2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

TřiVis: Versatile, Reliable, and High-Performance Tool for Computing Visibility in Polygonal Environments

Jan Mikula[1,2], Miroslav Kulich[1], and Libor Přeučil[1]

***Abstract*— Visibility is a fundamental concept in computational geometry, with numerous applications in surveillance, robotics, and games. This software paper presents TřiVis, a C++ library developed by the authors for computing numerous visibility-related queries in highly complex polygonal environments. Adapting the triangular expansion algorithm, TřiVis stands out as a versatile, high-performance, more reliable and easy-to-use alternative to current solutions that is also free of heavy dependencies. Through evaluation on a challenging dataset, TřiVis has been benchmarked against existing visibility libraries. The results demonstrate that TřiVis outperforms the competing solutions by at least an order of magnitude in query times, while exhibiting more reliable runtime behavior. TřiVis is freely available for private, research, and institutional use at https://github.com/janmikulacz/trivis.**

I. Introduction

This software paper presents TřiVis, a C++ library developed by the authors for computing visibility in 2D polygonal environments. The authors, focusing primarily on visibility-based route planning solutions for autonomous mobile robots, have encountered severe limitations in the currently available visibility implementations, including the 2D Visibility package in the widely used Computational Geometry Algorithms Library (CGAL). Motivated by the need for a fast, robust, versatile, and lightweight solution, the authors developed TřiVis to address these limitations and provide a high-performance, reliable, and easy-to-use solution for the robotics community and beyond.

The library is suitable for various applications that involve 2D polygonal environments where frequent visibility queries are essential. These applications include the design of security and surveillance systems [1] and robotics. In the field of robotics, specific examples include route planning for efficient environment inspection or search [2], [3], multi-agent systems [4], [5], and pursuit-evasion games [6], [7].

This paper is organized as follows: Sec. II defines the fundamental concepts of visibility and related terms. Sec. III provides a literature background on visibility algorithms and describes the primary algorithm used in TřiVis: the triangular expansion algorithm [8], [9]. This section also details the TřiVis-specific adaptations of the algorithm for handling other visibility queries. Sec. IV details the design of TřiVis, including the library's robustness strategies, internal dependencies, and a short snippet of the library's usage. Sec. V evaluates the performance of TřiVis on a dataset of complex polygonal environments and compares it with other visibility libraries. Finally, Sec. VI concludes the paper.

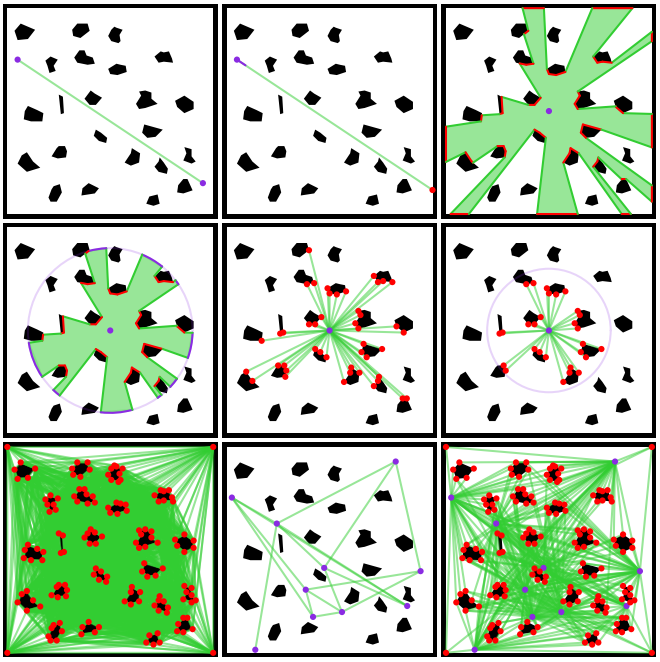

Fig. 1. Different visibility queries in the same polygonal environment. From left to right, first row: two-points visibility, ray-shooting, visibility polygon. Second row: d-visibility region, visible vertices, d-visible vertices. Third row: vertex-vertex visibility graph, point-point visibility graph, vertex-point visibility graph.

II. Definition of Visibility and Related Terms

This section defines the fundamental concepts of visibility and related terms, illustrated with a visual aid in Fig. 1.

Polygonal Environment: A polygonal environment $\mathcal{W}$ is a 2D region bounded by a single outer boundary and zero or more inner boundaries called holes. More formally, $\mathcal{W}$ is a non-empty, connected, closed, bounded subset of 2D Euclidean space. The notion of obstacles is implicitly included in the definition of $\mathcal{W}$, as $\mathcal{W}$ represents the entire see-through space, while its complement, $\mathbb{R}^2 \setminus \mathcal{W}$, represents the opaque space. Furthermore, all boundaries of $\mathcal{W}$ are simple polygons: closed, connected series of pairwise non-

This work was co-funded by the European Union under the project Robotics and advanced industrial production (reg. no. CZ.02.01.01/00/22_008/0004590) and by the Grant Agency of the Czech Technical University in Prague, grant no. SGS23/175/OHK3/3T/13.

[1]All authors are with the Czech Institute of Informatics, Robotics and Cybernetics, Czech Technical University in Prague, Jugoslavskych partyzanu 1580/3, Prague 6, 160 00, Czech Republic. {jan.mikula, miroslav.kulich, libor.preucil}@cvut.cz

[2]Jan Mikula is also with the Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague, Karlovo namesti 293/13, Prague 2, 121 35, Czech Republic.

intersecting line segments. We denote the set of all these segments, united from all the boundaries, as the edges $E_{\mathcal{W}}$ of $\mathcal{W}$, with endpoints referred to as the vertices $V_{\mathcal{W}}$ of $\mathcal{W}$.

Convex Polygonal/Triangular Mesh: Any polygonal environment $\mathcal{W}$ can be represented non-uniquely as a convex polygonal mesh $\mathcal{C}$, consisting of convex polygons such that their union forms $\mathcal{W}$, and the intersection of any two polygons is either empty, a single point, or a single line segment. We denote the set of all edges formed by the polygons as the edges $E_{\mathcal{C}}$ of $\mathcal{C}$, with endpoints referred to as the vertices $V_{\mathcal{C}}$ of $\mathcal{C}$. Unless specified otherwise, we assume that no additional vertices have been added to $\mathcal{C}$, i.e., $V_{\mathcal{C}} = V_{\mathcal{W}}$. Moreover, we consider a structure called a convex polygonal graph, which encapsulates $\mathcal{C}$ along with the topological relationships such as adjacency between the polygons, edges, and vertices. For simplicity, the convex polygonal graph is denoted by the same symbol $\mathcal{C}$ as its corresponding convex polygonal mesh. The process of constructing $\mathcal{C}$ from $\mathcal{W}$ is called convex partitioning. A special instance of the convex polygonal mesh is one in which all polygons are triangles. Both the triangular mesh and its corresponding triangular graph are denoted as $\mathcal{T}$. The process of constructing $\mathcal{T}$ from $\mathcal{W}$ is called triangulation.

