points lying on the plane. These space optimizations can only be made because the plane is axis-aligned.

4.8. PlaneEvent

The abstract class encodes the plane event structure described in Subsubsection 2.4.3. Theoretically, omitting the triangle identifier that generated the event is possible. However, it provides a more elegant alternative to the naive approach of classifying triangle positions.

4.9. Box

This class provides the implementation of the AABB, elaborated on by Section 2.3. Two triplets of floating point values define $minPoint$ and $maxPoint$. The ray-box intersection logic and convenience functions, like finding the AABB for a set of vertices, are also encapsulated there.

4.10. PlaneSelectionAlgorithm

This is an abstract superclass for a family of plane evaluation strategies required by the strategy pattern [21, p.368]. By deploying this software pattern, we can test and benchmark the approaches described in Subsection 2.4.3 and verify their theoretical runtime complexity. Another advantage is the reduced code duplication, which warrants this approach's use. The plane selection strategy for the current KD-tree is passed along as a field in `SplitParam`, depicted in Section 4.3.

4.11. NoTreePlane

Regardless of the scene, this evaluation strategy will always report infinite costs for the best possible split plane and return an arbitrary plane. During the evaluation of the termination criterion, refer to Subsubsection 2.4.1; the algorithm will then discard the returned plane due to $cost_{leaf}$ being lower. This strategy builds a KD-tree consisting of a single `LeafNode`, outlined by Section 4.6, and is used to check whether building the KD-tree improves runtime.

4.12. SquaredPlane

This subclass implements the naive plane evaluation strategy by simply nesting two for-loops described by Subsubsection 2.4.3. As it is the simplest tree-building method without optimizations, it is also less prone to code mistakes. This makes it ideal for verification of more sophisticated strategies.

4.13. PlaneEventAlgorithm

This abstract class provides functions used by plane evaluation strategies that utilize plane events. Those being `LogNPlane` and `LogNSquaredPlane`, defined later in Section 4.14 and Section 4.15. The motivation for this abstraction is the reduction of code duplication.

4.14. LogNSquaredPlane

This plane evaluation strategy has a runtime complexity of $O(n \cdot \log(n)^2)$ by utilizing `PlaneEvent` instances, described above in Section 4.8, to improve runtime. This algorithm version is still unoptimized and requires the generation of new plane event lists and sorting at every node.

4.15. LogNPlane

This implements the final and fastest plane evaluation strategy. It optimizes `LogNSquaredPlane`, introduced by Section 4.14, to reuse generated plane events and maintain their order throughout splicing, omitting unnecessary sorting.

