Visibility Queries in Simple Polygons

Sujoy Bhore
Department of Computer Science and Engineering, Indian Institute of Technology Bombay, Mumbai, India

Chih-Hung Liu
Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan

Anurag Murty Naredla
Department of Computer Science, University of Manitoba, Winnipeg, Canada

Yakov Nekrich
Michigan Technological University, Houghton, MI, USA

Eunjin Oh
POSTECH, Pohang, Republic of Korea

André van Renssen
The University of Sydney, Australia

Frank Staals
Department of Information and Computing Sciences, Utrecht University, The Netherlands

Haitao Wang
Kahlert School of Computing, University of Utah, Salt Lake City, UT, USA

Jie Xue
New York University Shanghai, China

Abstract

Given a simple polygon P with n vertices, we consider the problem of constructing a data structure for visibility queries: for any query point $q \in P$, compute the visibility polygon of q in P. To obtain $O(\log n + k)$ query time, where k is the size of the visibility polygon of q, the previous best result requires $O(n^3)$ space. In this paper, we propose a new data structure that uses $O(n^{2+\epsilon})$ space, for any $\epsilon > 0$, while achieving the same query time. If only $O(n^2)$ space is available, the best known result provides $O(\log^2 n + k)$ query time. We improve this to $O(\log n \log\log n + k)$ time. When restricted to $o(n^2)$ space, the only previously known approach, aside from the $O(n)$-time algorithm that computes the visibility polygon without preprocessing, is an $O(n)$-space data structure that supports $O(k \log n)$-time queries. We construct a data structure using $O(n \log n)$ space that answers visibility queries in $O(n^{1/2+\epsilon} + k)$ time. In addition, for the special case in which q lies on the boundary of P, we build a data structure of $O(n \log n)$ space supporting $O(\log^2 n + k)$ query time; alternatively, we achieve $O(\log n + k)$ query time using $O(n^{1+\epsilon})$ space. To achieve our results, we propose a new method for decomposing simple polygons, which may be of independent interest.

2012 ACM Subject Classification Theory of computation → Computational geometry; Theory of computation → Design and analysis of algorithms

Keywords and phrases simple polygons, visibility polygons, visibility queries, polygon decompositions

Digital Object Identifier 10.4230/LIPIcs.ICALP.2026.33

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: https://arxiv.org/abs/2605.03334

Funding *Sujoy Bhore*: Work supported in part by ANRF ARG-MATRICS, Grant 002465.
Chih-Hung Liu: Supported by Ministry of Education, Taiwan under Yushan Fellow Program with the grant number MOE-111-YSFEE-0003-006-P1.
Yakov Nekrich: Supported by the National Science Foundation under NSF grant 2203278.
Eunjin Oh: Supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.RS-2024-00440239, Sublinear Scalable Algorithms for Large-Scale Data Analysis) and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.RS-2024-00358505).
André van Renssen: This research was partially funded by the Australian Government through the Australian Research Council (project number DP240101353).

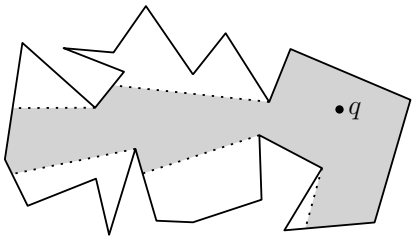

Figure 1 The grey region is Vis(q). Each edge of it is either a portion of an edge of P or a dotted segment (which is called a *window* in the literature [3]).

Acknowledgements This work originated at Shonan Meeting No. 234, "Path Planning and Routing in Geometric Environments." We thank the participants for their valuable discussions, and we are also grateful to the organizers and the National Institute of Informatics for hosting the event.

1 Introduction

Let P be a simple polygon with n vertices. A point $p \in P$ is said to be *visible* to a point $q \in P$ if the line segment $\overline{pq}$ lies entirely inside P. The *visibility polygon* Vis(q) of q is the set of all points in P that are visible to q; see Figure 1. It is well known that Vis(q) is a star-shaped polygon whose vertices are either: (i) vertices of P, or (ii) intersection points between edges of P and rays from q passing through reflex vertices of P.

In this paper, we study data structures for *visibility queries*: given a query point $q \in P$, compute the visibility polygon Vis(q) efficiently.

Previous work. Vis(q) can be computed in $O(n)$ time without any preprocessing [10, 16, 18]. If $O(\log n)$-time ray-shooting queries are available, then Vis(q) can be computed in $O(k \log n)$ time [2], where k is the size of Vis(q). Such ray-shooting data structures can be constructed in $O(n)$ time [9, 10, 20]. Ideally, one would like a query algorithm whose running time is linear in the output size, i.e., $O(k)$. Data structures achieving this are known, but at the cost of larger space. Bose, Lubiw, and Munro [3], and independently Guibas, Motwani, and Raghavan [19], obtained $O(\log n + k)$ query time using $O(n^3)$ space and $O(n^3 \log n)$ preprocessing time. Aronov, Guibas, Teichmann, and Zhang [2] constructed a data structure of $O(n^2)$ space in $O(n^2 \log n)$ time that supports $O(\log^2 n + k)$-time visibility queries. Note that $O(\log n + k)$ is the optimal query time [3, 19].