Point Location Query: This query is not necessarily related to visibility but is essential for the operation of TřiVis. It involves determining the location of a point $q \in \mathbb{R}^2$ with respect to $\mathcal{W}$, $\mathcal{C}$, or $\mathcal{T}$. The usual objective is to determine whether q lies inside or outside $\mathcal{W}$, and if it lies inside, whether it coincides with one of the vertices $v \in V_{\mathcal{W}}$ or lies on a specific edge $e \in E_{\mathcal{W}}$. For $\mathcal{C}$ and $\mathcal{T}$, the objective is similar, but an additional goal is to identify the polygon or triangle, respectively, that contains q.

Two-Points Visibility Query: Two points $q, p \in \mathcal{W}$ are said to be visible to each other if the line segment $\overline{qp}$ lies entirely within $\mathcal{W}$, i.e., $\overline{qp} \subset \mathcal{W}$. The query definition is as follows: Given $q, p \in \mathcal{W}$, determine whether $\overline{qp} \subset \mathcal{W}$.

Ray Shooting: This visibility-related query aims to find the first intersection point of a half-line, called a ray, with the boundary of $\mathcal{W}$. Given a point $q \in \mathcal{W}$ and a directional vector $u \in \mathbb{R}^n$, which defines the ray, the objective is to find the point $p = \arg\min_{\lambda \geq 0}\{q + \lambda u \mid q + \lambda u \in \partial\mathcal{W}\}$, where $\partial\mathcal{W}$ is the boundary of $\mathcal{W}$. Note that p is always defined because $\mathcal{W}$ is a closed, bounded region, while the ray starts inside $\mathcal{W}$ and extends indefinitely. Additionally, the above expression implies that q and p are visible to each other, i.e., $\overline{qp} \subset \mathcal{W}$.

Visibility Region/Polygon: The visibility region is the set of all points visible from a given point q, defined as $\mathcal{V}(q) = \{p \in \mathcal{W} \mid \overline{qp} \subset \mathcal{W}\}$. In polygonal environments, visibility regions take the form of polygons. These polygons are usually simple, except in cases where one-dimensional structures, known as antennas, are created. Antennas occur when the query point aligns with two vertices of the environment, restricting visibility from opposite sides.

Visible Points/Vertices Query: The visible points query aims to identify a subset of a given finite set of points $P \subset \mathcal{W}$ that are visible from a given query point $q \in \mathcal{W}$. Its definition resembles that of the visibility region, but the output domain is confined to P: $\mathcal{V}_P(q) = \{p \in P \mid \overline{qp} \subset \mathcal{W}\}$. In polygonal environments, we can consider a special case of the visible points query, called the visible vertices query, where $P \subset V_{\mathcal{W}}$. We denote the output in this case as $\mathcal{V}_V(q)$.

Visibility Graph/Subgraphs: The visibility graph is a structure used to represent the pairwise visibility relationships among a given finite set of points $Q \subset \mathcal{W}$. It is defined as a simple undirected graph $G_Q = (V_Q, E_Q)$, where $V_Q = Q$ and $E_Q = \{\{q, p\} \mid q, p \in Q \wedge q \neq p \wedge \overline{qp} \subset \mathcal{W}\}$ are the vertex and edge sets, respectively. In polygonal environments, we can classify three types of visibility subgraphs by writing the set of visibility graph query points as $Q = V \cup P$, where $V \subset V_{\mathcal{W}}$ is a set of vertices, $P \subset V_{\mathcal{W}}$ is a set of non-vertex points, and $V \cap P = \emptyset$. Then, the vertex-vertex visibility graph, denoted as $G_V = (V_V, E_V)$, is the subgraph of G_Q induced by V. The point-point visibility graph, denoted as $G_P = (V_P, E_P)$, is the subgraph of G_Q induced by P. Lastly, the vertex-point visibility graph is defined as $G_{VP} = (V_{VP}, E_{VP})$, where $V_{VP} = V_Q$ and $E_{VP} = E_Q \setminus E_V \setminus E_P$. Note the following relationships: $V_Q = V_{VP} = V_V \cup V_P$, $V_V \cap V_P = \emptyset$, $E_Q = E_V \cup E_P \cup E_{VP}$, and $E_V \cap E_P = E_V \cap E_{VP} = E_P \cap E_{VP} = \emptyset$.

Limited Visibility: Visibility may be subject to additional constraints. In particular, we can consider a limited visibility range d: two points q and p in $\mathcal{W}$ are considered d-visible to each other if the line segment $\overline{qp}$ lies entirely within $\mathcal{W}$ and has a length no greater than d, i.e., $\overline{qp} \subset \mathcal{W} \wedge \|\overline{qp}\| \leq d$. The set of all points d-visible from a given point q is called the d-visibility region and is defined as $\mathcal{V}_d(q) = \{p \in \mathcal{W} \mid \overline{qp} \subset \mathcal{W} \wedge \|\overline{qp}\| \leq d\}$. Note that the d-visibility region may no longer be a polygon, as it may contain circular arcs due to the limited visibility range. Similarly, we could define the d-visibility graph, d-visible points query, or d-visibility ray shooting (with a Null result if $\|\overline{qp}\| > d$).

III. Visibility Algorithms

This paper does not introduce new algorithms, nor does it aim to offer an exhaustive review of existing visibility algorithms for all queries defined in Sec. II. Instead, the presented library is centered around the triangular expansion algorithm (TEA), which has been adapted to handle all these visibility queries in polygonal environments. To provide a partial background, we present a literature review of algorithms for computing visibility polygons in Sec. III-A. Following this, the TEA is elaborated upon in Sec. III-B, and the section concludes with a description of the TřiVis-specific adaptations for handling other queries in Sec. III-C.

A. Literature Background on Visibility Polygons

Computing visibility polygons is a well-explored area within computational geometry originating in the late 1970s [10]. Since then, it has received continuous attention from numerous researchers, primarily focusing on the theoretically proven complexity of the algorithms rather than their practical implementation and experimental validation. Initially, the emphasis was on simple polygons, aiming to

reduce the query time to $\mathcal{O}(n)$, where n denotes the number of vertices in the polygon. The first correct $\mathcal{O}(n)$ solution was proposed by Joe and Simpson [11].