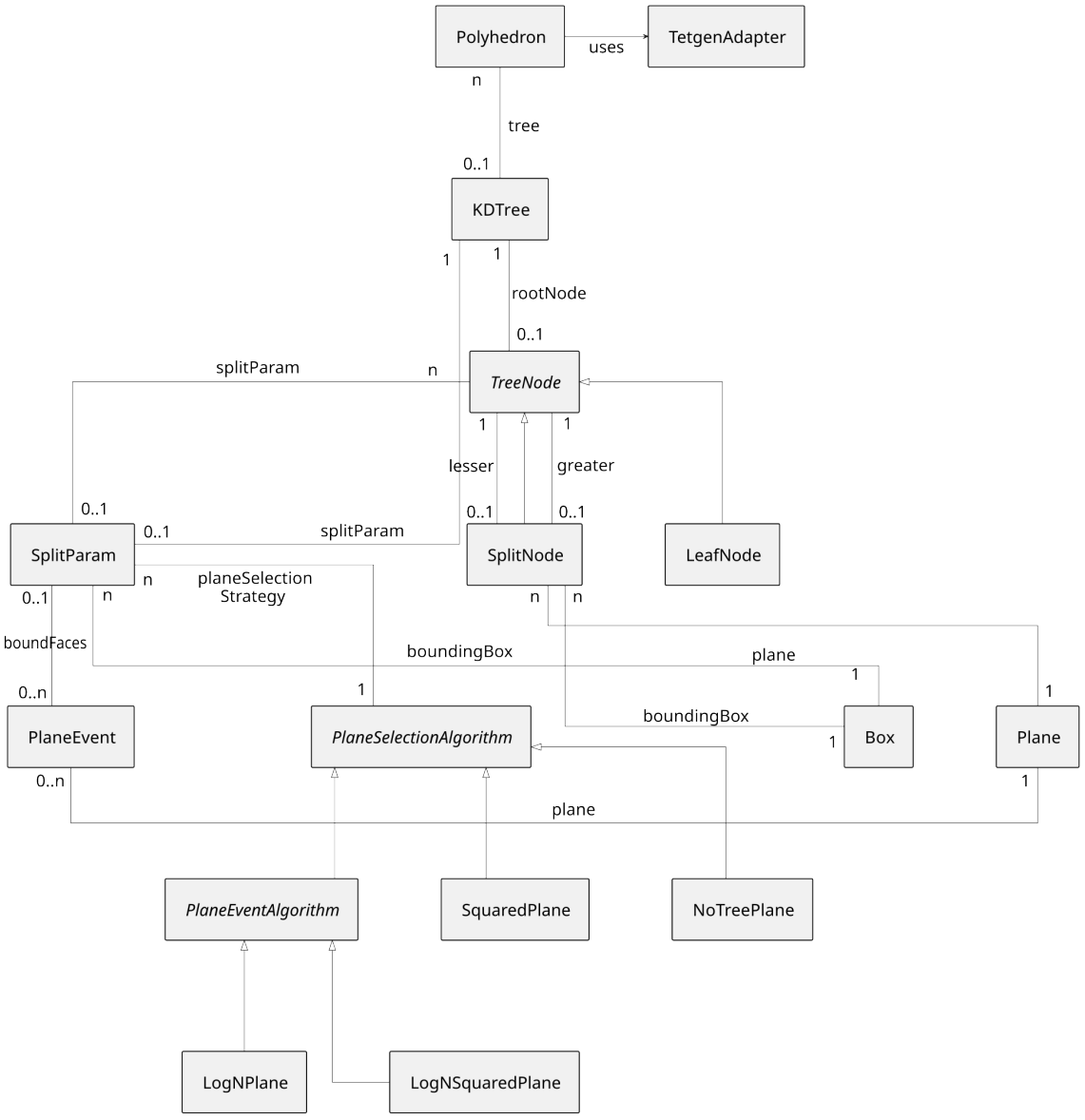


Figure 4.1.: UML Class Diagram (abstract classes are printed in italic font)

5. Usage

In this chapter, we briefly explain how to use our KD-tree implementation.

```
1  // vector of vertices: A vertex is an array of three coordinates
2  const std::vector<std::array<double, 3>> vertices{...};
3  // vector of faces: A face is defined by storing the indices of the vertices
4  const std::vector<std::array<unsigned long, 3>> faces{...};
5
6  const std::array<double, 3> origin{0,0,0};
7  const std::array<double, 3> rayDirection{0.5,0.5,1};
8
9  KDTree kdTree{vertices, faces};
10 std::set<std::array<double ,3>> intersectionPoints{};
11 // calculate intersection points and add them to intersectionPoints
12 kdTree.getFaceIntersections(origin, rayDirection, intersectionPoints);
```

Listing 5.1: Usage of the KD-tree implementation in C++17

The `KDTree` is built on top of objects represented by triangle meshes in a scene. We define a triangle vertex as an array of three `double` values stored in a `std::array`. Triangle faces are defined similarly, with the difference of the vertex indices being stored instead of coordinates. The vertices and faces of the triangles are stored in instances of `std::vector`. The constructor of `KDTree` takes the two vectors as arguments. The KD-tree can now be used to calculate intersection points. We call the `KDTree`'s `getFaceIntersections` function to do that. It requires the origin point of the ray, the ray direction, and an instance of `std::set` to store the intersection points in, passed as parameters. Refer to the code in Listing 5.1.

6. Testing

Verification is done with testing via Google Test[1]. We mainly use three techniques, fuzzing [22], regression testing [23] and integration testing:

Fuzzing

A random face F on the polyhedron is chosen to be intersected. Then, we generate two random barycentric coordinates $u \in [0, 1]$ and $v \in [0, 1 - u]$ relative to F. Per the definition of u and v, the point P they encode lies on the surface of F. We choose an arbitrary point of origin O. It becomes possible to construct $\overrightarrow{OP}$. We then pass O and $\overrightarrow{OP}$ to the KD-tree and let it calculate all intersection points with the polyhedron. Per construction, should the tree work correctly, P must be included in the set of intersection points returned. We repeat this procedure without rebuilding the tree. The random generator must be seeded with a constant value to guarantee reproducibility. This type of test aims to determine whether intersections are performed correctly with the tree the algorithms build.

Regression Testing

This type of test assumes that the $O(n^2)$ KD-tree building algorithm is correctly implemented, returning optimal split planes. A full KD-tree is built using an arbitrary building algorithm. Afterwards, the resulting split planes are recalculated using the $O(n^2)$ algorithm. Since both algorithms are supposed to return the optimal KD-tree for a scene, the split planes they produce should also be equal.

Integration Testing

Schuhmacher et al. [2] provide test cases for identifying faulty faces with wrong vertex ordering in the polyhedron. The motivation behind these tests is elaborated on in Subsection 3.1.1. The main principle behind the checks is ray tracing. We can reuse these tests by integrating the KD-tree into the ray tracing algorithm of Schuhmacher's `Polyhedron` class.

[1] `https://github.com/google/googletest`, last accessed: 21.01.2025

7. Runtime Measurements

This chapter measures the performance of the algorithms described by Subsection 2.4.3 and Chapter 4. For the test data set, we take a triangle mesh of the Eros asteroid[1] with 14744 faces. We additionally scale the mesh up and down to achieve variant problem sizes while retaining the overall geometry and shape. Upscaling works by splitting an existing triangle face into multiple and incorporating it in the new mesh. On the contrary, downscaling combines several faces into a single one.

For testing, a x86_64 Manjaro Linux (kernel version 6.11.10) machine with an AMD Ryzen 5 3600 4.2 GHz 6-Core processor, 16 GB RAM, and an AMD Radeon RX 570 Series graphics card was used. The measurements were taken using Google Benchmark.

We adapt the problem to be measured closely from the workings behind the polyhedral gravity model introduced in Subsection 3.1.1. Thus, every implementation is tasked with shooting one ray through every face of the mesh one at a time. Then, the benchmark calculates intersection points. By intersecting every face of the mesh instead of just a fraction, we can guarantee that a full KD-tree is built. Otherwise, the implemented construction algorithms using lazy loading may appear faster by performing less work.