Our results. There has been no progress on this problem for almost three decades since the conference version of [2] was published in 1998. In this paper, we present the following new results.

- First, we give the first-known subquadratic-space structure that achieves a query time linear in the output size: a data structure of $O(n \log n)$ space, $O(n \log^2 n)$ preprocessing time, and $O(n^{1/2+\epsilon} + k)$ query time. Throughout the paper, ϵ represents an arbitrarily small positive constant. Furthermore, we obtain the following trade-offs between the query time and preprocessing: for a parameter r satisfying $1 \leq r \leq n^{1-\epsilon}$, we can build a data structure of space $O(nr + n \log(n/r))$, $O((nr + n \log(n/r)) \log n)$ preprocessing time, and $O((n/r)^{1/2+\epsilon} + k)$ query time. Note that this result implies that for a large k (e.g., $k = \Omega(n^{\epsilon})$), we can achieve $O(k)$ query time with sub-quadratic preprocessing.

- Second, to achieve the optimal $O(\log n + k)$ query time, we construct a data structure of $O(n^{2+\epsilon})$ space and preprocessing time, for any $\epsilon > 0$. This improves the space bound of the previous $O(n^3)$-space structures [3, 19] by nearly a linear factor.
- Third, under an $O(n^2)$ space budget, we improve the $O(\log^2 n + k)$ query time in [2] to $O(\log n \log\log n + k)$, with a preprocessing time of $O(n^2 \log n)$.
  We remark that even a near-logarithmic improvement in query time is often considered significant. Indeed, for the closely related ray-shooting problem, reducing the query time from $O(\log^2 n)$ to $O(\log n)$ required substantial technical effort [9, 10, 20].

Table 1 summarizes these bounds alongside prior work.

Table 1 Summary of the results for visibility queries (k is the output size).

Results	Query time	Space	Preprocessing time
[9, 20]	$O(k \log n)$	$O(n)$	$O(n)$
[3, 19]	$O(\log n + k)$	$O(n^3)$	$O(n^3 \log n)$
[2]	$O(\log^2 n + k)$	$O(n^2)$	$O(n^2 \log n)$
Ours (near-linear space)	$O(n^{1/2+\epsilon} + k)$	$O(n \log n)$	$O(n \log^2 n)$
Ours (optimal query time)	$O(\log n + k)$	$O(n^{2+\epsilon})$	$O(n^{2+\epsilon})$
Ours (quadratic space)	$O(\log n \log\log n + k)$	$O(n^2)$	$O(n^2 \log n)$

In addition, we study a natural special case in which the query point q lies on the boundary of P. To the best of our knowledge, this *boundary case* has not been treated separately in the literature. We obtain a data structure of $O(n \log n)$ space and $O(n \log^2 n)$ preprocessing time that answers queries in $O(\log^2 n + k)$ time. Alternatively, we can achieve $O(\log n + k)$ query time using $O(n^{1+\epsilon})$ space and preprocessing time.

Other related work. Several related visibility query problems have also been studied. Chen and Daescu [11] considered a *segment-restricted* version, where the query point q lies on a given segment in P. Their data structure uses $O(n)$ space and supports queries in $O(\log n + k)$ time. However, their approach does not appear to extend to arbitrary query points.

Given a segment $s \subseteq P$, a point is *weakly visible* to s if it is visible to at least one point of s. The *weak visibility polygon* of s consists of all points of P visible to s, and can be computed in $O(n)$ time [18]. The corresponding weak visibility query problem, i.e., computing this polygon for an arbitrary query segment, has been investigated in several works [2–4, 13]. Using $O(n)$ space and preprocessing time, one can support queries in $O(k \log n)$ time; alternatively, using $O(n^3)$ space and preprocessing time, queries can be answered in $O(\log n + k)$ time [13].

As a special visibility problem, the *ray-shooting* problem asks for the first point on ∂P hit by a ray originating inside P. As mentioned earlier, ray-shooting queries can be answered in $O(\log n)$ time after $O(n)$-time preprocessing [9, 20].

Visibility and ray-shooting in polygons with holes have also been studied [1, 12, 21, 23, 25, 27], but the presence of holes makes the problems substantially more difficult. We refer the reader to [12] for an overview of results in that setting.

1.1 An overview of our approach

The methods of [3, 19] decompose P into $O(n^3)$ cells such that the combinatorial structure of $\mathrm{Vis}(q)$ is the same for every point q in the same cell. Since the decomposition itself has cubic size, these approaches cannot be directly adapted when one wishes to use subcubic space.

The approach [2] instead builds on the *balanced decomposition* of P [10, 17, 18] (see Section 2 for more details). Their method achieves $O(\log^2 n + k)$ query time using $O(n^2)$ space. The $O(\log^2 n)$ factor in query time appears inherent to the balanced decomposition: each subpolygon may be connected to the "outside world", i.e., the rest of P, by $O(\log n)$ diagonals and the decomposition has $O(\log n)$ recursive levels.