Subsequently, attention shifted towards polygons with holes until Heffernan and Mitchell [12] introduced the optimal $\mathcal{O}(n + h \log h)$ algorithm, where h signifies the number of holes, surpassing the previously leading $\mathcal{O}(n \log n)$-time rotational sweep algorithm (RSA) by Asano [13]. In subsequent decades, numerous algorithms emerged for polygons with holes, offering diverse trade-offs between preprocessing time and query time. For example, Zarei and Ghodsi [14] presented an algorithm with $\mathcal{O}(n^3 \log n)$ preprocessing time and $\mathcal{O}(K + \min(h, K) \log n)$ query time; Inkulu and Kapoor [15] introduced a method with $\mathcal{O}(n^2 \log n)$ preprocessing time and $\mathcal{O}(K \log^2 n)$ query time; and Chen and Wang [16] devised an approach with $\mathcal{O}(n^2 \log n)$ preprocessing time and $\mathcal{O}(K + \log^2 n + h \log(n/h))$ query time.

As previously mentioned, the research discussed thus far has primarily focused on theoretical aspects, with limited attention given to practical implementation, creating a notable gap in the field. This gap was addressed by Bungiu et al. [8], who implemented Joe and Simpson's algorithm for simple polygons [11] and Asano's RSA for polygons with holes [13] within the then-new CGAL 2D Visibility package.

Since the remaining algorithms presented in the literature were deemed too complex for practical implementation, Bungiu et al. [8] also introduced their own solution, the TEA, which they integrated into the same package. Although the TEA, with a worst-case query time of $\mathcal{O}(nh)$, does not rank among the theoretically fastest algorithms, the authors demonstrated that it performs two orders of magnitude faster than the RSA in real-world scenarios, establishing it as the favored algorithm for computing visibility polygons in practice. Although Bungiu et al.'s work is significant, we faced several difficulties while using the CGAL 2D Visibility package for our research, which motivated the development of TřiVis. These difficulties are apparent from our evaluation of the package in Sec. V.

It is worth noting that Xu and Güting [9] independently[1] developed an equivalent algorithm to the TEA for the visible vertices query. However, their work has received little to no attention compared to Bungiu et al.'s [8], as evidenced by the Google Scholar citation count.

B. Triangular Expansion Algorithm

We base our description on [8], as computing visibility polygons is more relevant to our evaluation in Sec. V. The following description assumes that the input polygonal environment $\mathcal{W}$ has been triangulated into $\mathcal{T}$ and that the query point $q \in \mathcal{W}$ has been located inside a triangle $\Delta \in \mathcal{T}$.

Basic Idea and Features: The underlying idea is simple: The TEA computes the visibility polygon $\mathcal{V}(q)$ by recursively traversing neighboring triangles, starting from Δ. The first notable feature is that it exclusively traverses triangles visible from the query point, making it output-sensitive to some extent. However, it may also traverse triangles that do not contribute directly to the boundary of the resulting visibility polygon [8], rendering it only partially output-sensitive. The second significant feature is the simplicity of the operations during mesh traversal. These operations essentially involve answering two orientation predicates per triangle traversal and computing at most two ray-segment intersections when encountering an edge of $\mathcal{W}$.

Detailed Operation of the TEA: Starting at Δ, a recursive expansion procedure is initiated for each of its edges, gradually forming a set of restricted views around q.[2] The recursive procedure expands along the current edge to the neighboring triangle, restricting the view from q between the edge's endpoints. In the new triangle, the other two edges are considered as candidates for the next recursion call only if they intersect the current restricted view from q. The new recursion call is eventually initiated for any of those candidate edges that neighbor another triangle. Otherwise, the edge is identified as a boundary edge of $\mathcal{W}$, and the current view from q is ultimately restricted between its endpoints, stored, and the recursion does not propagate further from that point. Once no further expansions are possible, the resulting $\mathcal{V}(q)$ is the union of all the restricted views formed around q. Additionally, an efficient TEA implementation takes advantage of pre-ordering the neighbor information in either clockwise (cw) or counterclockwise (ccw) order. The expansions then rotate around q while adhering to the same order, naturally forming the union of the restricted views.

Vertex Query: When the query point q coincides with one of the mesh vertices, a special case arises where the TEA must consider the union of all triangles incident to that vertex. This union forms the basis of the resulting visibility polygon, similarly to the initial triangle Δ in the general case. The expansion procedure is initiated for each outer edge of that union, where an outer edge is defined as an edge that is not shared with another triangle in the union. Otherwise, the operation is identical to the one described above.

Complexity Analysis: As stated by Bungiu et al. [8], the worst-case query time is $\mathcal{O}(n^2)$, where $n = |V_{\mathcal{T}}| = |V_{\mathcal{W}}|$. This is because the recursion may split into two views $\mathcal{O}(n)$ times, and each view may reach $\mathcal{O}(n)$ triangles. However, splits into two views that independently reach the same triangle may only occur at the holes of $\mathcal{W}$. This implies that the worst-case query time is rather $\mathcal{O}(nh)$.[3]

Preprocessing: When the input for the TEA is not the triangular mesh $\mathcal{T}$, but instead the polygonal environment $\mathcal{W}$ or the convex polygonal mesh $\mathcal{C}$, the TEA query phase must be preceded by the respective preprocessing phase. The preprocessing involves the triangulation of $\mathcal{W}$ or individual

[1] Xu and Güting's manuscript was submitted for publication nearly six months before Bungiu et al.'s manuscript appeared on ArXiv, which, to our knowledge, remains unpublished. Therefore, it is highly likely that these two independent research groups were unaware of each other's work.

[2] An animated visualization of the TEA operation is available at https://www.youtube.com/watch?v=gKSA6lxVTKw.

[3] The possibility for two independent views to reach the same triangle by splitting at a hole is an essential property of the TEA, which seems to not have been realized by the other independent authors of the TEA, Xu and Güting [9]. Therefore, their complexity analysis, which states the worst-case query time to be $\mathcal{O}(n)$, is incorrect.

polygons in $\mathcal{C}$. For example, this can be achieved by the constrained Delaunay triangulation (CDT), for which exist algorithms running in $\mathcal{O}(n \log n)$ [17] time.