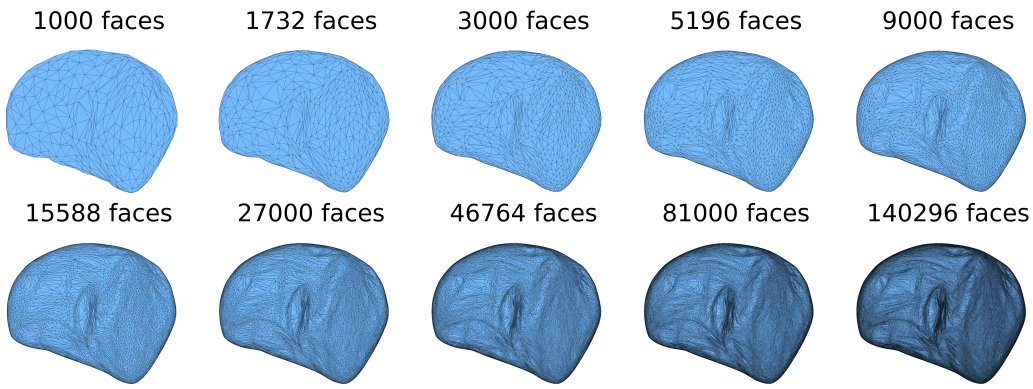


Figure 7.1.: Scaled Eros meshes

[1] `https://github.com/darioizzo/geodesyNets/blob/master/3dmeshes/eros.pk`, last accessed: 21.01.2025

7.1. Sequential Execution

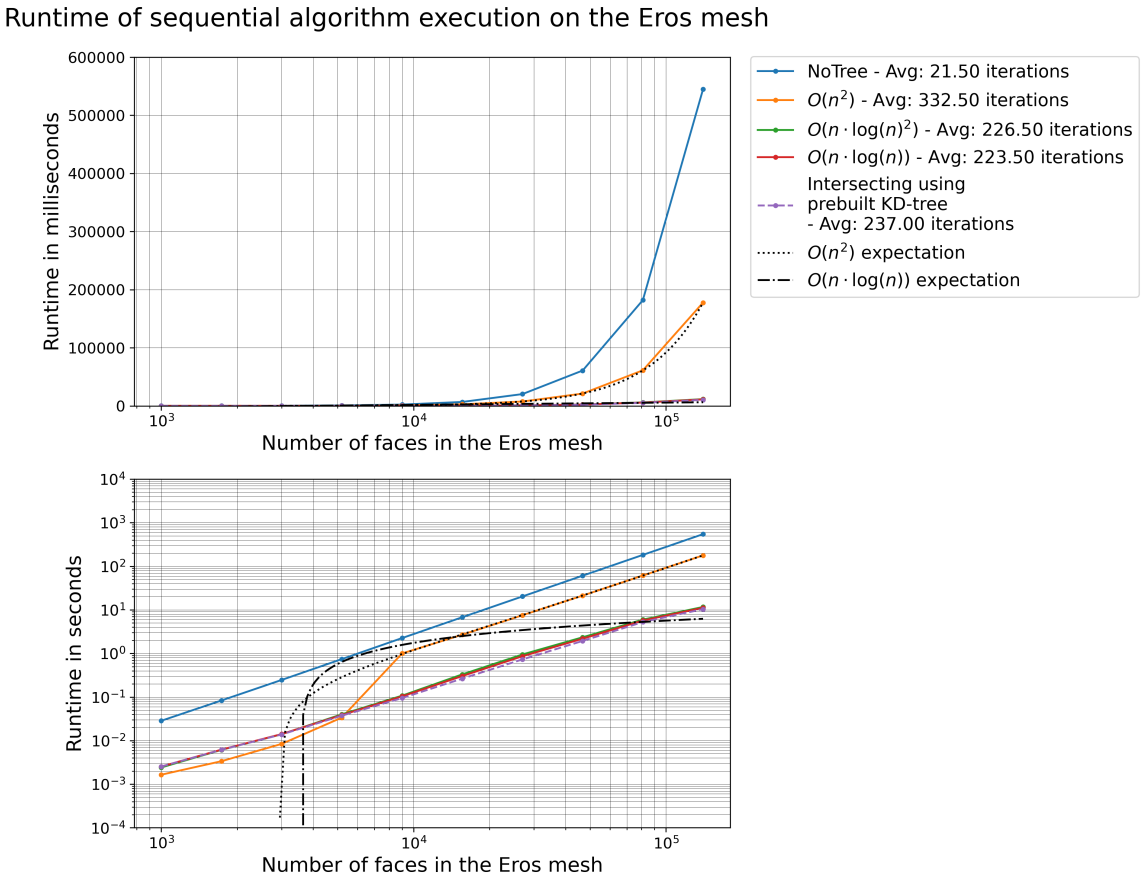


Figure 7.2.: Sequential execution runtime measurements of different algorithm classes on scaled Eros mesh