To obtain $O(\log n + k)$ query time with subcubic space, we therefore require a new strategy. Our main idea is to introduce a *new polygon decomposition* with a crucial structural feature (which we call *the gate property*): every subpolygon connects to the outside world through a *single* diagonal, which we call its *gate*. As in the balanced decomposition, we recursively divide P into subpolygons whose sizes shrink geometrically; however, the gate property fundamentally changes how visibility information propagates across subregions.

Intuitively, suppose a query point q lies in a subpolygon P_d with gate d. If we have already computed the portion of P_d visible to q, denoted by $\mathrm{Vis}(P_d, q)$, then the remaining part of the visibility region, $\mathrm{Vis}(P \setminus P_d, q)$, is exactly the portion of $P \setminus P_d$ visible through the diagonal d. We refer to computing this region as the *diagonal-separated subproblem*. The entire query algorithm reduces to solving this subproblem and recursing inside P_d (i.e., to compute $\mathrm{Vis}(P_d, q)$, we can apply the algorithm recursively).

Our optimal-query-time data structure is built on this decomposition and uses $O(1)$ recursive steps by choosing a parameter $r = n^\epsilon$. This yields $O(n^{2+\epsilon})$ space and $O(\log n + k)$ query time. Our quadratic-space data structure also relies on our new decomposition but groups its levels into $O(\log \log n)$ *super-levels*. This reduces the space to $O(n^2)$ at the cost of increasing the query time to $O(\log n \log \log n + k)$. In contrast, the balanced decomposition of [2] has $\Theta(\log n)$ levels, resulting in an unavoidable $O(\log^2 n)$ factor.

Our near-linear space data structure follows the high-level strategy of [2] but solves the diagonal-separated subproblem more efficiently. While [2] used an $O(n^2)$-space data structure supporting $O(\log n + k)$ queries for this subproblem, we instead reduce the problem to simplex stabbing queries and adapt the recent structure of Chan and Zhang [5]. This gives an $O(n)$-space structure supporting queries in $O(n^{1/2+\epsilon} + k)$ time, yielding our result.

For the boundary case, the diagonal-separated subproblem simplifies substantially: with q constrained to lie on the boundary of P, the remaining computation becomes essentially one-dimensional. We obtain a solution of $O(n)$ space and $O(\log n + k)$ query time. Plugging this into the balanced decomposition method of [2], we obtain our first data structure for the boundary case. Plugging this into our new decomposition yields our second data structure.

Outline. In the following, after introducing notation in Section 2, we present our near-linear space data structure in Section 3. In Section 4, we describe our new polygon decomposition, which serves as a key component for the optimal-query-time data structure in Section 5 and the quadratic-space solution in Section 6. Due to the space limit, many details and proofs are omitted, but can be found in the full paper. The boundary case is also in the full paper.

2 Preliminaries

We follow the notation introduced in Section 1, e.g., P, n, and $\mathrm{Vis}(q)$. For any subpolygon $P' \subseteq P$, we define $\mathrm{Vis}(P', q) = \mathrm{Vis}(q) \cap P'$, i.e., the portion of $\mathrm{Vis}(q)$ contained in P'.

For convenience, we assume that each edge of P is an open segment that does not include its endpoints; thus, the vertices of P are disjoint from its edges. We refer to either a vertex or an edge of P as an *element* of P. For any geometric object R, let ∂R denote its boundary. For two points p and q, $\overline{pq}$ denotes the line segment connecting them.

It is well known that $\mathrm{Vis}(q)$ is star-shaped and can be represented by a cyclic (say, clockwise) list of its vertices. Alternatively, given the cyclic list of vertices and edges of P that appear on the boundary $\partial\mathrm{Vis}(q)$, one can reconstruct $\mathrm{Vis}(q)$ in $O(|\mathrm{Vis}(q)|)$ time [2,3]. Hence, to compute $\mathrm{Vis}(q)$, it suffices to determine this cyclic list, called the *combinatorial representation* of $\mathrm{Vis}(q)$ [2]. It is also known that this cyclic order of vertices and edges of P on $\partial\mathrm{Vis}(q)$ is consistent with their order along ∂P. In the following, depending on the context, we may use $\mathrm{Vis}(q)$ to refer to its combinatorial representation.

Balanced decomposition. We will use the *balanced decomposition* of P, which has been applied to various problems in simple polygons [2, 10, 17, 18]. A *diagonal* is a line segment connecting two vertices of P that is fully contained in P but not an edge of P. Chazelle [6] proved that P always admits a diagonal that splits it into two subpolygons, each having between $n/3$ and $2n/3$ vertices. The balanced decomposition of P is obtained by recursively partitioning each subpolygon using such diagonals until all subpolygons are triangles. This decomposition can be represented by a tree $\mathcal{T}(P)$, called the *BD-tree*. Each node v of $\mathcal{T}(P)$ corresponds to a subpolygon P_v of P: the root corresponds to P itself, and each leaf corresponds to a triangle. The diagonal used to partition P_v into its two child subpolygons is stored at v and denoted by d_v. The triangles at all leaves of $\mathcal{T}(P)$ form a triangulation of P. The entire decomposition and BD-tree $\mathcal{T}(P)$ can be computed in $O(n)$ time [7, 17].