Optimal Mesh: TEA authors have not examined how the choice of triangular mesh affects the query performance of the TEA. To fill this gap, we address this issue in our prior research [18], where we derive the optimal triangular mesh under the assumption that query points are uniformly distributed over $\mathcal{W}$. Unfortunately, constructing this optimal mesh involves solving an NP-hard problem. Thus, we propose a parametric heuristic approach that balances mesh quality and preprocessing time. We assess the TEA's performance with the approximate optimal mesh, denoted as MinVT, on a dataset of complex polygonal environments (the same dataset as used in this paper) and compare it with the performance of the TEA with CDT. Our experiments show that the MinVT mesh can improve the query time by 12–16% compared to CDT, at the expense of 9–212 seconds of preprocessing, depending on the chosen parametrization. While this improvement may not meet our initial expectations, it is still significant enough to justify the use of the MinVT mesh in applications where preprocessing time is not a critical factor. However, for simplicity, in this paper, we opt for the CDT mesh.

C. Adaptations for the Other Visibility Queries

In TřiVis, the TEA is the primary algorithm for computing visibility in polygonal environments. Initially, the library was developed to compute visibility polygons but has since been adapted to handle other visibility queries from Sec. II. A brief overview of these adaptations follows.

Adapting the TEA for the visible vertices query involves a straightforward modification, where the output is now a collection of the traversed triangles' vertices $\mathcal{V}_V(q)$ visible from the query point q. Furthermore, there is no need to compute any intersections with the boundary of $\mathcal{W}$, which reduces the computational effort. The visible points query follows a similar process, but it requires precomputed point location queries with respect to $\mathcal{T}$ for the input set of points P. These input points are then stored within the corresponding triangle containing them. During the query phase, the algorithm looks up the stored points for each traversed triangle and assesses whether they fall within the current restricted view from the query point. The subset of P containing points that have been visible at some point during the traversal is returned as the output at the end of the expansion procedure.

In the context of the two-points visibility query, we denote one of the query points as the source q and the other as the target p. Similar to other queries, TřiVis initiates the expansion from q. However, unlike the previously mentioned queries, it does not need to expand in all directions; rather, it expands directly towards p. This means that only triangles along the line segment $\overline{qp}$ are traversed, which implies that no views are split at holes and the worst-case query time is $\mathcal{O}(n)$ for this query. If the target is reached before encountering the boundary of $\mathcal{W}$, indicating that the target is visible from the source, the algorithm returns true; otherwise, it returns false. By adjusting this approach, we can derive the ray-shooting query, where no target is considered, and only the direction vector u guides the expansions. In this case, the expansion procedure terminates as soon as the first intersection with the boundary of $\mathcal{W}$ is found, and the intersection point is returned.

Computing the visibility graph for a set of query points $Q = V \cup P$ is a more complex task, as it involves computing the three types of visibility subgraphs and, if desired, merging them into a single graph. For the vertex-vertex and vertex-point visibility graph, the visible vertices query is executed for each vertex in V and each point in P, respectively. For the point-point visibility graph, the visible points query is executed for each point in P.

TřiVis can also handle d-visibility queries by incorporating d as an additional input to the TEA, resulting in a variant we denote as d-TEA. A branch of the d-TEA's expansion procedure terminates prematurely if the distance from the query point to the edge being expanded exceeds d. This can significantly reduce the number of traversed triangles, especially for small values of d relative to the expected distance between two points in $\mathcal{W}$ that are visible to each other. To ensure correctness, the query outputs are also adjusted to account for the distance constraint, typically involving straightforward distance checks between the query point and the output points. The most complex case arises with the d-visibility region, where the output of the expansion procedure, $\mathcal{V}'_d$, is a superset of the correct result, $\mathcal{V}_d \subset \mathcal{V}'_d$, and the final output is obtained by intersecting $\mathcal{V}'_d$ with a circle centered at the query point with radius d.

IV. Library Design and Usage

This section outlines TřiVis's design, covering robustness strategies (Sec. IV-A), the visibility query scheme (Sec. IV-B), and point location implementation (Sec. IV-C). We also review the library's internal dependencies in Sec. IV-D and provide a usage code snippet in Sec. IV-E.

A. Robustness Strategies

TřiVis relies on floating-point arithmetic and incorporates a unique combination of adaptive robust geometry predicates [19] and ϵ-geometry [20]. The predicates are employed in TřiVis's expansion procedure, ensuring consistent answers to orientation queries. This prevents the algorithm from encountering infinite loops and considerably enhances its performance compared to using the non-adaptive versions [19].

Additionally, ϵ-geometry is employed to address persistent issues despite the utilization of robust predicates. In particular, we have encountered the following two issues during the development of TřiVis. First, due to numerical inaccuracies, computing the correct intersection points for visibility queries when the query point is near some vertex of $\mathcal{W}$ has been difficult. To address this, we implemented the so-called vertex snapping technique controlled by the ϵ_2 value, as described in Sec. IV-B.

Second, we faced instances where the robust predicates failed to determine that a point lies on a triangle edge, despite the point being computed as the midpoint of that edge. For example, considering triangle vertices represented as vectors a and b, with c computed as $c = (a + b) / 2$ in floating-point representation, the robust geometric predicates indicate that c lies outside the triangle in high percentage of cases. While this behavior is not necessarily a problem of the robust predicates themselves, it is likely undesirable from the user's perspective. To mitigate this issue, we implemented the point location query as a combination of exact and inexact arithmetic, as described in Sec. IV-B, where the inexact arithmetic is controlled by a sequence of increasing ϵ_1 values.

B. General Scheme for Answering Visibility Queries

The TřiVis library consists of a set of C++ classes and functions, with its primary class being TRIVIS. This class serves as the core of the library, responsible for executing visibility queries, managing query input and output, and preprocessing the polygonal environment. Query execution relies on the TEA and its adaptations, detailed in Sec. III-B and Sec. III-C, respectively, and follows a generic scheme outlined in Alg. 1.

Algorithm 1 Generic Visibility Query Scheme

```
1:  function X-QUERY(q, E_1, ε_2, ...)
2:      Δ ← FINDTRIANGLE(q)
3:      while Δ is Null do
4:          if E_1 is Empty then
5:              return Null
6:          ε_1 ← POPSMALLEST(E_1)
7:          Δ ← FINDTRIANGLEWITHEPSILON(q, ε_1)
8:      for v ∈ V_Δ do
9:          if ||v̄q|| ≤ ε_2 then
10:             return X-EXPANDVERTEX(v, ...)
11:     return X-EXPANDTRIANGLE(q, Δ, ...)
```

Each visibility query requires the query point q as mandatory input, except for visibility graphs, which are constructed through multiple calls to this type of query function. Additionally, the query function accepts specific arguments depending on the query type, such as the target point p for the two-points visibility query, directional vector u for the ray-shooting query, and the set of points P for the visible points query, or the distance d for all the d-visibility queries. The query function also accepts a set of ϵ values (a sequence $\mathcal{E}_1$ and a single ϵ_2), which enhance the robustness and control the behavior of the query. The motivation behind these ϵ values is discussed in Sec. IV-A; here, we focus on the precise logic of their usage.

