tf::tree: A General-Purpose Spatial Hierarchy for Real-Time Geometry Queries

Žiga Sajovic, Dejan Knez, Robert Korez
XLAB d.o.o.

***Abstract*—We present `tf::tree`, a high-performance, open-source spatial hierarchy for general-purpose geometry processing. It combines a data-oriented, balanced n-ary structure with in-place partitioning—driven by a pluggable framework of state-of-the-art selection algorithms—to achieve real-time construction speeds. Its novel Top-K sorted stack traversal outperforms traditional heap-based methods for proximity queries by up to $3\times$. The system supports real-time queries, including collision, proximity, and (self) intersections, even on models with millions of primitives, making `tf::tree` a versatile and efficient foundation for complex geometric tasks.**

***Index Terms*—Geometry processing, spatial data structures, intersection queries, proximity search, selection algorithms.**

I. Introduction

SPATIAL acceleration structures are central to modern geometry processing. They enable efficient queries that facilitate collision detection, proximity tests, and geometric intersections. These operations form the backbone of modeling, simulation, and interactive design systems, where they support key pipeline components such as automatic positioning, object matching, and boolean operations.

A. Problem Statement

Despite the abundance of acceleration structures, there remains a lack of practical tools that combine general-purpose applicability with real-time performance and seamless integration. Many existing libraries are optimized for narrow domains (e.g., ray tracing), lack control over traversal behavior, or impose structural and semantic constraints that hinder reuse in more general workflows.

What is needed is a spatial data structure that:

- Supports a broad range of queries (e.g. collision, intersection, self-intersection, nearness, picking), between a tree and a primitive, two trees, and a tree and itself.
- Maintains a balanced n-ary structure for predictable, shallow traversal depth.
- Scales to millions of primitives while offering real-time performance.
- Exposes internal mechanics (e.g., traversal and partitioning) for customization.
- Integrates easily into existing pipelines with minimal boilerplate.

A spatial structure that meets these criteria would unify a wide range of tasks under a single, consistent interface—enabling the same hierarchy to support actor picking, object-inside-manifold tests (on 3D models or 2D curves), primitive matching, pairwise intersections (for model cutting and boolean operations), and nearness reasoning. Fast construction times elevate such structures to algorithmic building blocks—used freely, like sorted arrays, within larger algorithms without incurring prohibitive performance costs. This reduces fragmentation in geometry pipelines and simplifies integration in systems where real-time interaction and general-purpose spatial logic must coexist.

B. Contributions

We introduce `tf::tree`, a spatial hierarchy designed for high-performance, general-purpose geometry queries. The system has been tested in production and offers both methodological and implementation contributions to the field.

Methodological Contributions:

- A novel Top-K sorted stack traversal for proximity queries, offering up to $3\times$ speedup over priority-queue-based approaches while maintaining practical cost-efficiency.
- An in-depth analysis of selection algorithms for spatial partitioning, with support for pluggable strategies such as `nth_element`, PDQSelect, Floyd–Rivest, and Alexandrescu's algorithm.

Implementation Contributions:

- An open-source implementation of `tf::tree`[1] that supports search and nearness queries required for interactive geometric modeling; enabling complex processing pipelines such as automatic positioning and boolean operations.
- Real time performance in both construction and query time, enabling use in interactive systems. See Figure 1 and Figure 2 for an overview.

II. Related Work

This section reviews prior work relevant to the design and implementation of spatial acceleration structures, focusing on volume hierarchies, selection algorithms, and practical open-source libraries. We highlight where `tf::tree` aligns with or diverges from these approaches.

Corresponding author: ziga.sajovic@xlab.si

[1]https://github.com/xlabmedical/trueform

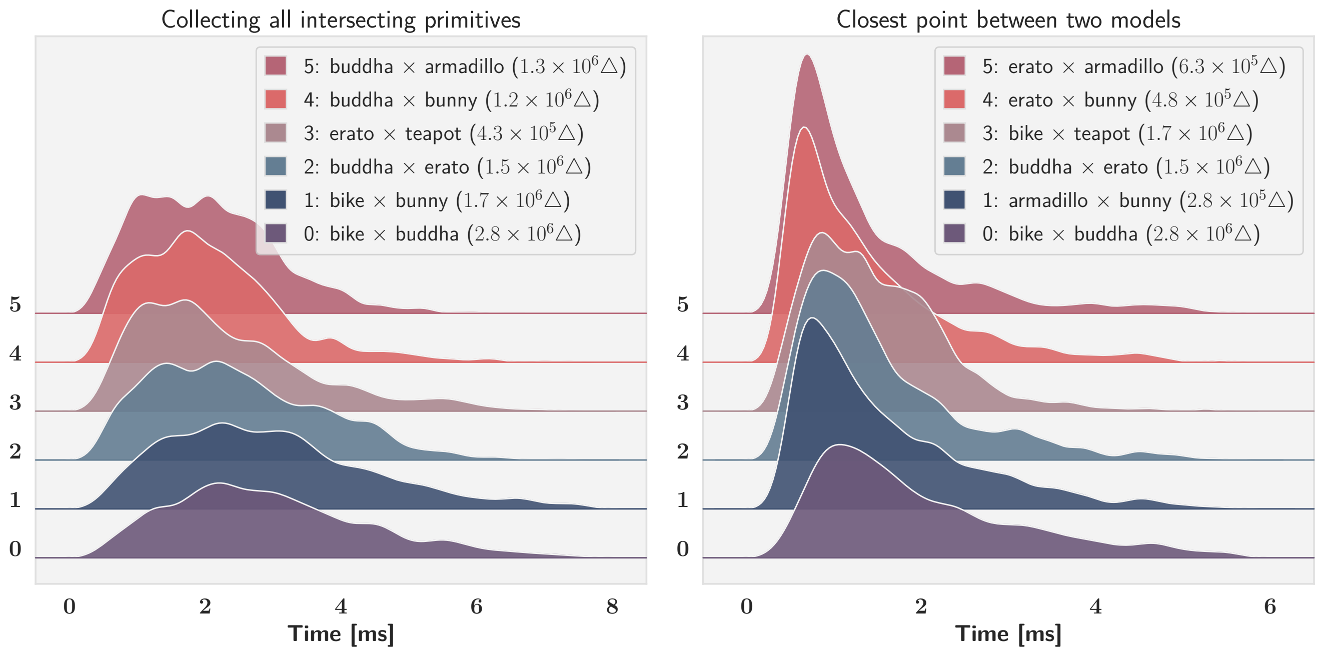

Figure 1. Runtime distributions for queries between two models (from Computer Graphics Archive [1]) using `tf::tree` The total number of triangles per query is marked next to $\triangle$. Each distribution was estimated on 10,000 samples of relative positions. **Left:** Time needed to compute all intersecting primitives between two meshes. Each sample contained an intersecting configuration. **Right:** Time needed to compute a pair of closest points between the two models. Each sample contain a non-intersecting configuration.