Figure 7.2 shows the measurement line graph taken when the algorithms were executed on a single thread. Due to the poor distinction between runtimes for meshes with less than 10,000 faces, we also provide the same measurements on a logarithmic scale. Interpreting the results, building a KD-tree is the better choice for meshes of arbitrary size when we expect many intersections to be computed. It is worth noting that both intersection and tree-building processes are incorporated into the measurements. By caching the built KD-tree, all algorithms are expected to perform better in the long term than simply repeating intersections without a tree every time. The prebuilt tree performs slightly better for our data set than the optimal $O(n \cdot \log(n))$ algorithm. Thus, the expectation holds. All variants outperform NoTree, with the fastest reducing runtime by more than half. Algorithm $O(n \cdot \log(n))$ is faster than the $O(n \cdot \log(n)^2)$ counterpart. However, the difference in the total time spent is barely noticeable. Hotspot analysis reveals that the $O(n \cdot \log(n))$ algorithm spends 37% of the total runtime and 86% of all KD-tree operations calculating intersection points using the Möller-Trumbore algorithm from Subsection 2.2.2. This algorithm is exclusively used in LeafNodes and does not impact tree building. To verify, we ran separate measurements,

where the program only built the KD-tree without performing triangle intersection. The results are plotted in Figure 7.3. Comparison with Figure 7.2 yields that the KD-tree build time of the $O(n \cdot \log(n))$ construction algorithm stays consistently under 2 percent of the whole intersection check runtime.

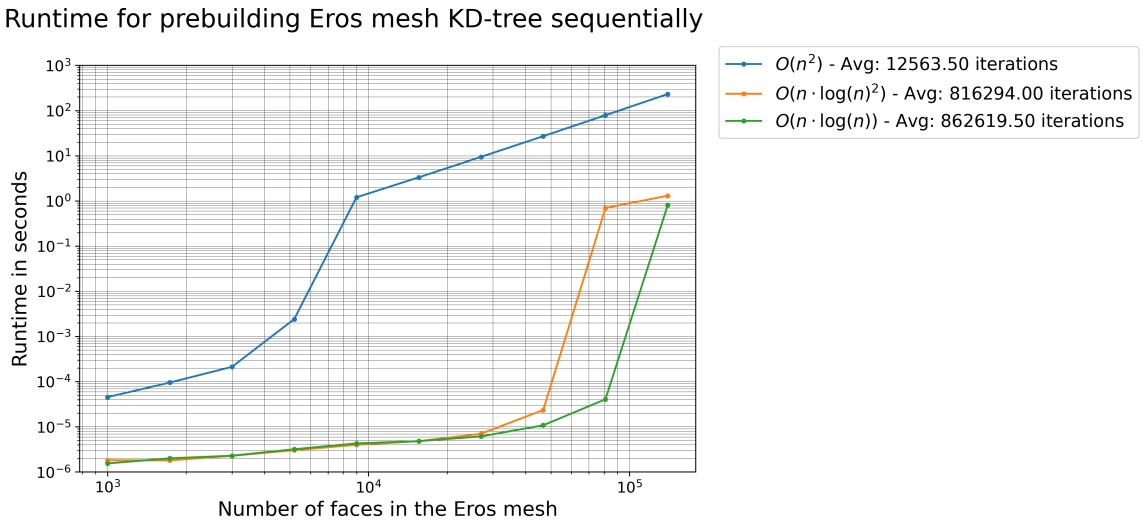


Figure 7.3.: Building complete KD-tree sequentially without intersecting

Our observations are based on measurements performed on the Eros mesh. We execute the same benchmarks on the sphere mesh from Figure 2.1 to show that they also apply to other meshes. The findings are rendered in Figure 7.4