The generic query function in Alg. 1 first attempts to find the triangle Δ containing q using the robust geometric predicates (line 2). If this fails, it resorts to ϵ-geometry, employing the sequence of increasing ϵ_1 values to locate Δ (lines 3–7). If Δ is still not found, it is interpreted that q lies outside $\mathcal{W}$, and the function returns Null (line 5) to indicate this. If Δ is found, the function checks whether q is near any vertices of Δ using the ϵ_2 value (lines 8–9). If so, the vertex query is executed (line 10), expanding all outer edges of the union of triangles incident to the vertex. When $\epsilon_2 > 0$ and $q \neq v$, the operation at lines 8–10 can be interpreted as vertex snapping, where q is snapped to v. Otherwise, the standard expansion procedure is initiated, expanding all edges of Δ. The procedure is recursive, and the function returns the query output once it eventually terminates (line 10 or 11).

C. Bucketing for Point Location

The point location query identifies the triangle Δ containing the query point, which is the first step of Alg. 1. In TřiVis, we implement the bucketing technique from [21] to achieve an average time complexity of $\mathcal{O}(1)$. This technique preprocesses the triangular mesh $\mathcal{T}$ by subdividing the bounding box of $\mathcal{W}$ into square cells called buckets, each storing the triangles intersecting it. During the query phase, the bucket containing the query point is found by rounding the point's coordinates. Finally, the bucket's triangles undergo the point-in-triangle test, resulting in the identification of triangle Δ. The test is performed exactly using the robust predicates at line 2 of Alg. 1, and using the ϵ_1 values at line 7.

D. Internal Dependencies

TřiVis, implemented in C++17, is self-contained, ensuring that its core functionality is independent of external libraries. However, internally, it depends on some third-party software that is freely available for private, research, and institutional use, and is integrated into the library's source code. Most importantly, TřiVis relies on Robust-Predicate[4] for the adaptive robust geometry predicates, and Triangle[5] [22], [23] for computing triangular meshes. Furthermore, Clipper2[6] is integrated for polygon-related operations such as clipping and offsetting. Although the core functionality of TřiVis does not necessarily require Clipper2, it is utilized in the library's evaluation in Sec. V. The remaining core functionality of TřiVis, as outlined in this paper, represents original contributions from the authors.

E. Library Usage

TřiVis offers a user-friendly interface through the TRIVIS class for executing all the visibility queries defined in Sec. II. Fig. 2 shows a code snippet for computing a d-visibility region from a query point q. This snippet is simplified for brevity and requires the user to provide the environment data, the query point, and set the visibility range d (the latter two have default values). The output is a polygonal approximation of the d-visibility region, with circular arcs sampled using line segments at an angle no greater than $\pi/180$. For detailed usage and complete example projects, including visualization, refer to the library's `README.md` on the GitHub repository at https://github.com/janmikulacz/trivis.

[4] Available at https://github.com/dengwirda/robust-predicate.
[5] Available at https://www.cs.cmu.edu/~quake/triangle.html.
[6] Available at https://github.com/AngusJohnson/Clipper2.

```cpp
#include "trivis/trivis.h"
int main() {
  using namespace trivis;
  geom::PolyMap environment; // TODO: Fill the environment.
  Trivis vis{std::move(environment)}; // Construct the visibility object.
  geom::FPoint q{0.0, 0.0}; // TODO: Fill the query point q.
  std::optional<double> d = 10.0; // TODO: Set the visibility range d.
  // Locate the query point q:
  std::optional<Trivis::PointLocationResult> pl = vis.LocatePoint(q);
  if (!pl.has_value()) return EXIT_FAILURE; // q is outside the environment.
  // Compute the d-visibility region from q:
  AbstractVisibilityRegion abs = vis.VisibilityRegion(q, pl.value(), d);
  // Note: abs is an intermediate representation w/o computed intersections.
  RadialVisibilityRegion reg = vis.ToRadialVisibilityRegion(abs);
  if (d.has_value()) reg.IntersectWithCircleCenteredAtSeed(d);
  // Optional post-processing:
  if (d.has_value()) reg.SampleArcEdges(M_PI / 180.0);
  geom::FPolygon poly_approx = reg.ToPolygon();
  // TODO: Do something with the polygonal approx. of the visibility region.
  return EXIT_SUCCESS;
}
```

Fig. 2. Code snippet for computing a d-visibility region from a query point q using TřiVis.

V. Performance Evaluation

TřiVis's performance evaluation focuses on computing visibility polygons in polygonal environments and comparing it with other implementations in terms of runtime behavior and computational time. The evaluation methodology is detailed in Sec. V-A, and the results are presented in Sec. V-B.

A. Methodology

Evaluated Implementations: Among the other evaluated implementations are the RSA [13] and TEA [8] from the CGAL 2D Visibility package, version 5.6.[7] These are evaluated with exact predicates and either exact (CE) or inexact (CI) construction kernels. In addition to CGAL, VisiLibity1[8] is included, with its documentation claiming an average $\mathcal{O}(n \log n)$ query performance with no preprocessing; however, the specific algorithm employed in VisiLibity1 is not disclosed in the documentation.

Map Dataset: Our benchmark environments are derived from a dataset of 35 maps from the Iron Harvest video game, introduced in [24], and were chosen specifically for their scale and complexity. A notable advantage of this dataset is that it includes three representations of the maps—polygonal, mesh, and grid—while ensuring that the maps maintain the same topology. This provides benefits over robotic datasets, which are either limited to grid representations or, if polygonal, are usually too small and lack the complexity of the Iron Harvest maps. We selected the polygonal representation for our purposes. To ensure each environment is well-formed and connected, we preprocess it by selecting the largest polygon to represent the outer boundary and incorporating all its holes. We discard any additional disconnected artifacts, such as polygons that are not connected to the main polygon with holes or those fully enclosed within the holes. The resulting polygonal environments are characterized by thousands of vertices and dozens to hundreds of holes, typically spanning across 400×400 units. We assessed the validity of the resulting environments using CGAL's CGAL::Arrangement_2::is_valid method, excluding a single invalid instance, scene_sp_cha_02, from the dataset.

[7] Available at https://www.cgal.org/.

[8] Available at https://karlobermeyer.github.io/VisiLibity1/.

Fig. 3. Examples of polygonal environments from the Iron Harvest dataset.

To illustrate their complexity, Fig. 3 showcases three instances used in the evaluation.