A. *Spatial Hierarchies and Partitioning*

Bounding volume hierarchies (BVHs) [2] are a foundational technique in spatial queries, collision detection, and ray tracing. Variants such as AABB trees [3], kD-trees [4], BIHs [5], and R-Trees [6] differ in their partitioning schemes and traversal heuristics, but all aim to minimize overlap and traversal cost. Common BVH construction strategies include top-down recursive partitioning, bottom-up agglomeration, and hybrid methods.

Many BVH builders use heuristics like the Surface Area Heuristic (SAH) [7] to guide node splitting. While SAH produces high-quality trees, its exact evaluation is computationally expensive, and n-ary balanced trees offer a trade-off between construction and traversal efficiency.

SAH is primarily optimized for ray tracing, where the goal is to minimize the expected cost of directional queries such as ray intersections. This can lead to deep, unbalanced trees that favor sparse, single-ray workloads. In contrast, the queries we target—such as collisions, intersections, and proximity searches—are dense, symmetric, and involve pairwise interactions between primitives. These tasks benefit from shallower, balanced n-ary hierarchies that distribute work more uniformly and reduce traversal depth. We therefore focus on building balanced trees rather than optimizing for expected ray traversal cost.

1) Binning Approaches: Binning is widely used to accelerate SAH evaluation by discretizing the spatial domain and approximating split costs across bins [8]. This reduces the complexity of finding optimal partitions and enables SAH-based builders to achieve the optimal $\mathcal{O}(n \log n)$ construction complexity [9]. However, it introduces sensitivity to bin count, spatial distribution, and workload assumptions.

In contrast, `tf::tree` targets a broader class of spatial queries that benefit from shallow, balanced hierarchies. It avoids binning entirely by using in-place partitioning via state-of-the-art selection algorithms, which ensures $\mathcal{O}(n \log n)$ average-case construction. This approach simplifies implementation, improves cache behavior, and supports query workloads like intersection and proximity, where n-ary balanced depth often outperforms heuristic-driven binary splits.

B. *Selection Algorithms*

While partitioning is central to spatial data structures, much of the literature has focused on full sorting [10] or binning heuristics optimized for directional queries. In contrast, the role of linear-time selection algorithms for computing balanced splits has received comparatively little attention—despite their potential benefits for general-purpose acceleration structures.

A long line of research has developed efficient algorithms for finding order statistics in linear or near-linear time, beginning with the Median of Medians algorithm [11], followed by improvements such as Floyd–Rivest selection [12] and more recent high-performance variants like PDQSelect [13], and Median of Ninthers (Alexandrescu's algorithm) [14].

`tf::tree` builds on this foundation by exposing the partitioning strategy as a template parameter, enabling interchangeable use of these algorithms during construction. In our experiments, `std::nth_element` consistently offers the best trade-off between speed, cache behavior, and determinism—making it a practical choice for in-place spatial partitioning in general workloads.

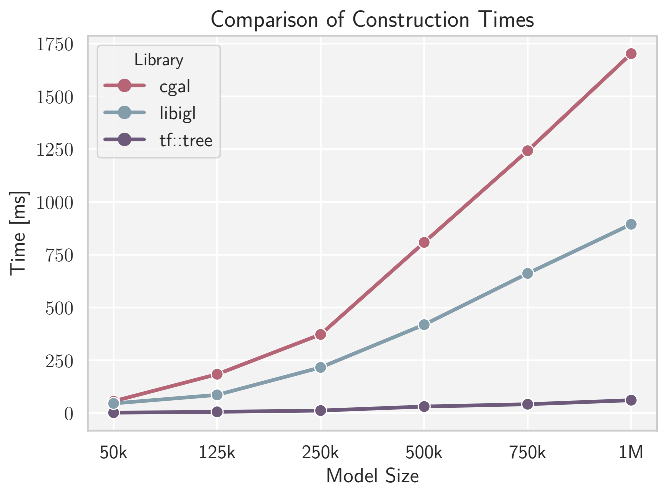

Figure 2. Tree construction times for models from 50k to 1M triangles using `tf::tree`, CGAL, and libigl. `tf::tree` is significantly faster, building a 1M-triangle model in 61 ms versus 894 ms (libigl) and 1.7 s (CGAL), highlighting its suitability for real-time applications.

C. Nearness Queries and Traversal Strategies

Proximity queries, such as nearest-neighbor and k-nearest-neighbor searches, are fundamental operations in spatial data structures. R-trees [15] commonly employ best-first traversal strategies using priority queues to order nodes by their minimum distance to the query point, enabling effective pruning of subtrees. This approach has been further refined in subsequent studies [16], [17]. In contrast, binary spatial partitioning trees like kD-trees often utilize depth-first search (DFS) for proximity queries. DFS explores branches recursively, backtracking when necessary. However, DFS may not prioritize closer nodes, potentially leading to suboptimal performance.

To address these limitations, `tf::tree` introduces a Top-K sorted stack traversal strategy. Our evaluations demonstrate that this approach achieves up to a $3\times$ speedup over traditional priority-queue-based strategies, despite an increase in the number of AABB inspections. These results suggest that, for general geometry queries on balanced, n-ary trees, lightweight traversal strategies like Top-K sorting offer a compelling alternative to classic best-first search heuristic.

D. Open Source Libraries

Open-source libraries offering spatial acceleration structures can be broadly divided into three categories: those optimized for ray tracing, those optimized for collision detection and physics, and those designed for general-purpose geometric processing. `tf::tree` belongs to the third category.

Libraries such as Embree [18], OptiX [19], and nanort [20] are highly optimized for ray tracing and real-time rendering workloads. These systems emphasize ray traversal speed and often rely on SAH-based hierarchies tailored for directional queries. However, their APIs and internal designs make them less suitable for general spatial queries such as collision detection, primitive intersection, or proximity-based reasoning.

A second category, including libraries for real-time physics and robotics like Bullet Physics [21] and FCL [22], prioritizes fast and stable collision detection for simulations. Their algorithms are often tuned with concepts like collision margins, making them unsuitable for fine-grained geometric modeling tasks, such as collecting the complete set of primitive intersections required for mesh booleans.

Finally, libraries like CGAL [23] and libigl [24] provide general-purpose tools for geometric computation. While these libraries support a wide range of operations, including boolean modeling and proximity tests, their acceleration structures are often not optimized for real-time performance. They may also lack the fine-grained control over traversal, construction strategies, and query types that `tf::tree` offers. A comparison of construction times is presented in Figure 2.

III. Methodology

This section presents the core components of `tf::tree`: its data structures, construction algorithm, and query mechanisms.