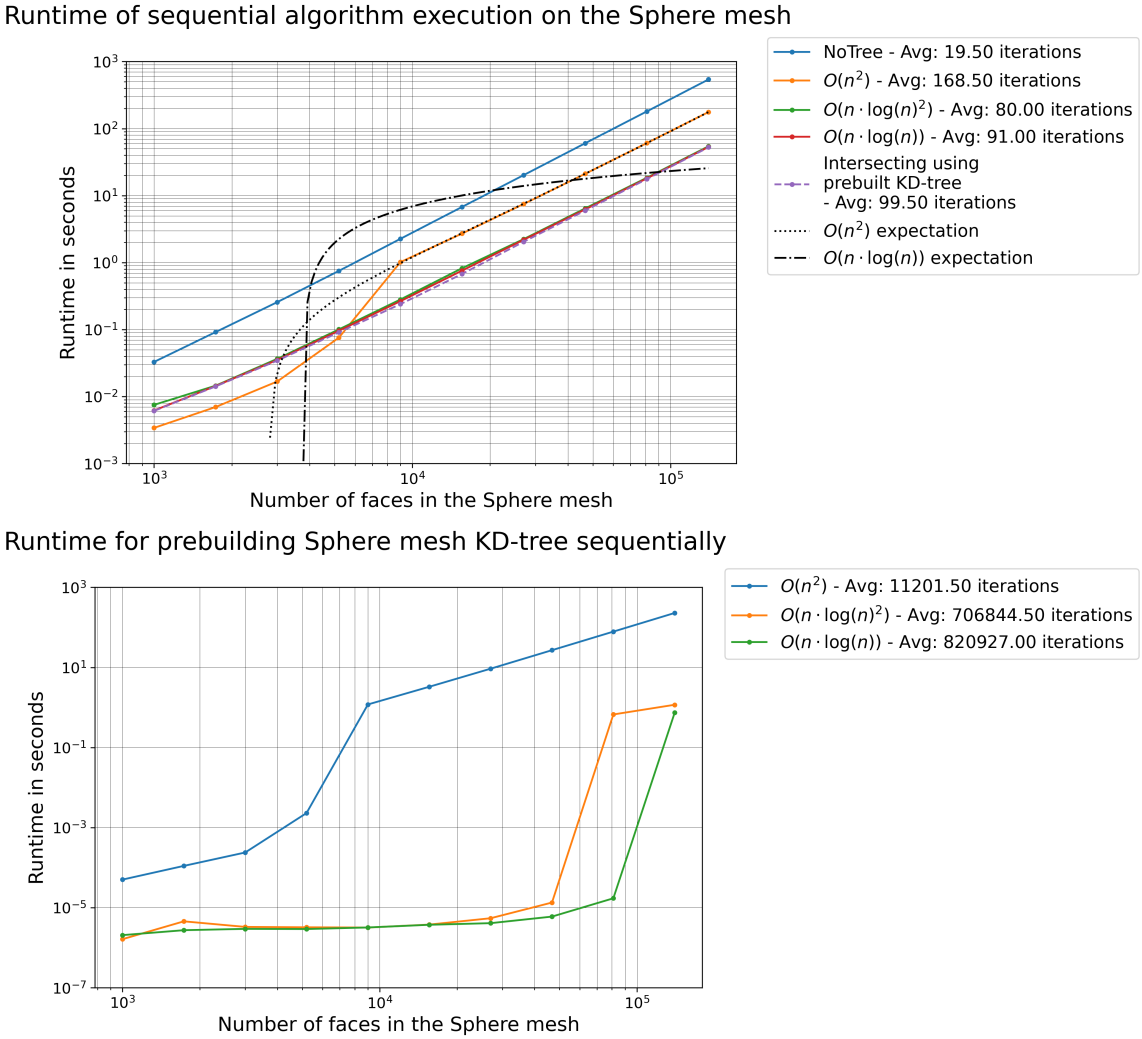


Figure 7.4.: Benchmarks executed on the Sphere mesh

7.2. Parallel Execution

Our implementation is thread-safe for queries and set up to use multithreading internally during tree building. Testing showed that allocating threads for both processes simultaneously leads to thread starvation. Usually, this is not a problem since the KD-tree is entirely built before intersections are performed. Through lazy loading, this does not apply to our application. We limit ourselves to deploying parallelization techniques only in tree-building processes to resolve this problem. Figure 7.5 depicts runtime measurements with multithreading enabled. We can immediately confirm a significant performance speedup with roughly a factor of 10 to Figure 7.2. NoTree is now faster for larger meshes than the naive KD-tree build algorithm. This is because the underlying algorithm is better suited to be executed parallelly. Building the KD-tree is still the better choice, similar to Figure 7.2. We plotted the data again on a logarithmic scale to showcase time discrepancies for smaller meshes. The following results are recorded by benchmarking the $O(n \cdot \log(n))$