Query Points: We generated six sets of 1,000 query points for all maps: *In*, *BB*, *Ver*, *NearV*, *Mid*, and *NearM*. The *In* set contains points uniformly distributed inside $\mathcal{W}$, and the *BB* set within its bounding box. The *Ver* set includes randomly selected vertices of $\mathcal{W}$, while the *Mid* set consists of midpoints of edges $\{a, b\} \in \mathcal{T}$, where $\mathcal{T}$ is the CDT of $\mathcal{W}$. The *NearV* and *NearM* sets are based on *Ver* and *Mid* with added normally distributed noise (σ randomly selected from $\{10^{-15}, 10^{-14}, \ldots, 10^{-1}\}$). For the 34 valid maps, this results in 34,000 points per set, totaling 204,000 query points.

Experimental Setup: The TřiVis's epsilon values are set to $\mathcal{E}_1 = (10^{-18}, 10^{-17}, \ldots, 10^{-9})$ and $\epsilon_2 = 10^{-12}$, and the bucket size for the point location query is set to 1 unit. In VisiLibity1, the same value of ϵ_2 is applied for edge and vertex snapping, as recommended in the documentation. Furthermore, all implementations are single-threaded, compiled in Release mode using GCC 12.3.0, and tested on a personal laptop, the Lenovo Legion 5 Pro 16ITH6H, with an Intel Core i7-11800H (4.60GHz), 16GB of RAM, and running Ubuntu 20.04.6 LTS.

Runtime Behaviors—Definitions: We use CGAL's TEA with exact predicates and constructions (CGAL-TEA-CE) as the reference implementation $\mathcal{R}$. For the tested implementation $\mathcal{A}$, we define the following exclusive runtime behaviors:

- *Crash:* $\mathcal{A}$ crashes due to a segmentation fault, unhandled exception or is killed by the OS.
- *Inf:* $\mathcal{A}$ does not finish within its usual time frame likely due to an infinite loop and must be killed manually.
- *NoRef:* $\mathcal{A}$ finishes, but $\mathcal{R}$'s output is unavailable for comparison due to *Crash* or *Inf*.
- *Null:* Both $\mathcal{A}$ and $\mathcal{R}$ return Null.
- *A0R1:* $\mathcal{A}$ returns Null when $\mathcal{R}$ returns a non-Null.
- *A1R0:* $\mathcal{A}$ returns a non-Null when $\mathcal{R}$ returns Null.
- *Same:* $\mathcal{A}$ and $\mathcal{R}$ return the same non-Null output. This is determined by computing the XOR of the two outputs using Clipper2 (recall Sec. IV-D) and checking if the resulting area is at most 10^{-9} times the map area.
- *Weak:* A special case where $\mathcal{A}$ and $\mathcal{R}$ return different non-Null outputs, but the query point is placed precisely on a weakly simple vertex of $\mathcal{W}$. A weakly simple vertex is one with more than one pair of incident edges. For example, two holes touching at a single vertex result in two pairs of incident edges for that vertex.

TABLE I

RUNTIME BEHAVIOR OF THE EVALUATED IMPLEMENTATIONS

Points	$\mathcal{A}$ (tested imp.)	*%Crash*	*%Inf*	*%NoRef*	*%Null*	*%A0R1*	*%A1R0*	*%Same*	*%Weak*	*%Snap*	*%Diff*
In	CGAL-TEA-CE	-	-	-	-	-	-	100.00	-	-	-
	CGAL-TEA-CI	7.418	-	-	-	-	-	92.58	-	-	-
	CGAL-RSA-CE	5.391	-	-	-	-	-	94.06	-	-	0.55
	CGAL-RSA-CI	11.635	-	-	-	-	-	87.92	-	-	0.44
	VisiLibity1	-	-	-	-	-	-	82.86	-	-	17.14
	TřiVis	-	-	-	-	-	-	100.00	-	-	-
BB	CGAL-TEA-CE	-	-	-	42.66	-	-	57.34	-	-	-
	CGAL-TEA-CI	4.838	-	-	42.66	-	-	52.51	-	-	-
	CGAL-RSA-CE	3.456	-	-	42.66	-	-	53.61	-	-	0.28
	CGAL-RSA-CI	7.471	-	-	42.66	-	-	49.64	-	-	0.23
	VisiLibity1	-	-	-	42.66	-	-	47.12	-	-	10.22
	TřiVis	-	-	-	42.66	-	-	57.34	-	-	-
Ver	CGAL-TEA-CE	0.012	0.018	-	0.02	-	-	99.95	-	-	-
	CGAL-TEA-CI	4.282	0.029	-	0.02	-	-	95.67	-	-	-
	CGAL-RSA-CE	3.868	-	0.026	0.02	-	-	95.70	-	-	0.39
	CGAL-RSA-CI	14.103	-	0.024	0.02	-	-	64.74	0.80	-	20.31
	VisiLibity1	-	-	0.029	-	-	0.02	87.57	1.41	-	10.97
	TřiVis	-	-	0.029	-	-	0.02	98.48	1.47	-	-
NearV	CGAL-TEA-CE	-	0.003	-	39.45	-	-	60.54	-	-	-
	CGAL-TEA-CI	2.821	0.003	-	39.45	-	-	57.72	-	-	-
	CGAL-RSA-CE	2.421	-	0.003	39.45	-	-	57.64	-	-	0.49
	CGAL-RSA-CI	5.506	-	0.003	39.45	-	-	52.37	0.06	-	2.60
	VisiLibity1	-	-	0.003	32.79	0.0029	6.67	50.53	0.13	3.69	6.19
	TřiVis	-	-	0.003	24.44	-	15.01	58.01	0.13	2.40	-
Mid	CGAL-TEA-CE	-	-	-	44.81	-	-	55.19	-	-	-
	CGAL-TEA-CI	2.971	-	-	44.81	-	-	52.21	-	-	-
	CGAL-RSA-CE	2.647	-	-	44.81	-	-	52.15	-	-	0.39
	CGAL-RSA-CI	5.194	-	-	44.81	-	-	49.64	-	-	0.35
	VisiLibity1	-	-	-	-	-	44.81	47.47	-	0.02	7.70
	TřiVis	-	-	-	-	-	44.81	55.19	-	-	-
NearM	CGAL-TEA-CE	-	-	-	25.84	-	-	74.16	-	-	-
	CGAL-TEA-CI	3.865	-	-	25.84	-	-	70.29	-	-	-
	CGAL-RSA-CE	3.268	-	-	25.84	-	-	70.36	-	-	0.53
	CGAL-RSA-CI	6.538	-	-	25.84	-	-	67.14	-	-	0.48
	VisiLibity1	-	-	-	17.49	-	8.35	64.54	-	0.27	9.34
	TřiVis	-	-	-	13.11	-	12.73	74.16	-	-	-
Overall	CGAL-TEA-CE	0.002	0.003	-	25.46	-	-	74.53	-	-	-
	CGAL-TEA-CI	4.366	0.005	-	25.46	-	-	70.16	-	-	-
	CGAL-RSA-CE	3.508	-	0.005	25.46	-	-	70.59	-	-	0.44
	CGAL-RSA-CI	8.408	-	0.004	25.46	-	-	61.91	0.14	-	4.07
	VisiLibity1	-	-	0.005	15.49	0.0005	9.98	63.35	0.26	0.66	10.26
	TřiVis	-	-	0.005	13.37	-	12.10	73.86	0.27	0.40	-