We begin by describing the tree layout, designed for memory locality and structural simplicity. We then introduce a construction procedure based on in-place partitioning using state-of-the-art selection algorithms Finally, we outline how the tree supports collision, (self) intersection, and nearest-neighbor queries using stack-based traversals that outperform traditional priority queues in practice.

A. Tree Data Structure

For performance reasons, we must consider about how the processor accesses memory [25]. That means paying attention to temporal and spatial locality. In practice, this leads to two important design rules: first, all our data should be stored in contiguous arrays; second, elements that are accessed together—like the children of a node—should also be laid out next to each other in memory. These simple but strict constraints shape the structure of our tree and how we represent its nodes, primitives, and their relationships.

We now define the core data structures underlying our tree representation.

Definition 1 (ID range)**.** For a collection of primitives P_i, indexed by i, an ID range is an array I containing a permutation of the sequence $[0 : N)$, where N is the number of primitives. An ID sub-range $I[i : j]$ is a sub-array of I, containing elements $[I_i, I_{i+1}, \ldots, I_{j-1}]$. The sub-range $I[i : j]$ may also be denoted by $(i, j - i) \subseteq I$.

Definition 2 (Node)**.** A node N_i is part of an array of nodes N, indexed by i. It consists of an axis-aligned bounding box (*aabb*) and associated metadata.

A node is classified as an *inner node* if its children are other nodes N_c, and as a *leaf node* if its children are primitives P_k.

The *aabb* of N_i encloses all primitives in its subtree. The metadata includes:

- an *axis*, indicating the dimension along which the node was partitioned (non-negative for inner nodes, negative for leaves), and
- a *data* range, given by an offset o and size s, indexing into its children.

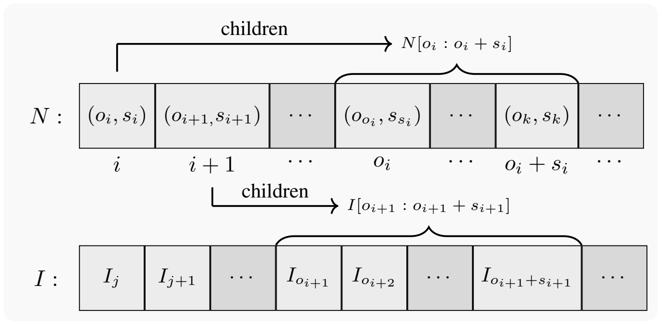

Figure 3. The node (N) and ID (I) arrays of tf::tree. The *data* array (o, s) specifies subranges within these arrays. The node N_i is an inner node with children $N[o_i : o_i + s_i]$, while N_{i+1} is a leaf node with children $I[o_{i+1} : o_{i+1} + s_{i+1}]$. All children are contiguous.

For leaf nodes, this range refers to an ID sub-range (o, s). For inner nodes, it refers to a node sub-range $N[o : o + s]$ within the array N.

Thus, all child nodes N_c of node N_i must appear consecutively in N; i.e., $N[o : o + s]$.

When interacting with the tree, all children of a node are always accessed together—either to decide whether to descend into them (in the case of an inner node) or to process the corresponding primitives P_i (for a leaf node). This access pattern reinforces our data layout strategy, which adheres closely to locality requirements. Fig. 3 illustrates how nodes and ID ranges are stored and referenced in memory. These two arrays—one representing the spatial hierarchy, the other indexing primitives—form the foundation of our data structure.

Definition 3 (tf::tree)**.** A `tf::tree` is defined as the pair $\langle I, N \rangle$, where I is an ID array (Def. 1) and N is an array of nodes (Def. 2). In a balanced tree, we additionally specify the number of children of inner-nodes $k \in \mathbb{N}$ and the maximum number of children in an inner node $l \in \mathbb{N}$.

Together, these arrays encode both the structure and contents of the tree, enabling efficient construction, traversal and query.

B. Tree Construction

The construction of spatial partitioning trees generally follows a common pattern: the current node is divided into k partitions, and recursion proceeds independently into each of them as they become child nodes. Because these child nodes are independent, the recursion can be parallelized. Thus, the computational bottleneck shifts to the partitioning step – our primary area of contribution with tree construction.

1) Partitioning: Partitioning, in practical terms, means reordering the ID range $I[n_0 : n_k]$ (see Def. 1) associated with a node's primitives, and dividing it into k non-overlapping sub-ranges:

$$\{I[n_0 : n_1],\ I[n_1 : n_2],\ \ldots,\ I[n_{k-1} : n_k]\}$$

Each of these sub-ranges corresponds to one child of an inner node (Def. 3), and together they fully cover $I[n_0 : n_k]$.

When a balanced tree is desired, one often turns to techniques in the spirit of Sort-Tile-Recursive (STR) trees [10], where primitives are sorted by their AABB centroids before the range is evenly partitioned.

Algorithm 1: In-place tree construction with nth_element-based partitioning of ID range.

```
Input: AABBs of primitives: $\mathcal{B}$,  ID sub-range: $I'$,
       node index: $n$,  offset of $I$ in the original
       array: $o$,  configuration: $c$
Output: Updated node array $N$,  partitioned ID array
        $I$
1  Compute bounding box $A_n = \bigcup_{i \in I'} \mathcal{B}[i]$;
2  if $|I| \leq c$.leaf_size then
3      Mark $N[n]$ as a leaf with children $(o, |I'|) \subseteq I$;
4      return
5  end
6  Set split axis $a \leftarrow \arg\max \mathrm{diag}(A_n)$;
7  Set $N[n]$.axis $\leftarrow a$;
8  Initialize $work\_ids \leftarrow I'$,
     $p \leftarrow$ balanced_split($c$.inner_size),
     $child \leftarrow$ first_child_index($n$);
9  while !$work\_ids$.empty() do
10     if $|work\_ids| \geq p$ then
11         Apply F::partition (with $n = p$) to
             $work\_ids$ along axis $a$;
12     end
13     $next\_ids \leftarrow$ take($work\_ids, p$);
14     Dispatch on node $N[child]$ with $next\_ids$, offset
         $o + (|I| - |work\_ids|)$;
15     $work\_ids \leftarrow$ drop($work\_ids, |next\_ids|$);
16     $child \leftarrow child + 1$;
17 end
18 Mark $N[n]$ as an inner node with children
19 (first_child_index($n$), $c$.inner_size) $\subseteq N$;
```

To improve upon this and achieve $\mathcal{O}(n \log n)$ construction time (on average), we leverage the insight that elements $I_i^{n_a}$ and $I_j^{n_b}$ from different sub-ranges must be separated, but elements within a sub-range need not be sorted. The same insight is commonly used in binning-based partitioning approaches [2], though they do not guarantee balanced splits.

The requirement, then, becomes: given a position n in the range I, reorder the elements such that $\forall_{j<n} I[j] \leq I[n]$ and $\forall_{j>n} I[j] \geq I[n]$. That is, $I[n]$ is the partitioning nth element.