Convention on stating query time. For simplicity, when we state that a data structure supports queries in $O(Q(n))$ time for computing $\mathrm{Vis}(q)$ (or a portion thereof, e.g., $\mathrm{Vis}(P', q)$ for a subpolygon $P' \subseteq P$), we mean that a (balanced) binary search tree storing the combinatorial representation of $\mathrm{Vis}(q)$ can be computed in $O(Q(n))$ time, and the full visibility polygon $\mathrm{Vis}(q)$ can then be output in an additional $O(|\mathrm{Vis}(q)|)$ time.

For conciseness, we say that a data structure has complexity $O(T(n), S(n), Q(n))$ if its preprocessing time, space, and query time are $O(T(n))$, $O(S(n))$, and $O(Q(n))$, respectively.

3 A near-linear space data structure

In this section, we present our near-linear space data structure. Our algorithm follows the general framework of [2], but we address a key component, the diagonal-separated subproblem, in a different way, as discussed in Section 1.1. This subproblem arises in essentially all of our subsequent data structures. The same framework will also be partially used in our quadratic-space data structure in Section 6 and in the boundary case. Therefore, the discussions in this section are also instrumental to the later parts of the paper.

We first discuss the diagonal-separated subproblem in Section 3.1, present its linear-space solution in Section 3.2, and finally describe the general algorithmic framework in Section 3.3.

3.1 A diagonal-separated subproblem

Let d be a diagonal that divides P into two subpolygons P_ℓ and P_r. Without loss of generality, assume that d is vertical and P_ℓ lies locally to the left of d. The *diagonal-separated subproblem* is to compute $\mathrm{Vis}(P_\ell, q)$ for any query point $q \in P_r$. We describe our solution below.

Since $q \in P_r$, for any point $p \in \mathrm{Vis}(P_\ell, q)$, $\overline{pq}$ must cross d. In other words, the visibility of q within P_ℓ occurs "through" d. Because P is a simple polygon, $\mathrm{Vis}(d, q)$ is a contiguous segment of d [2]. Let $s = \mathrm{Vis}(d, q)$. If $s = \emptyset$, then $\mathrm{Vis}(P_\ell, q) = \emptyset$. Otherwise, let $C_q(s)$ denote the *cone* formed by the rays from q through all points $p \in s$; we say that $C_q(s)$ is *defined* by q and s, and refer to q as the *apex* of the cone (see Figure 2).

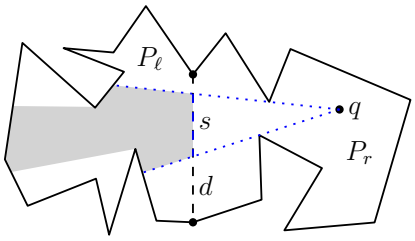

Figure 2 The gray region is $\mathrm{Vis}(P_\ell, q)$. $C_s(q)$ is bounded by the two blue dotted segments.

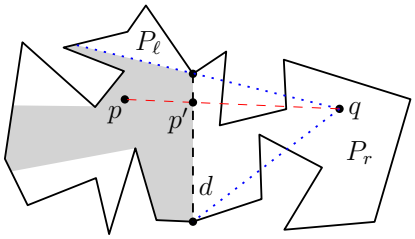

Figure 3 The gray region is $\mathrm{Vis}_d(q)$.

Using the two-point shortest path data structure of Guibas and Hershberger [17] (henceforth referred to as the *GH data structure*), the two bounding rays of $C_q(s)$ can be computed in $O(\log n)$ time via shortest-path queries. To compute $\mathrm{Vis}(P_\ell, q)$, it thus suffices to determine the portion of P_ℓ visible to q through the cone $C_q(s)$.

We use $\mathrm{Vis}_s(q)$ to denote $\mathrm{Vis}(P_\ell, q)$. We also define an analogous notion $\mathrm{Vis}_d(q)$, which consists of all points $p \in P_\ell$ such that $\overline{pq}$ crosses d and $\overline{pp'} \subseteq P_\ell$, where $p' = d \cap \overline{pq}$. Note that whether the segment $\overline{qp'}$ lies inside P is irrelevant. See Figure 3.

Following [2], we first compute a binary search tree that stores the combinatorial representation of $\mathrm{Vis}_d(q)$, i.e., the cyclic order of the elements of P_ℓ (recall that an element refers to either a vertex or an edge of P) appearing on $\partial\mathrm{Vis}_d(q)$. We then perform a *clipping* operation on this tree using $C_q(s)$; the resulting tree represents $\mathrm{Vis}_s(q)$.