- *Snap:* A special case where $\mathcal{A}$ and $\mathcal{R}$ return different non-Null outputs, but the query point has been snapped to some vertex of $\mathcal{W}$.
- *Diff:* $\mathcal{A}$ and $\mathcal{R}$ return different non-Null outputs, and none of the special cases *Weak* or *Snap* apply.

To ensure compatibility with the above behaviors, all implementations have been modified to return Null when detecting that the query point lies outside $\mathcal{W}$. Assuming $\mathcal{R}$ is flawless, the most desirable behaviors are *Same* and *Null*, while the least desirable are certainly *Crash* and *Inf*, followed by *Diff* and *A0R1*. The special cases *Weak* and *Snap* are considered acceptable, as they are not necessarily indicative of an error but may hint at a feature-specific behavior. The same applies to *A1R0* when the query point is ϵ_1-close to an edge of $\mathcal{W}$ (as discussed in Sec. IV-A), but in other cases, it is considered undesirable. Although the *NoRef* case should be impossible assuming $\mathcal{R}$ is flawless, as we demonstrate later in this section, it had to be included as CGAL-TEA-CE occasionally encounters *Crash* or *Inf*.

B. Results

The runtime behavior of the evaluated implementations, presented as a percentage of the number of query points is shown in Tab. I. For better readability, zero values have been replaced with a dash (-). Recall that CGAL-TEA-CE serves as the reference implementation $\mathcal{R}$; thus, where $\mathcal{R}$ defines the behavior, the CGAL-TEA-CE values represent a form of 'comparison to itself.'

TABLE II

COMPUTATIONAL TIME OF THE EVALUATED IMPLEMENTATIONS

$\mathcal{A}$ (tested imp.)	Avg. init. [µs]	Avg. prep. [µs]	Avg. query [µs]	PL [%]
CGAL-TEA-CE	820,793 ± 657,133	8,129 ± 3,464	142 ± 63	51.1
CGAL-TEA-CI	618,622 ± 492,005	5,887 ± 2,446	88 ± 42	64.8
CGAL-RSA-CE	818,563 ± 656,220	-	8,861 ± 3,937	0.8
CGAL-RSA-CI	619,976 ± 494,630	-	5,243 ± 2,355	1.1
VisiLibity1	49 ± 20	-	28,421 ± 12,912	1.2
TřiVis	1 ± 1	16,303 ± 7,037	9 ± 6	2.7

First, note that CGAL implementations have a high crash rate, making them largely unusable. The exception is the reference implementation, CGAL-TEA-CE, which crashed only four times and looped seven times out of 204,000 runs. All these issues occurred with the *Ver* or *NearV* sets. In contrast, VisiLibity1 and TřiVis showed no crashes or infinite loops. Additionally, CGAL implementations, including those with exact constructions (CE), are not mutually consistent in their outputs, as shown by the 0.44% occurrence of *Diff* for CGAL-RSA-CE. Our manual inspections of some of these cases revealed that CGAL-TEA-CE was the one providing correct outputs, making it the most reliable among the CGAL options. VisiLibity1, while not crashing or looping, is inconsistent with the reference in 10.26% of cases, as indicated by *Diff*, and is therefore unreliable as well.

Moving our attention to TřiVis, we see that it is the most reliable, as it never crashed or looped and is consistent with the reference outputs in all cases, except for the *Weak*, *Snap*, and *A1R0* cases, as evidenced by zero occurrences of *Diff* and *A0R1*. As previously mentioned, the *Weak*, *Snap*, and *A1R0* cases are not necessarily indicative of an error but may hint at feature-specific behavior. The *Weak* case occurs only for the *Ver* and *NearV* sets and suggests a different strategy used in handling weakly-simple vertex queries in TřiVis compared to the reference. Upon manual inspection, we found that CGAL-TEA-CE selects only one pair of incident edges as the initial edges for expansion, while TřiVis considers all of them, making it consistent with our visibility region definition. The *Snap* case occurs only for the *NearV* set and is caused by the vertex snapping technique, as described in Sec. IV-B, which helps TřiVis compute all intersection points correctly and prevent more significant errors. Upon manual inspection, we found that TřiVis's outputs for the *Snap* cases align with our expectations and show no indications of more significant errors. The *A1R0* case occurs only for the *Mid* and *NearM* sets. For the *Mid* set, TřiVis returns non-null outputs in 100% of cases. This suggests that TřiVis can compute more visibility polygons in cases where the query point is ϵ_1-close to an edge of $\mathcal{W}$, as discussed in Sec. IV-A, while the reference implementation discards some of these query points as being outside $\mathcal{W}$.

The computational time of the evaluated implementations is presented in Tab. II. This table shows, for the *In* query set, the average initialization (init.), preprocessing (prep.), and query times, along with the percentage of time (out of the query time) spent on the point location query (PL). Note that init. and prep. times are averaged over the 34 maps, while the query time is averaged over the 34,000 query points.

The init. time represents the duration needed to construct the implementation-specific representation of the environment (e.g., CGAL::ARRANGEMENT_2 for CGAL) from a simple polygonal representation. The prep. time covers the more formal preprocessing operations. While RSA and VisiLibity1 do not require preprocessing, the prep. time for the TEA implementations includes the time needed to construct the triangular mesh and bucketing structures for the point location query in TřiVis. Notably, TřiVis requires twice as much prep. time compared to CGAL-TEA-CE, but the total pre-query time is almost two orders of magnitude lower due to the high init. times of the CGAL implementations. Moreover, TřiVis has the lowest query time among all evaluated implementations. It is one order of magnitude lower than CGAL's TEA implementations, four orders lower than CGAL's RSA implementations, and five orders lower than VisiLibity1. It is noteworthy that CGAL's TEA implementations spend about half of the query time on the point location query, which could potentially be improved with a more efficient point location algorithm.[9] Even if we halve the average query time of CGAL-TEA-CE, TřiVis still outperforms it by nearly a factor of eight.