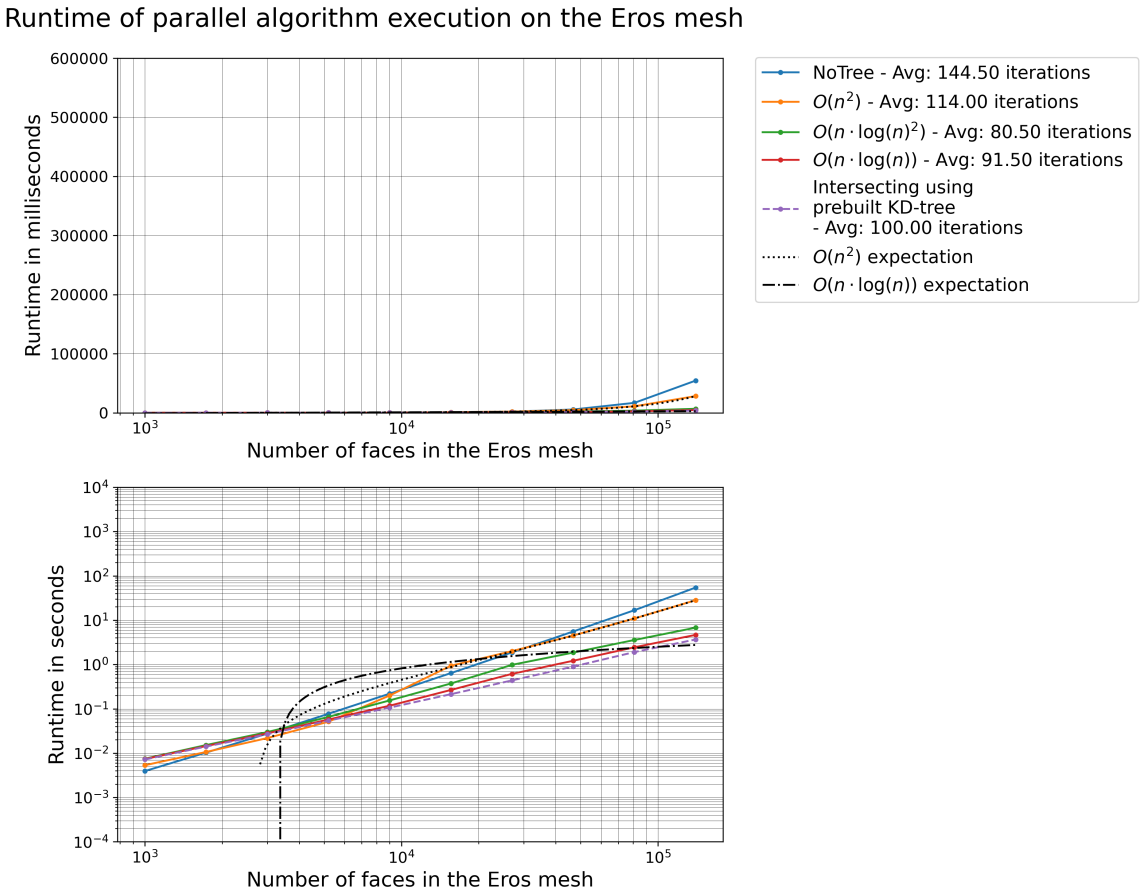


Figure 7.5.: TBB parallelized execution runtime measurements of different algorithm classes on scaled Eros mesh

plane evaluation strategy. Analysis using AMDμProf shows that CPU hardware thread downtime is low, illustrated by Figure 7.7. All threads are concurrently active for roughly 80% of the total runtime. Measuring with Valgrind Massif shows that 14 MB worth of input files from a polyhedron with 140,000 faces result in a KD-tree 350 MB large which is stored in heap memory.

As we did in the previous section, we measure again using the Sphere mesh for a basis. Refer to Figure 7.8.

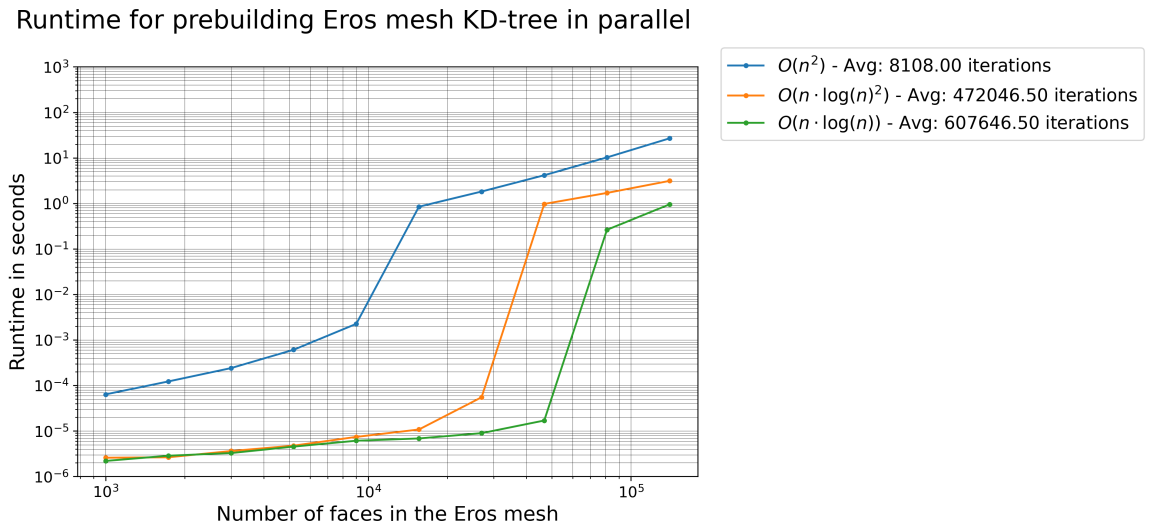


Figure 7.6.: Building complete KD-tree in parallel without intersecting during Eros benchmark