`F::partition`: is a function in the that satisfies this requirement with average-case complexity $\mathcal{O}(n)$. Our default is `nth_element` from the C++ standard library [26]. It is typically implemented using a combination of `Quickselect` [27] and `Introselect` [28]. A major advantage of using this general-purpose algorithm is that it benefits from continued optimization and tuning [29]–[31].

In practice, we use `F::partition` to partition the ID range I into k disjoint sub-ranges that satisfy the partitioning requirement. This is achieved through $k - 1$ applications of the function, using the center of the primitives' AABBs along the axis of the node being partitioned with the largest spatial extent. We compare the usage of state-of-the-art selection algorithms at this task in Section IV-A2.

2) Implementation: The construction procedure is detailed in Algorithm 1. Before invoking the algorithm, we initialize the ID array I with the sequence $[0 : P]$, where P is the number of primitives. The node array N is then allocated to accommodate the structure of a perfectly balanced `k`-ary tree. The index of the first child of node n is computed as

$$\text{first\_child\_index}(n) = k \cdot n + 1,$$

where k is the fixed number of children for each inner node. Hence, the nodes are laid out in memory in such a manner, that the children of a node are contiguous. Furthermore, all nodes on a specific level of the tree are contiguous. Since child nodes are independent, each recursive dispatch can be performed in parallel, using `tbb` [32].

Theorem 1. Algorithm 1 runs in $\mathcal{O}(n \log n)$ time on average, and $\mathcal{O}(n \log^2 n)$ in the worst case.

Proof. This follows from the Master Theorem [33], based on the complexity guarantees of `F::partition`. □

API.: Tree construction is performed via `tf::tree::build` method that accepts three parameters: a range of input objects, an AABB generator, and a configuration object. The range can be any object satisfying the C++range concept. The second parameter, `make_aabb`, is a callable that receives an object from the range and returns its `tf::aabb`. The third parameter is a `tf::tree_config` struct specifying inner-size and leaf-size. Optionally it can be passed an additional first parameter, i.e. `build(F,...)`, where `F` specifies how the partitioning step is performed via a static `F::partition` method. `F` defaults to using `nth_element`. We provide implementations of various state-of-the-art algorithms [34] (see Section IV-A2) for comparison in the namespace `tf::strategy`.

C. Search Queries

Search queries operate by traversing the tree and descending into nodes whose AABBs contain or intersect the query volume. Queries may originate from a single primitive or from another spatial tree or itself.

1) Query by Primitive: This is the simplest case. Large regions of space can be quickly discarded by checking AABBs. Our implementation avoids recursion and instead uses a static stack (via `small_vector` [35] for safety) to improve both performance and cache locality.

Such queries are useful for a variety of tasks, including collecting all primitives that satisfy a predicate or performing early-exit collision checks at the primitive level.

API.: A query by primitive is performed using the function `tf::search`, which accepts three parameters: the `tf::tree` to be queried, a callable `check_aabb`: $\text{tf::aabb} \rightarrow$ `bool` that filters internal nodes based on their AABB, and `apply_f`: id $\in I \rightarrow$ `bool`, which is invoked on the primitive id in the leaf nodes. If `apply_f` returns `true`, the traversal terminates early. For example, in a collection query, `apply_f` would return `false` after appending the intersecting results. We additionally supply `tf::search_broad`, where `apply_f`: $I[n_i : n_j] \rightarrow$ `bool` is invoked on the sub-ID range within leaf nodes.

Algorithm 2: `Search`(N_0, N_1, `fs`)

```
Input: Node $N_0$, Node $N_1$, Functions fs
1  if fs.abort_f() or !fs.check_aabbs($N_0$.aabb, $N_1$.aabb) then
2      return;
3  end
4  if $N_0$ and $N_1$ are both leaves then
5      fs.apply_f($N_0$.ids($I$), $N_1$.ids($I$));
6  end
7  else if $N_0$ is a leaf then
8      foreach child $N_i$ of $N_1$ do
9          Dispatch Search($N_0, N_i$, fs);
10     end
11 end
12 else if $N_1$ is a leaf then
13     foreach child $N_i$ of $N_0$ do
14         Dispatch Search($N_i, N_1$, fs);
15     end
16 end
17 else
18     foreach child $N_i$ of $N_0$ and $N_j$ of $N_1$ do
19         Dispatch Search($N_i, N_j$, fs);
20     end
21 end
```

2) Query by Tree: In this case, the goal is to find all overlapping primitive pairs between two trees. The traversal resembles a nested search: for each pair of overlapping nodes (N_0, N_1), we recurse into their children until both are leaf nodes. At that point, we process the candidate primitive pairs. All child dispatches can be executed in parallel. This type of query follows the recursive structure shown in Algorithm 2.

API.: A query by tree is performed using the function `tf::search`, which accepts four parameters: a pair of `tf::trees`, a callable `check_aabb`: $(\text{tf::aabb}, \text{tf::aabb}) \rightarrow$ `bool` that filters internal nodes based on their AABBs, and `apply_f`: $(I_0[n_i : n_j], I_1[m_k : m_l]) \rightarrow$ `bool`, which is invoked on the pair of sub-ID ranges within leaf nodes. If `apply_f` returns `true`, the traversal terminates early. For example, in a collection query, `apply_f` would return `false` after appending the intersecting results. We additionally supply `tf::search_broad`, where `apply_f`: $(I_0[n_i : n_j], I_1[m_k : m_l]) \rightarrow$ `bool` is invoked on the sub-ID range within leaf nodes.

Note that `apply_f` must be thread-safe if it performs any shared-state mutation, such as populating a global list of collisions.

3) Query by Self: `tf::tree` supports queries by self via the `tf::search_self` and `tf::search_self_broad` functions. Their API mirrors that of Query by Tree. This feature is useful for detecting self-intersections in meshes and merging primitives in ε-proximity.

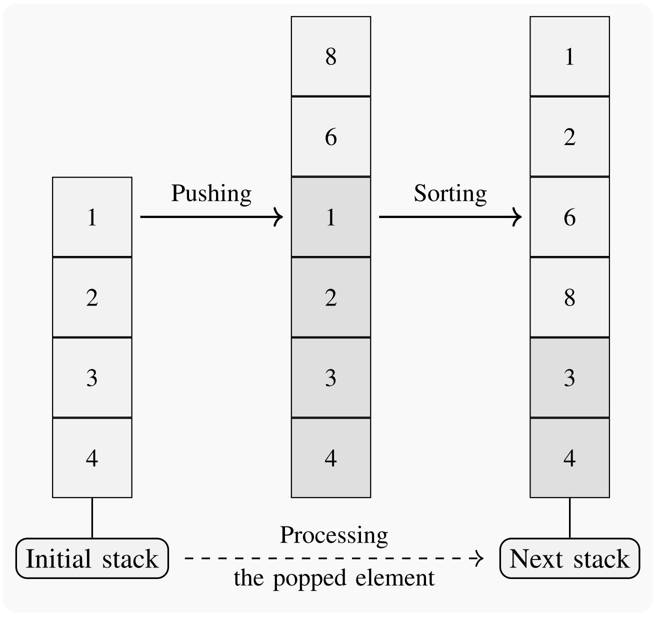

Figure 4. Visualization of a Top-K Sorted Stack for a binary tree. On the left is the initial state of the stack after a pop operation. During processing, elements 6 and 8 are added. The top 4 elements (2 newly added and 2 existing) are then sorted, resulting in the stack for the next iteration.