▶ **Observation 1.** [2, 3] *The order of the vertices and edges of P_ℓ along $\partial Vis_s(q)$ (resp., $\partial Vis_d(q)$) is consistent with their order along ∂P_ℓ.*

We sort all elements of P_ℓ along its boundary and assign each of them an *index* in this order. Let $A_d(q)$ denote the set of elements of P that appear on $\partial\mathrm{Vis}_d(q)$. By Observation 1, the elements of $A_d(q)$ are sorted in index order in the combinatorial representation of $\mathrm{Vis}_d(q)$.

For each vertex $v \in P_\ell$, as discussed above, all points of d visible to v within P_ℓ form a segment s_v of d. Let C_v denote the cone defined by s_v and v. Observe that v belongs to $A_d(q)$ if and only if $q \in C_v$. For each edge e of P_ℓ, as discussed in [18], the points of d weakly visible to e also form a segment s_e of d, such that e belongs to $A_d(q)$ if and only if q lies in the cone C_e defined by s_e and a point to the left of d (see Figure 4).

For reference, we summarize the above discussion in the following observation.

▶ **Observation 2.** *For each vertex v (resp., edge e) of P_ℓ, v (resp., e) appears in the combinatorial representation of $Vis_d(q)$ if and only if q lies in the cone C_v (resp., C_e).*

All cones C_v and C_e for the vertices and edges of P_ℓ can be computed in $O(n)$ time using the shortest path trees of the two endpoints of d [18]. Let $\mathcal{C}$ denote the set of all these cones. We refer to v (resp., e) as the *generator* of C_v (resp., C_e). To compute $A_d(q)$, by

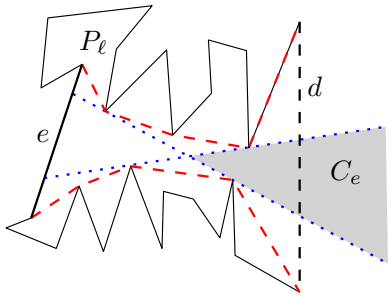

Figure 4 The gray region is C_e. The two dashed red paths are shortest paths from the two endpoints of e to the two vertices of d. The two bounding lines of C_e are tangent to these two paths.

Observation 2, the problem reduces to the following *cone stabbing query*: given q, compute all cones of $\mathcal{C}$ that are stabbed by q (we say that a cone is *stabbed* by q if the cone contains q). To accommodate the clipping operation with $C_q(s)$, we require the data structure to return a binary search tree storing the generators of the stabbed cones in their index order.

A quadratic-space solution. A quadratic-space solution is given in [2]. We briefly review it below as it also serves as a subroutine in some of our later algorithms.

Let R_d denote the halfplane to the right of the supporting line of d. If $q \notin R_d$, then $\mathrm{Vis}_d(q) = \emptyset$, so we assume $q \in R_d$. Let $\mathcal{A}_d$ be the arrangement in R_d formed by the supporting lines of the bounding rays of all cones in $\mathcal{C}$. By Observation 2, points q in the same cell of $\mathcal{A}_d$ share the same (combinatorial representation of) $\mathrm{Vis}_d(q)$, and $\mathrm{Vis}_d(q)$ of q in adjacent cells differ by at most one element of P_ℓ. Since there are $O(n)$ cones, $\mathcal{A}_d$ has $O(n^2)$ cells.

We use persistent binary search trees [14, 26] to store the combinatorial representations of $\mathrm{Vis}_d(q)$ for all cells of $\mathcal{A}_d$ in a total of $O(n^2)$ space and $O(n^2 \log n)$ preprocessing time. Additionally, we construct a point location data structure on $\mathcal{A}_d$ in $O(n^2)$ time and space [15, 22, 26]. Given a query point q, we can locate the cell of $\mathcal{A}_d$ containing q in $O(\log n)$ time and thereby obtain access to the corresponding binary search tree that stores $\mathrm{Vis}_d(q)$. This yields a data structure of complexity $O(n^2 \log n, n^2, \log n)$, i.e., $O(n^2 \log n)$ preprocessing time, $O(n^2)$ space, and $O(\log n)$ query time for computing $\mathrm{Vis}(P_\ell, q)$ for any $q \in P_r$.

The boundary case. In the boundary case, each query point q is on $\partial P \cap P_r$. In this case the cone stabbing queries become 1D *interval stabbing* queries on $\partial P \cap P_r$. Using persistent trees [14, 26], we obtain a data structure of complexity $O(n \log n, n, \log n)$. This result combining with the algorithmic framework given in Section 3.3 yields a data structure of complexity $O(n \log^2 n, n \log n, \log^2 n)$ for computing $\mathrm{Vis}(q)$. See the full paper for details.

3.2 A linear-space solution to the diagonal-separated subproblem

We now present our solution to the cone stabbing problem and thus solve the diagonal-separated subproblem.