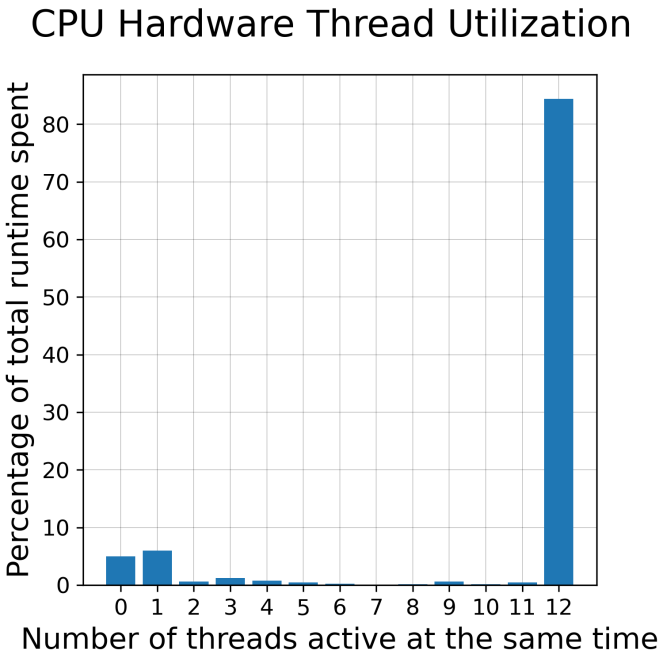


Figure 7.7.: Core utilization during Eros benchmark

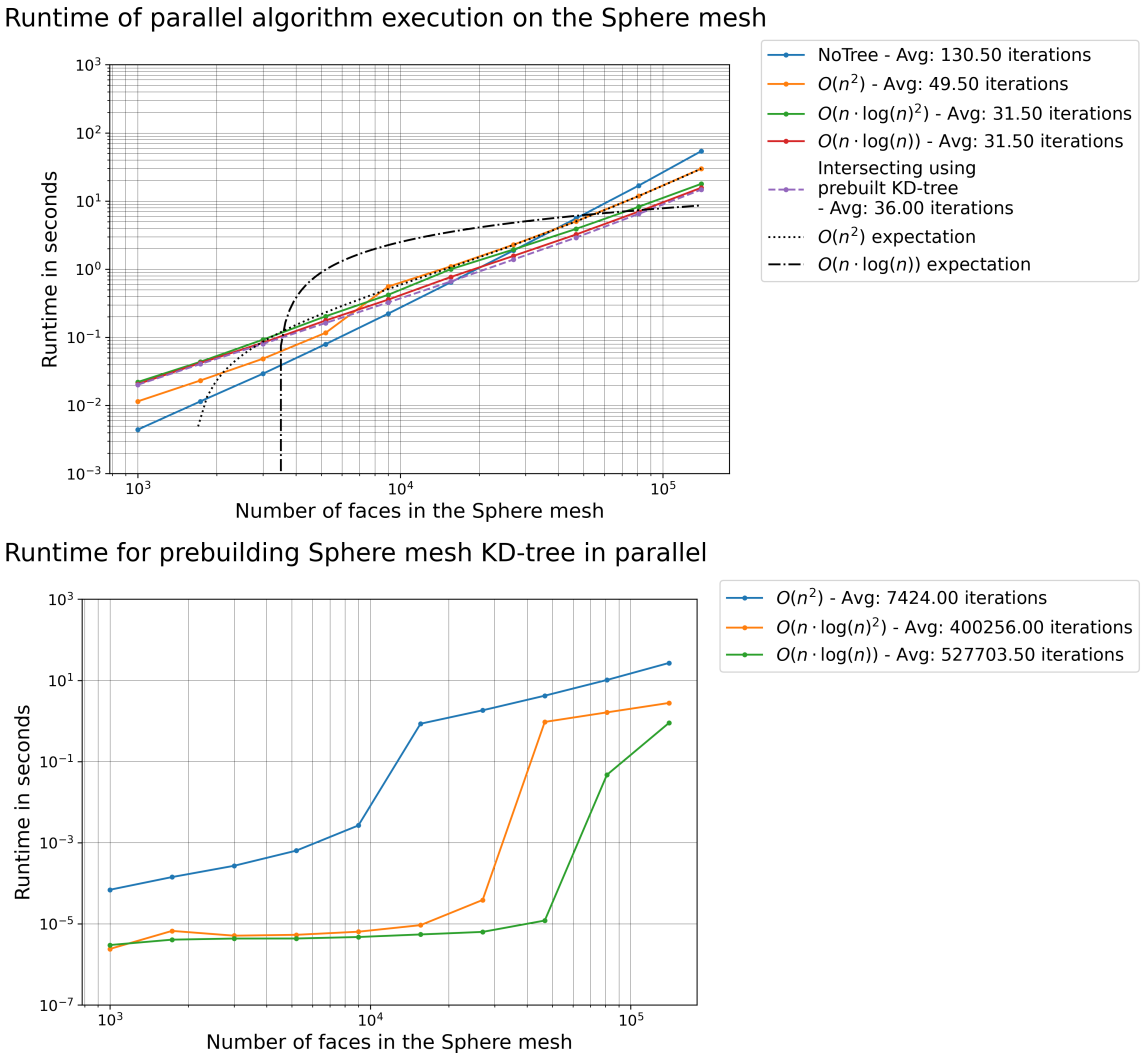


Figure 7.8.: TBB parallelized benchmarks executed on the Sphere mesh

Part III.

Conclusion

8. Summary

In this thesis, we have explored the shortcomings of ray tracing and the resulting motivation for KD-trees. We began by introducing the mathematical concepts and algorithms deployed by a KD-tree and discussed different approaches to construction. Afterward, we designed the software architecture and measured its performance. We rely on fuzzing, regression- and integration testing to guarantee the implementation's correctness.

The primary objective of this thesis is to develop a modern C++17 KD-tree implementation tailored for use in ray tracing algorithms. The implementation is designed to focus on simplicity and usability, featuring a straightforward interface and prioritizing essential functionalities. The codebase is modular and extensible, enabling integration into diverse software solutions and facilitating adaptation by other researchers.

The library employs lazy loading techniques to optimize performance and leverages CPU parallelization using the Thrust framework. The KD-tree construction algorithms are based on the methods described by Wald et al. [13], ensuring a robust foundation. The algorithms offer different runtime complexities, with the best performing reaching $O(n \cdot \log(n))$ computation times.

Throughout the development process, best practices in software engineering were strictly followed. Comprehensive documentation is provided through inline comments and Doxygen annotations, ensuring clarity and ease of use for developers.

9. Outlook

In this thesis, our primary focus is optimizing runtime without considering memory consumption. However, as discussed in Section 3.4, less memory-intensive variants of KD-trees exist. Implementing a hybrid approach that balances memory usage and runtime performance may be feasible without significantly impacting overall efficiency.