D. *Nearness Queries*

Nearness queries—whether between a tree and a primitive, or between two trees—typically follow a standard strategy: nodes (or node pairs) are inserted into a priority queue, ordered by their distance to the query primitive, or by the distance between their AABBs in the tree-to-tree case. This approach allows early pruning of branches using heuristics such as the minimal distance and the maximum-of-minimal distances [15], [16].

The use of a priority queue guarantees that the closest nodes are explored first, minimizing the number of checks required. However, research has shown that on modern processors, reducing the number of operations does not always translate into faster execution time [14]. For instance, linear search outperforms binary search on small sorted arrays due to better cache locality and branch prediction.

1) Top-K Sorted Stack: Our implementation follows the standard strategy, but replaces the priority queue with a stack. To preserve ordering where it matters most, we sort the top k elements of the stack after processing the popped element—as this processing may have pushed several new elements onto the stack.

Here, k is defined as the number of elements pushed onto the stack while processing the popped element, plus the number of nodes in the current inner-leaf. See Figure 4 for a demonstration.

The Top-K Sorted Stack imposes sufficient ordering on the traversal of the search space that, even though the nearness query performs more operations than when using a priority queue, it achieves better overall performance in practice. These results are evaluated in Section IV-C1.

2) Implementation: We provide implementations for both query types: between a tree and a primitive, and between two trees. The function `tf::nearness_search` supports an optional first parameter that selects the traversal strategy: `top_k_sorted` or `priority_queue`, both defined in the `tf::strategy` namespace. The remaining parameters follow the APIs described below.

API: Primitive to Tree: This variant accepts three arguments: the `tf::tree` being queried, a callable `aabb_metric_f`: $\mathrm{tf} :: \mathrm{aabb} \to \mathbb{R}$, which computes the distance to an AABB, and a callable `closest_point_f`: $i \in I \to \mathrm{tf} :: \mathrm{closest\_point}$, which maps a primitive ID to a closest point result.

API: Tree to Tree: This variant accepts four arguments: a pair of `tf::tree` instances to be queried, a callable `aabb_metric_f`: $(\mathrm{tf} :: \mathrm{aabb}, \mathrm{tf} :: \mathrm{aabb}) \to \mathrm{tf} :: \mathrm{aabb\_metrics}$, which returns the minimum and min–max distances between bounding volumes [16], and a callable `closest_point_f`: $(i_0 \in I_0, i_1 \in I_1) \to \mathrm{tf} :: \mathrm{closest\_point\_pair}$, which maps a pair of primitive IDs to a closest point pair result.

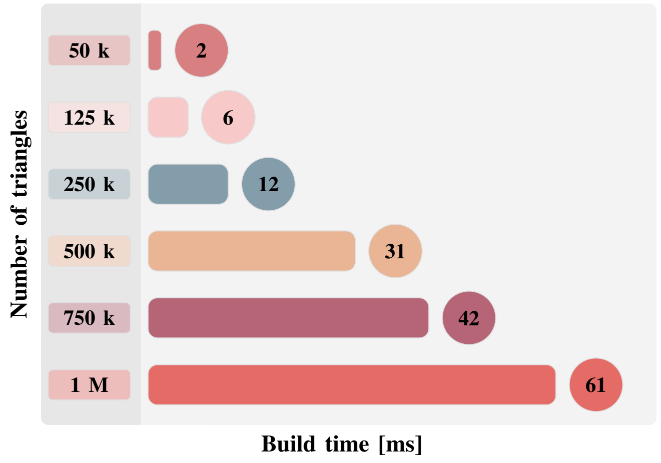

Figure 5. `tf::tree` construction times for models of various sizes, displaying near-linear scalability in number of primitives. We observe a construction rate of approximately $1.6 \cdot 10^7 \, \frac{\text{triangles}}{\text{second}}$.

E. *Dynamics of `tf::tree`*

All query APIs operate on user-provided callables, allowing dynamic behavior to be expressed at the query level. Tree dynamics can thus be achieved by applying transformations within the supplied callables.

We provide the utility function `tf::transformed`, which modifies the tree's AABBs to reflect a given transformation while preserving structural consistency. Likewise, users may apply transformations to primitives within the relevant callables, enabling queries over geometry undergoing linear motion or deformation, without rebuilding the tree.

IV. Evaluations

We evaluate the performance of our approach across three core tasks: tree construction using state-of-the-art selection algorithms, search queries for collision and (self) intersection detection, and nearness queries. For the latter, we compare the use of a top-k sorted stack against a standard priority queue.