Overview. Our algorithm computes a collection of "canonical subsets" of $\mathcal{C}$, where each subset is represented by a binary search tree storing the cone generators in their index order. We perform a clipping operation on each tree using the cone $C_s(q)$. The elements remaining in the clipped trees of all canonical subsets collectively form the set $A_s(q)$, which consists of the elements of P lying on $\partial\mathrm{Vis}_s(q)$. We then sort the elements of $A_s(q)$ by their index order to obtain $\mathrm{Vis}_s(q)$. Finally we build a binary search tree to store them and return this tree as the answer to the query. The details are given below.

We adapt Chan and Zhang's data structure [5, Section 3.1] for simplex stabbing queries (given a query point q, compute all simplices stabbed by q among a set of given simplices; each of our cones is a special simplex in 2D). The data structure has multiple levels and it is constructed using Matoušek's simplicial partition [24]. To solve our problem, we can build the data structure on the cones of $\mathcal{C}$ with $j = 2$ levels. We briefly discuss this and the reader is referred to [5, Section 3.1] for details.

Determining whether q is in a cone is equivalent to testing whether q is in the intersection of two halfplanes bounded by the cone's bounding lines. In the dual plane, this is to check whether the dual line q^* of q lies in the "correct" side of the two dual points of the bounding lines of the cone. We apply Matoušek's simplicial partition [24] on the dual points of the j-th halfplanes of the cones of $\mathcal{C}$, with $j = 2$. For each cell Δ of the partition that intersects q^*, we recurse on the canonical subset of the dual points of Δ. For each cell Δ lying entirely below q^* (or above q^*, depending on which is the correct side), we recurse on the canonical subset of Δ but as a level-$(j-1)$ problem. For each canonical subset in a level-0 problem, we build a binary search tree to store all generators of cones in the subset. Following the analysis in [5, Section 3.1], the resulting data structure is of $O(n)$ space and can be constructed in $O(n \log n)$ time.

Given a query point q, the query algorithm returns $O(n^{1/2+\epsilon})$ canonical subsets whose union corresponds to the set of cones of $\mathcal{C}$ stabbed by q. For each canonical subset, we perform a clipping operation using the cone $C_q(s)$, taking $O(\log n)$ additional time. Hence, we can compute $A_s(q)$ in $O(n^{1/2+\epsilon} \log n + |A_s(q)|)$ time, which simplies to $O(n^{1/2+\epsilon} + |A_s(q)|)$ since the $\log n$ factor is absorbed by n^ϵ.

Once $A_s(q)$ is obtained, we must sort its elements by their index order. A straightforward comparison sort would require $O(|A_s(q)| \log |A_s(q)|)$ time, introducing an undesired logarithmic factor in the overall query time. However, since the index of each element of $A_s(q)$ is an integer in $[1, 2n]$, we can perform the sorting in $O(n^\epsilon + |A_s(q)|)$ time using radix sort in the following lemma.

▶ **Lemma 3.** *For any $\epsilon > 0$, the elements of $A_s(q)$ can be sorted by their indices in $O(n^\epsilon + |A_s(q)|)$ time.*

Proof. Let I denote the set of indices of the elements in $A_s(q)$, and our goal is to sort the integers in I. Since each index in I is an integer in $[1, 2n]$, it can be represented using at most $\log n + 1$ bits. We partition these $\log n + 1$ bits into $O(1)$ groups, each consisting of $\epsilon \log n$ bits. We refer to each such group as a *digit*. Hence, each integer in I has $O(1)$ digits. We apply radix sort on the integers of I using these digits. Sorting I by a single digit requires $O(2^{\epsilon \log n} + |I|) = O(n^\epsilon + |I|)$ time. Since there are only $O(1)$ digits, the entire sorting process takes $O(1)$ passes, resulting in a total running time of $O(n^\epsilon + |I|)$. ◀

After $A_s(q)$ is sorted, we obtain the combinatorial representation of $\mathrm{Vis}_s(q)$. We then construct a binary search tree to store this representation in an additional $O(|\mathrm{Vis}_s(q)|)$ time. Combining all the above results yields the following lemma.

▶ **Lemma 4.** *Given a simple polygon P of n vertices and a diagonal d dividing P into two subpolygons P_ℓ and P_r, a data structure of $O(n)$ space can be constructed in $O(n \log n)$ time such that $Vis(P_\ell, q)$ can be computed in $O(n^{1/2+\epsilon} + |Vis(P_\ell, q)|)$ time for any point $q \in P_r$.*

3.3 A general algorithmic framework

We now describe the algorithmic framework. In the preprocessing, we compute a balanced decomposition of P together with the BD-tree $\mathcal{T}(P)$ as defined in Section 2.

Given a query point $q \in P$, let v_q be the leaf of $\mathcal{T}(P)$ whose triangle P_{v_q} contains q. The query algorithm proceeds in a bottom-up manner along the path from v_q to the root of $\mathcal{T}(P)$. For each node v on this path, let u denote the parent of v and w the sibling of v (i.e., the other child of u). Suppose that $\mathrm{Vis}(P_v, q)$ has already been computed (this is true initially when $v = v_q$ since $\mathrm{Vis}(P_v, q) = P_{v_q}$). Our goal at node u is to compute $\mathrm{Vis}(P_u, q)$, which equals the union of $\mathrm{Vis}(P_v, q)$ and $\mathrm{Vis}(P_w, q)$.