Current parallelization is limited to either queries on the KD-tree or the tree construction algorithms. We propose extending the program to enable switching to query parallelization dynamically once the tree has been built entirely or to a certain degree.

Thrust offers multiple parallelization backends. The current implementation supports only CPU parallelization, but further optimization might be achieved by adapting the algorithms to utilize the CUDA[1] backend.

Currently, the KD-tree is designed to subdivide three-dimensional space. However, existing code type definitions allow refactoring to generalize to arbitrary dimensions with moderate effort.

A primary design goal for this application is the ease of use. Thus, we also want to implement a pybind11[2] interface for the library and publish it via PyPi[3] and conda-forge[4]. To use our software, the developer does not need to know C++ anymore, targeting a broader audience.

[1] `https://developer.nvidia.com/cuda-toolkit`, last accessed: 30.01.2025
[2] `https://github.com/pybind/pybind11`, last accessed: 28.01.2025
[3] `https://pypi.org/`, last accessed: 28.01.2025
[4] `https://conda-forge.org/`, last accessed: 28.01.2025

Part IV.

Appendix

List of Figures

Bibliography

[1] B. Caulfield, "NVIDIA Unveils GeForce RTX, World's First Real-Time Ray Tracing GPUs," Aug. 2018.

[2] J. Schuhmacher, E. Blazquez, F. Gratl, D. Izzo, and P. Gómez, "Efficient polyhedral gravity modeling in modern C++ and python," *Journal of Open Source Software*, vol. 9, no. 98, p. 6384, 2024.

[3] H. Halmaoui and A. Haqiq, "Computer graphics rendering survey: From rasterization and ray tracing to deep learning," in *Innovations in Bio-Inspired Computing and Applications* (A. Abraham, A. M. Madureira, A. Kaklauskas, N. Gandhi, A. Bajaj, A. K. Muda, D. Kriksciuniene, and J. C. Ferreira, eds.), (Cham), pp. 537–548, Springer International Publishing, 2022.

[4] M. Botsch, M. Pauly, C. Rossl, S. Bischoff, and L. Kobbelt, "Geometric modeling based on triangle meshes," in *ACM SIGGRAPH 2006 Courses*, Siggraph '06, (New York, NY, USA), pp. 1–es, Association for Computing Machinery, 2006.

[5] T. Möller and B. Trumbore, "Fast, minimum storage ray/triangle intersection," in *ACM SIGGRAPH 2005 Courses on - SIGGRAPH '05*, (Los Angeles, California), p. 7, ACM Press, 2005.

[6] S. Lang, *Linear Algebra*. Springer Science & Business Media, 1987.

[7] A. F. Möbius, *Der Barycentrische Calcul, Ein Hülfsmittel Zur Analytischen Behandlung Der Geometrie (Etc.)*. Barth, 1827.

[8] V. Skala, "Barycentric coordinates computation in homogeneous coordinates," *Computers & Graphics*, vol. 32, pp. 120–127, Feb. 2008.

[9] C. Ericson, *Real-Time Collision Detection*. CRC Press, Dec. 2004.

[10] B. Smits, "Efficient bounding box intersection," *Ray tracing news*, vol. 15, no. 1, 2002.

[11] A. Williams, S. Barrus, R. K. Morley, and P. Shirley, "An efficient and robust ray-box intersection algorithm," in *ACM SIGGRAPH 2005 Courses*, Siggraph '05, (New York, NY, USA), pp. 9–es, Association for Computing Machinery, 2005.

[12] V. Havran, *Heuristic Ray Shooting Algorithms*. PhD thesis, Ph. d. thesis, Department of Computer Science and Engineering, Faculty of . . . , 2000.

[13] I. Wald and V. Havran, "On building fast kd-Trees for Ray Tracing, and on doing that in O(N log N)," in *2006 IEEE Symposium on Interactive Ray Tracing*, (Salt Lake City, UT, USA), pp. 61–69, IEEE, Sept. 2006.

[14] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, pp. 153–166, May 1990.

[15] I. E. Sutherland and G. W. Hodgman, "Reentrant polygon clipping," *Communications of The Acm*, vol. 17, pp. 32–42, Jan. 1974.

[16] D. Tsoulis, "Analytical computation of the full gravity tensor of a homogeneous arbitrarily shaped polyhedral source using line integrals," *Geophysics*, vol. 77, no. 2, pp. F1–F11, 2012.

[17] J. Schuhmacher, "Efficient polyhedral gravity modeling in modern C++," Master's thesis, Technical University of Munich, Dec. 2022.

[18] T. Ize, I. Wald, and S. G. Parker, "Ray tracing with the BSP tree," in *2008 IEEE Symposium on Interactive Ray Tracing*, pp. 159–166, Aug. 2008.

[19] D. Libes, "Modeling dynamic surfaces with octrees," *Computers & graphics*, vol. 15, no. 3, pp. 383–387, 1991.

[20] G. Gutiérrez, R. Torres-Avilés, and M. Caniupán, "cKd-tree: A compact kd-tree," *IEEE access : practical innovations, open solutions*, vol. 12, pp. 28666–28676, 2024.

[21] A. Shvets, *Dive into Design Patterns*. Refactoring.Guru, 2019.

[22] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *Acm Computing Surveys*, vol. 54, Sept. 2022.

[23] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *Acm Computing Surveys*, vol. 50, May 2017.