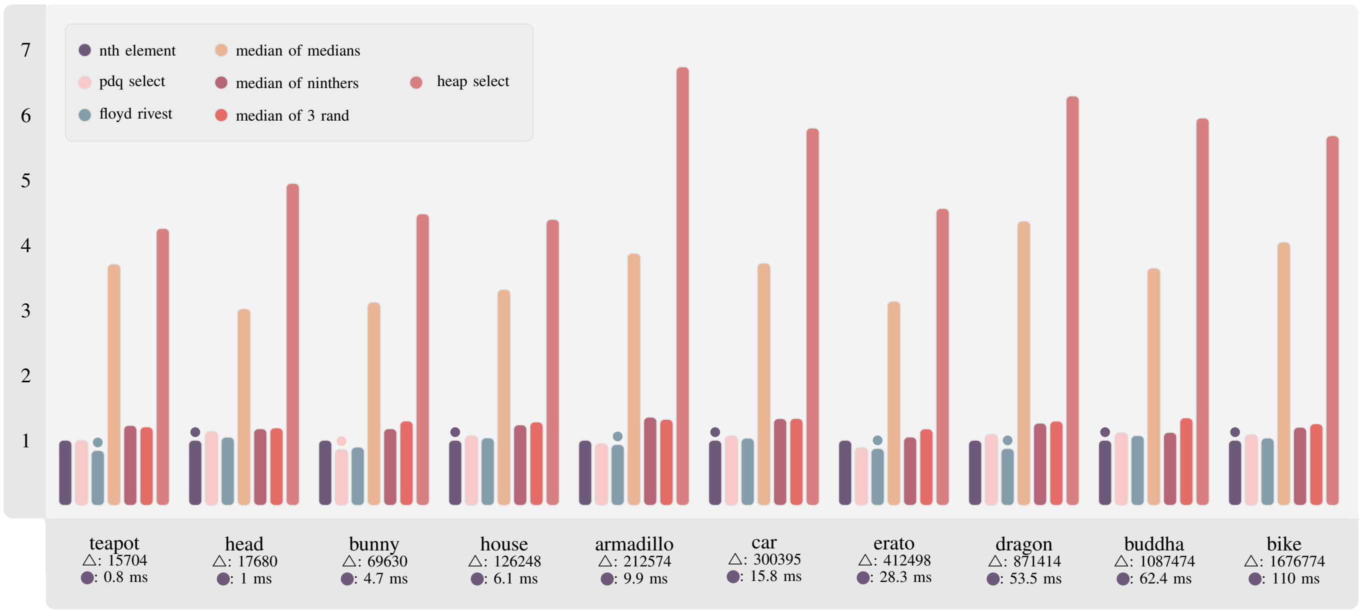

Figure 6. Build times of `tf::tree` using different selection algorithms. The chart shows times normalized relative to the baseline `nth_element`. A small circle over the bar marks the fastest algorithm of the group. Each group corresponds to a specific model, with the number of triangles indicated next to △ and the baseline time shown alongside ●.

All evaluations were conducted using our open-source implementation [36], compiled with `clang18` using maximum optimization settings. Benchmarks were performed on an Intel i7-9750H CPU with 12 threads.

The trees we evaluate on were configured using `tf::tree_config`$(4, 4)$, meaning a 4-ary tree, unless specified otherwise.

The models used for these evaluations are available at Computer Graphics Archive [1] and The Stanford 3D Scanning Repository [37].

A. *Tree Construction*

We evaluate construction times across models of varying sizes, ranging from 15k to 1.6M triangles [36]. These baseline measurements are shown in Figure 6. On the tested hardware, we observe a construction rate of approximately $1.6 \cdot 10^7 \, \frac{\text{triangles}}{\text{second}}$.

1) Scalability: To assess scalability with respect to mesh size, we remeshed the Stanford Dragon model into six variants containing $[50\text{k}, 125\text{k}, 250\text{k}, 500\text{k}, 750\text{k}, 1\text{M}]$ triangles. The results, presented in Figure 5, demonstrate near-linear scaling behavior.

2) Partitioning: We evaluate the impact of various state-of-the-art selection algorithms on the partitioning step of Algorithm 1. The algorithms considered include:

- PDQSelect (based on PDQSort) [13],
- Floyd–Rivest [12],
- Median of Medians [11],
- Median of Ninthers (also known as Alexandrescu's algorithm [14]),
- Median of Three (Quickselect with the median of three random elements),
- HeapSelect.

The comparative performance of Algorithm 1 with these selection strategies is shown in Figure 6, where all times are normalized to our baseline `nth_element`.

While performance varies across models, the three best-performing algorithms are consistently `nth_element`, `pdq_select`, and `floyd_rivest_select`. Among these, `nth_element` most often yields the fastest construction times.

B. *Search Queries*

We evaluate search queries by applying `tf::tree` to the tasks of collision detection and computing all pairs of (self) intersecting primitives enclosed by two spatial trees. Prior to evaluation, each pair of models was uniformly rescaled to fit within a common bounding sphere centered at the origin.

1) Collision Detection: To assess collision detection performance, we sampled 10,000 relative configurations between each model pair and checked for collision in each case. Results show that the average time for detecting a collision between two models is 47 ns. These values are reported in Table I under the **collide** sub-heading.

In practice, we employ `tf::tree` for collision detection in automatic model positioning pipelines, where models are continuously checked for contact during iterative placement.

2) Intersecting Primitive Pairs: To evaluate the performance of `tf::tree` in computing intersecting primitive pairs, we sampled 10,000 relative configurations resulting in intersection between each model pair. For each sample, we computed all intersecting pairs of primitives. The average evaluation times are reported in Table I under the **intersect** sub-heading.

Our results show that `tf::tree` supports real-time computation of intersecting primitive pairs (on the order of milliseconds), even for large models. This capability forms the

Table I
COMPREHENSIVE PERFORMANCE EVALUATION OF `TF::TREE`. GIVEN TWO INPUT MODELS, WE SAMPLE 10,000 CONFIGURATIONS FROM THE SPACE OF ALL RELATIVE PLACEMENTS. THE TABLE REPORTS AVERAGE RESULTS ACROSS TWO CATEGORIES: **SEARCH** (INTERSECTING CONFIGURATIONS) AND **NEARNESS** (NON-INTERSECTING CONFIGURATIONS). **SEARCH:** THE COLUMN **COLLIDE** REPORTS THE TIME REQUIRED TO DETECT WHETHER THE TWO MODELS INTERSECT, WHILE **INTERSECT** GIVES THE TIME NEEDED TO COLLECT ALL INTERSECTING PRIMITIVE PAIRS. BOTH QUERIES EXECUTE IN REAL TIME, EVEN FOR MODELS WITH A LARGE NUMBER OF PRIMITIVES. **NEARNESS:** THE COLUMN **TOP-K** SHOWS THE TIME NEEDED TO FIND THE CLOSEST POINT BETWEEN MODELS USING A TOP-k SORTED STACK; **HEAP** REPORTS THE SAME USING A PRIORITY QUEUE. $\mathbf{f}_{time}$ INDICATES THE SPEED-UP FACTOR ACHIEVED WITH THE SORTED STACK, WHILE $\mathbf{f}^{cost}$ SHOWS THE CORRESPONDING INCREASE IN AABB INSPECTIONS. DESPITE PERFORMING MORE INSPECTIONS, THE TOP-k STRATEGY YIELDS SIGNIFICANTLY BETTER PERFORMANCE IN PRACTICE.