By definition, P_u is partitioned into P_v and P_w by the diagonal d_u. Since $\mathrm{Vis}(P_v, q)$ is known, we next compute $\mathrm{Vis}(P_w, q)$ and then merge $\mathrm{Vis}(P_v, q)$ and $\mathrm{Vis}(P_w, q)$ along d_u to obtain $\mathrm{Vis}(P_u, q)$. Computing $\mathrm{Vis}(P_w, q)$ is precisely an instance of the diagonal-separated subproblem. Therefore, in the preprocessing, for each node $u \in \mathcal{T}(P)$, we construct a diagonal-separated data structure for both P_v and P_w (with respect to d_u). During a query, these data structures allow us to compute $\mathrm{Vis}(P_w, q)$ efficiently.

Once $\mathrm{Vis}(P_w, q)$ is computed, we obtain $\mathrm{Vis}(P_u, q)$ by merging $\mathrm{Vis}(P_w, q)$ and $\mathrm{Vis}(P_v, q)$ along $s = \mathrm{Vis}(d_u, q)$. Note that if $s = \emptyset$, then $\mathrm{Vis}(P_w, q) = \emptyset$; otherwise, s must be an edge of $\mathrm{Vis}(P_v, q)$. Merging $\mathrm{Vis}(P_v, q)$ and $\mathrm{Vis}(P_w, q)$ can be done in $O(\log n)$ time. For reference purpose, we use Lemma 5 to summarize this merging step (see the full paper for the proof). After computing $\mathrm{Vis}(P_u, q)$, the algorithm continues upward until reaching the root of $\mathcal{T}(P)$, at which point $\mathrm{Vis}(q)$ is obtained.

▶ **Lemma 5.** *Suppose a diagonal d divides P into two subpolygons P_ℓ and P_r. Let q be a point in P_r. Given a binary search tree storing $\mathrm{Vis}(P_r, q)$ and a binary search tree storing $\mathrm{Vis}(P_\ell, q)$, we can obtain a binary search tree storing $\mathrm{Vis}(q)$ in $O(\log n)$ time.*

If we plug the quadratic-space diagonal-separated data structure of [2] (also reviewed in Section 3.1) into the framework, then, since the sizes of the subpolygons decrease geometrically along any root-to-leaf path of $\mathcal{T}(P)$, we obtain a data structure of complexity $O(n^2 \log n, n^2, \log^2 n)$ for computing $\mathrm{Vis}(q)$ for any $q \in P$. This is the result of [2].

If we instead apply our linear-space result from Lemma 4, then, because the total size of all subpolygons at each level of $\mathcal{T}(P)$ is $O(n)$ (as they form a partition of P), we obtain a data structure of complexity $O(n \log^2 n, n \log n, n^{1/2+\epsilon})$. Furthermore, by applying a somewhat standard tradeoff technique based on hierarchical cuttings [8] in conjunction with Lemma 4, we can obtain the following result (see the full paper for the detailed proof).

▶ **Theorem 6.** *Given a simple polygon P with n vertices, let r be a parameter satisfying $1 \leq r \leq n^{1-\epsilon}$ for any constant $\epsilon > 0$. A data structure of size $O(nr + n \log(n/r))$ can be constructed in $O((nr + n \log(n/r)) \log n)$ time such that $\mathrm{Vis}(q)$ can be computed in $O((n/r)^{1/2+\epsilon})$ time for any query point $q \in P$. In particular, setting $r = 1$ yields a data structure of $O(n \log n)$ space, $O(n \log^2 n)$ preprocessing time, and $O(n^{1/2+\epsilon})$ query time.*

4 A new polygon decomposition

In this section, we introduce a new decomposition of P together with its associated data structure for answering visibility queries.

Our decomposition decomposes P into *geodesic triangles*, a concept first introduced in [9], in which a balanced geodesic triangulation of P was proposed and it also decomposes P into geodesic triangles. However, our decomposition differs fundamentally from the balanced geodesic triangulation in [9] and possesses several key properties, such as the gate property described in Section 1.1, that are essential for the efficiency of our visibility query algorithm.

In the following, after defining geodesic triangles, we present a data structure for handling visibility queries with respect to a single geodesic triangle in Section 4.1. We then describe our new polygon decomposition in Section 4.2.

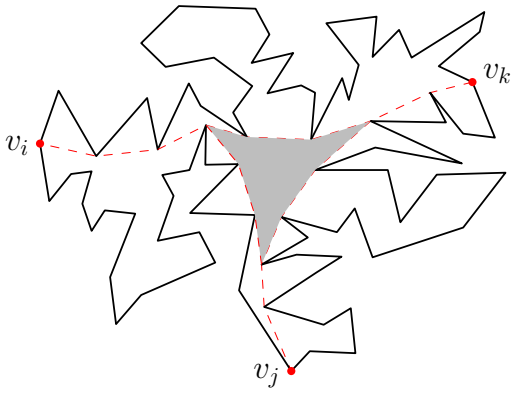

Figure 5 Illustrating a geodesic triangle (the gray region) formed by three geodesic paths.