VI. Concluding Remarks

This paper presented TřiVis, a C++ library for computing various visibility-related queries in polygonal environments. We have demonstrated that TřiVis excels in several aspects compared to similar available implementations:

- *Versatility:* TřiVis offers an extensive set of features, each with a specialized, user-friendly interface. These features include computing visibility polygons, performing two-points and ray-shooting visibility queries, identifying visible points and vertices, constructing visibility graphs, and executing rapid point location queries. It also provides various utility functions for managing polygonal environments and query outputs. Additionally, TřiVis supports efficient computations of all queries with an optional limited range constraint.
- *Reliability:* TřiVis is characterized by its reliable and predictable behavior. It avoids crashing or infinite looping and consistently produces outputs that align with user expectations, as confirmed by our evaluation on highly complex query instances.
- *Performance:* With an average query time of $9 \pm 6\,\mu s$, TřiVis outperforms all other implementations by at least an order of magnitude, while maintaining preprocessing times below 20 ms for the benchmark instances.

Moreover, the core functionality is independent of external libraries. TřiVis is freely available for private, research, and institutional use at https://github.com/janmikulacz/trivis.

[9]It is important to note that all CGAL visibility polygon implementations assume that the query point is located inside $\mathcal{W}$ and require the user to provide the edge or vertex on which the query point is located, if applicable. This necessitates using the point location implementations from the CGAL 2D Arrangements package prior to performing the visibility query. We use the CGAL::ARR_WALK_ALONG_LINE_POINT_LOCATION implementation, which was the fastest among those available in CGAL that provided correct results without crashing, based on our preliminary experiments.

References

[1] P. K. Agarwal, E. Ezra, and S. K. Ganjugunte, "Efficient Sensor Placement for Surveillance Problems," in *5th International Conference on Distributed Computing in Sensor Systems*. Springer, 2009, pp. 301–314.

[2] J. Mikula and M. Kulich, "Towards a Continuous Solution of the d-Visibility Watchman Route Problem in a Polygon With Holes," *IEEE Robotics and Automation Letters*, vol. 7, no. 3, pp. 5934–5941, 2022.

[3] M. Kulich and L. Přeučil, "Multirobot search for a stationary object placed in a known environment with a combination of GRASP and VND," *International Transactions in Operational Research*, vol. 29, no. 2, pp. 805–836, 2022.

[4] P. Tokekar and V. Kumar, "Visibility-based persistent monitoring with robot teams," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2015, pp. 3387–3394.

[5] P. Maini, P. Tokekar, and P. B. Sujit, "Visibility-Based Persistent Monitoring of Piecewise Linear Features on a Terrain Using Multiple Aerial and Ground Robots," *IEEE Transactions on Automation Science and Engineering*, vol. 18, no. 4, pp. 1692–1704, 2021.

[6] E. Lozano, I. Becerra, U. Ruiz, L. Bravo, and R. Murrieta-Cid, "A visibility-based pursuit-evasion game between two nonholonomic robots in environments with obstacles," *Autonomous Robots*, vol. 46, no. 2, pp. 349–371, 2022.

[7] L. G. Fletcher, P. Perali, A. Beathard, and J. M. O'Kane, "A Visibility-Based Escort Problem," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2023, pp. 4804–4811.

[8] F. Bungiu, M. Hemmer, J. Hershberger, K. Huang, and A. Kröller, "Efficient Computation of Visibility Polygons," 2014, arXiv:1403.3905.

[9] J. Xu and R. H. Güting, "Querying visible points in large obstructed space," *GeoInformatica*, vol. 19, no. 3, pp. 435–461, 2015.

[10] L. S. Davis and M. L. Benedikt, "Computational models of space: Isovists and isovist fields," *Computer Graphics and Image Processing*, vol. 11, no. 1, pp. 49–72, 1979.

[11] B. Joe and R. B. Simpson, "Corrections to Lee's Visibility Polygon Algorithm," *BIT Numerical Mathematics*, vol. 27, no. 4, p. 458–473, 1987.

[12] P. J. Heffernan and J. S. B. Mitchell, "An Optimal Algorithm for Computing Visibility in the Plane," *SIAM Journal on Computing*, vol. 24, no. 1, pp. 184–201, 1995.

[13] T. Asano, "An efficient algorithm for finding the visibility polygon for a polygonal region with holes," *IEICE Transactions*, vol. 68, no. 9, pp. 557–559, 1985.

[14] A. Zarei and M. Ghodsi, "Query point visibility computation in polygons with holes," *Computational Geometry*, vol. 39, no. 2, pp. 78–90, 2008.

[15] R. Inkulu and S. Kapoor, "Visibility queries in a polygonal region," *Computational Geometry*, vol. 42, no. 9, pp. 852–864, 2009.

[16] D. Z. Chen and H. Wang, "Visibility and ray shooting queries in polygonal domains," *Computational Geometry*, vol. 48, no. 2, pp. 31–41, 2015.

[17] L. P. Chew, "Constrained Delaunay triangulations," in *Proceedings of the third annual symposium on Computational geometry*. ACM Press, 1987, pp. 215–222.

[18] J. Mikula and M. Kulich, "Optimizing Mesh to Improve the Triangular Expansion Algorithm for Computing Visibility Regions," *SN Computer Science*, vol. 5, no. 2, p. 262, 2024.

[19] J. Richard Shewchuk, "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates," *Discrete & Computational Geometry*, vol. 18, no. 3, pp. 305–363, 1997.

[20] S. Fortune and C. J. Van Wyk, "Efficient exact arithmetic for computational geometry," in *Proceedings of the ninth annual symposium on Computational geometry*. ACM Press, 1993, pp. 163–172.

[21] M. Edahiro, I. Kokubo, and T. Asano, "A new point-location algorithm and its practical efficiency: comparison with existing algorithms," *ACM Transactions on Graphics*, vol. 3, no. 2, pp. 86–109, 1984.

[22] J. R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," *Applied Computational Geometry Towards Geometric Engineering*, vol. 1148, pp. 203–222, 1996.

[23] ——, "Delaunay refinement algorithms for triangular mesh generation," *Computational Geometry*, vol. 22, no. 1-3, pp. 21–74, 2002.

[24] D. Harabor, R. Hechenberger, and T. Jahn, "Benchmarks for Pathfinding Search: Iron Harvest," in *Proceedings of the International Symposium on Combinatorial Search*, vol. 15, no. 1, 2022, pp. 218–222.