Input			**Search**		**Nearness**			
model$_0$	**model**$_1$	**primitives**	**collide** [ns]	**intersect** [ms]	**top-k** [ms]	**heap** [ms]	$\mathbf{f}_{time}$	$\mathbf{f}^{cost}$
armadillo	bunny	282204	33.20	0.76	1.49	3.00	2.02	1.26
erato	teapot	428202	39.15	2.14	2.55	6.16	2.42	1.47
erato	bunny	482128	35.42	1.42	1.40	2.60	1.86	1.40
erato	armadillo	625072	50.99	1.46	1.36	2.35	1.72	1.51
buddha	teapot	1103178	29.76	3.71	5.43	16.40	3.02	1.10
buddha	bunny	1157104	38.17	2.01	2.90	6.07	2.09	1.47
buddha	armadillo	1300048	52.51	1.99	2.79	5.21	1.87	1.52
buddha	erato	1499972	46.50	2.41	1.47	2.50	1.70	1.40
bike	teapot	1692478	25.89	8.65	1.40	3.46	2.47	1.13
bike	bunny	1746404	48.86	2.93	1.21	2.37	1.96	1.47
bike	armadillo	1889348	63.46	3.15	1.21	2.17	1.79	1.59
bike	erato	2089272	66.89	3.34	0.78	1.51	1.92	1.25
bike	buddha	2764248	61.07	2.94	2.30	3.87	1.68	1.58

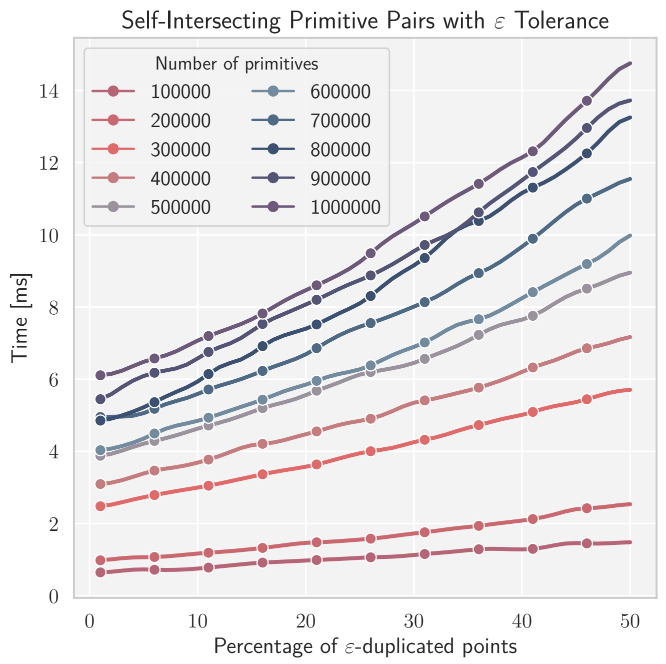

Figure 7. Performance of `tf::tree` on detecting ε-self-intersections. Each curve corresponds to a point cloud of fixed size, with the x-axis indicating the percentage of points duplicated and displaced by ε. The results show that `tf::tree` maintains interactive self-query times even under growing redundancy and scale.

foundation of our real-time mesh boolean pipeline, where fast and reliable detection of intersections is the first step [38].

3) Self-Intersections with ε Tolerance: To evaluate `tf::tree` on tolerant self-intersection detection, we generated synthetic point clouds of varying sizes. For each size, a percentage p of points was duplicated and offset by ε in a random direction—simulating geometric redundancy common in real-world data.

The goal is to find all primitive pairs within ε proximity, i.e., all ε-self-intersections. These pairs guide merging or deduplication, reducing the number of distinct points by roughly $p\%$.

As shown in Figure 7, `tf::tree` sustains interactive self-query times even at large scales and high densities.

C. Nearness Queries

We evaluate the ability of `tf::tree` to efficiently compute the closest point between two models across a wide range of configurations. Given a pair of models, we first normalize them to fit within a common bounding sphere. We then sample 10,000 relative positions that result in an intersection, and for each configuration, we compute the closest point using two different strategies: a top-k sorted stack and a standard priority queue. The priority queue was implemented using a flat heap, employing the `std::make_heap`, `std::push_heap`, and `std::pop_heap` functions of the standard C++library.

The results, averaged across all configurations, are reported in Table I under the **Nearness** heading.

1) Top-K Stack: We evaluate the effectiveness of the top-k sorted stack strategy for nearness queries, which we configured as specified in Section III-D1. Specifically, we compare its runtime performance and the computational cost in terms of AABB inspections against the baseline heap-based approach. The results highlight the trade-off between increased traversal effort and practical speed-up.

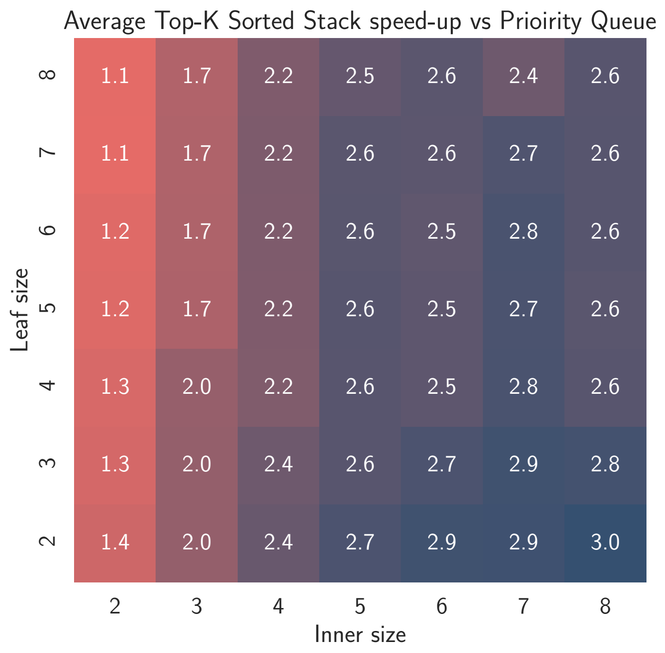

Figure 8. Heatmap comparing the performance of Top-K Sorted Stack traversal versus Priority Queue for nearness queries across various tree configurations (varying tree arity and leaf size). The values represent the relative speed-up factor. Results indicate that the performance benefit of Top-K Sorted Stack increases with higher tree arity.

AABB Inspections: When using the top-k sorted stack, `tf::tree` inspects more AABBs during the search process compared to the heap-based approach. This is expected: the partially sorted stack imparts approximate ordering on the traverasal and does not prune optimally. The increased number of inspections is quantified by the column $\mathbf{f}^{cost}$, which reports the ratio of AABB inspections between the top-k sorted stack and the heap strategies.

Results show that the top-k sorted stack performs $1.37\times$ AABB inspections as the priority queue based strategy. Despite this increase, the sorted stack maintains a highly efficient query structure by reducing traversal overhead and taking advantage of cache-locality and simpler branching behavior.

Peformance: In practice, the top-k sorted stack consistently outperforms the heap-based alternative in terms of runtime. As shown in the $\mathbf{f}_{time}$ column, it yields a speed-up factor ranging from $1.7\times$ to over $3\times$ on complex models. Furthermore, the performance benefit increases with higher tree arity, as demonstrated in Figure 8.

While the heap strategy optimizes for minimal node visits, the stack-based approach emphasizes speed through structural simplicity and execution efficiency. These results demonstrate that even with more geometric operations, the top-k method provides superior overall performance in practical applications.

V. Conclusions

In this section, we summarize the key contributions of our work, acknowledge limitations, and outline possible directions for future development. We conclude with a brief reflection on the broader impact of our approach.

A. Summary

We introduced `tf::tree`, a spatial acceleration structure built for performance, simplicity, and flexibility in geometric computing. By combining data-oriented layouts, in-place partitioning with state-of-the-art selection algorithms (ensuring $\mathcal{O}(n\log(n))$ average construction), and stack-based traversal schemes, `tf::tree` achieves high efficiency across construction, collision detection, (self) intersection and nearness queries.

Although `tf::tree` is structurally static, its query APIs support geometry undergoing linear transformations through user-defined callables. This enables dynamic queries over moving or deforming models without rebuilding the tree.

Empirical evaluations show that `tf::tree` achieves near-linear scalability in construction and delivers real-time query performance on models with millions of primitives. Notably, in nearness queries, our Top-K sorted stack traversal provides a practical speedup of up to $3\times$ over traditional heap-based methods by trading an increase in AABB inspections for superior cache performance and execution efficiency, demonstrating the effectiveness of hardware-aware algorithmic design.

B. Limitations and Future Work

While `tf::tree` supports dynamic queries through transformed callables, its internal structure remains fixed after construction. Extending the system to support incremental updates or fully dynamic scenes—particularly those involving free-form deformation—remains a compelling direction for future work. Although we employ such capabilities in practice, the procedure has not yet been formalized or exposed through the public API.

At present, search queries such as collision detection and intersection are parallelized, whereas nearness queries remain sequential. Parallelizing nearness queries presents an opportunity to further improve performance, especially on high-core-count systems. This is a natural extension of our current design and will be addressed in future work.

C. Final Remarks

`tf::tree` offers a practical and performant foundation for spatial queries in geometry processing. Its modular, open-source design is intended to be easily adapted and extended. We hope it proves valuable both as a production tool and as a reference for building fast and clean spatial structures.

References

[1] M. McGuire, "Computer graphics archive," July 2017. https://casual-effects.com/data.

[2] D. Meister and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," *Computer Graphics Forum*, vol. 38, no. 1, pp. 27–51, 2019.

[3] G. J. van den Bergen, "Efficient collision detection of complex deformable models using aabb trees," *Journal of Graphics Tools*, vol. 2, no. 4, pp. 1–14, 1997.

[4] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[5] C. Wächter and A. Keller, "Instant ray tracing: The bounding interval hierarchy," in *Rendering Techniques*, pp. 139–149, Eurographics Association, 2006.

[6] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pp. 322–331, 1990.

[7] J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision," *The Visual Computer*, vol. 6, no. 6, pp. 153–166, 1990.

[8] I. Wald, "On fast construction of sah-based bounding volume hierarchies," in *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pp. 33–40, IEEE, 2007.

[9] I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in o(n log n)," in *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pp. 61–69, IEEE, 2006.

[10] S. T. Leutenegger, M. A. Lopez, and J. Edgington, "Str: A simple and efficient algorithm for r-tree packing," in *Proceedings 13th international conference on data engineering*, pp. 497–506, IEEE, 1997.

[11] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan, "Time bounds for selection," *Journal of Computer and System Sciences*, vol. 7, no. 4, pp. 448–461, 1973.

[12] R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," *Communications of the ACM*, vol. 18, no. 3, pp. 165–172, 1975.

[13] O. R. L. Peters, "Pattern-defeating quicksort," 2021.

[14] A. Alexandrescu, "Sorting algorithms: Speed is found in the minds of people," in *Proceedings of the CppCon Conference*, CppCon, 2019. Conference talk.

[15] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," *SIGMOD Rec.*, vol. 24, p. 71–79, May 1995.

[16] K. L. Cheung and A. W.-C. Fu, "Enhanced nearest neighbour search on the r-tree," *SIGMOD Rec.*, vol. 27, p. 16–21, Sept. 1998.

[17] L. Arge, M. D. Berg, H. Haverkort, and K. Yi, "The priority r-tree: A practically efficient and worst-case optimal r-tree," *ACM Trans. Algorithms*, vol. 4, Mar. 2008.

[18] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst, "Embree: A kernel framework for efficient cpu ray tracing," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 143, 2014.

[19] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: A general purpose ray tracing engine," *ACM Transactions on Graphics (TOG)*, vol. 29, no. 4, p. 66, 2010.

[20] S. Takahashi, "Nanort: Single header only modern ray tracing kernel." https://github.com/lighttransport/nanort, 2015. Accessed: 2025-05-23.

[21] E. Coumans *et al.*, "Bullet physics sdk." https://pybullet.org/, 2020.

[22] J. Pan, S. Chitta, and D. Manocha, "FCL: A general purpose library for collision and proximity queries," in *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 3859–3866, 2012.

[23] The CGAL Project, *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 ed., 2024.

[24] A. Jacobson and D. Panozzo, "libigl: prototyping geometry processing research in c++," in *SIGGRAPH Asia 2017 Courses*, SA '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[25] Y. Sharvit, *Data-oriented Programming: Reduce Software Complexity*. Simon and Schuster, 2022.

[26] ISO/IEC, "Iso/iec 14882:2020 — programming languages — c++," 2020. See §25.4.3 [alg.nth.element] for `std::nth_element`.

[27] H. M. Mahmoud, R. Modarres, and R. T. Smythe, "Analysis of quickselect: An algorithm for order statistics," *Informatique Théorique et Applications*, vol. 29, no. 4, pp. 255–276, 1995.

[28] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.

[29] M. Kuba, "On quickselect, partial sorting and multiple quickselect," *Information processing letters*, vol. 99, no. 5, pp. 181–186, 2006.

[30] D. R. Musser, "Introspective sorting and selection algorithms," *Software: Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.

[31] A. Alexandrescu, "Fast deterministic selection," *arXiv preprint arXiv:1606.00484*, 2016.

[32] Intel Corporation, *Intel® Threading Building Blocks (TBB)*. Intel Corporation, 2020.

[33] J. L. Bentley, D. Haken, and J. B. Saxe, "A general method for solving divide-and-conquer recurrences," *ACM SIGACT News*, vol. 12, no. 3, pp. 36–44, 1980.

[34] D. Kutenin, "miniselect: Generic selection and partial ordering algorithms." https://github.com/danlark1/miniselect, 2024. Accessed: 2025-05-16.

[35] LLVM Project, "llvm::smallvector — llvm 18.1.3 documentation." https://llvm.org/doxygen/classllvm_1_1SmallVector.html, 2024. Accessed: 2025-05-18.

[36] Žiga Sajovic, "true_form: High-performance geometric modeling." https://github.com/xlabmedical/true_form, 2024. Accessed: 2025-05-16.

[37] Stanford Computer Graphics Laboratory, "The stanford 3d scanning repository." https://graphics.stanford.edu/data/3Dscanrep/, 1994. Accessed: 2025-05-22.

[38] Žiga Sajovic and D. Knez, "trueform: Real-time mesh booleans that commute with mesh idealization." TechRxiv, May 2025. Preprint.