Geodesic triangles. The *geodesic path* $\pi(p, q)$ is a shortest path in P between two points p and q. For three vertices v_i, v_j, and v_k of P ordered clockwise along ∂P, consider the three geodesic paths $\pi(v_i, v_j)$, $\pi(v_j, v_k)$, and $\pi(v_k, v_i)$. If we remove the subpaths shared by any two of these three paths, then the region bounded by the remaining portions of the three paths is a *geodesic triangle*; see Figure 5. A geodesic triangle has three *sides*, each being a concave chain that is a subpath of one of the three geodesic paths, and three *apexes*, corresponding to the endpoints of its sides. In general, a geodesic triangle has exactly three apexes, although it may degenerate to an empty region when one of the three vertices v_i, v_j, or v_k lies on the geodesic path connecting the other two.

4.1 A data structure for geodesic triangles

Consider a geodesic triangle $\triangle$, defined by three vertices p_i, p_j, and p_k as discussed above. Each edge of $\triangle$ is either an edge of P or a diagonal. For each diagonal d on $\partial\triangle$, let $P_d(\triangle)$ denote the subpolygon of P bounded by d that does not contain $\triangle$. We refer to $P_d(\triangle)$ as the *pocket of P separated by d from $\triangle$*. For simplicity, if d is an edge of P, we let $P_d(\triangle)$ denote d itself. When $\triangle$ is clear from context, we simply write P_d.

We study two subproblems with respect to $\triangle$. In the first subproblem, the query point q lies in $\triangle$, and the goal is to compute $\mathrm{Vis}(q)$. In the second subproblem, the query point q lies in P_d for some diagonal d of $\partial\triangle$, and we wish to compute $\mathrm{Vis}(P \setminus P_d, q)$, i.e., the portion of $\mathrm{Vis}(q)$ that lies outside the pocket P_d. Our result is summarized in the following lemma, which will be used later. The proof is in the full paper.

▶ **Lemma 7.** *With respect to a geodesic triangle $\triangle$, a data structure of size $O(n^2)$ can be constructed in $O(n^2 \log n)$ time such that, for any query point q, if $q \in \triangle$, then $\mathrm{Vis}(q)$ can be computed in $O(\log n)$ time, and if $q \in P_d$ for some diagonal d on the boundary of $\triangle$, then $\mathrm{Vis}(P \setminus P_d, q)$ can be computed in $O(\log n)$ time.*

Intuitively, the motivation for having a query algorithm that computes $\mathrm{Vis}(P \setminus P_d, q)$ is as follows. If $\mathrm{Vis}(P_d)$ is already available, then after obtaining $\mathrm{Vis}(P \setminus P_d, q)$, we can compute $\mathrm{Vis}(q)$ by merging the two. To compute $\mathrm{Vis}(P_d)$ itself, we would like to apply the same algorithm recursively. For this recursive approach to work, we require an "appropriate" polygon decomposition of P. This motivates the new decomposition introduced in Section 4.2.

4.2 The new decomposition

We now introduce a new decomposition of P, defined with respect to a designated edge of P. Let $p_1, p_2, \ldots, p_n$ be the vertices of P ordered clockwise along ∂P. In the following, we describe the decomposition with respect to the edge $\overline{p_1 p_n}$.

We represent the decomposition by a *decomposition tree* Ψ. Let v^* denote the root of Ψ. Each node v of Ψ corresponds to a subpolygon P_v and a *gate* d_v, where d_v is a diagonal of P that separates P_v from the "outside world" $P \setminus P_v$, and all other edges of P_v are edges of P. Initially, we have $P_{v^*} = P$ and $d_{v^*} = \overline{p_1 p_n}$ (since v^* is the root, as a special case we allow d_{v^*} not to be a diagonal, i.e., we assume d_{v^*} connects P with the outside).

Each node v also corresponds to a subpolygon $P_{\text{core}}(v) \subseteq P_v$, called the *core* of P_v, which is the union of a collection of geodesic triangles. The core $P_{\text{core}}(v)$ and the subpolygons P_u of all children u of v form a partition of P_v (thus $P_u \subseteq P_v$ for each child u). Moreover, for every child u of v, we have $|P_u| \leq |P_v|/2$. Hence, the sizes of the subpolygons P_v decrease geometrically along any root-to-leaf path in Ψ, and the height of Ψ is $O(\log n)$.

At each node v of Ψ, we explicitly store the edges of $P_{\text{core}}(v)$. The cores of all nodes are interior-disjoint and together form a partition of P. Since no new vertices are introduced and no two edges of the decomposition intersect, the overall size of the decomposition and thus the total size of all cores is $O(n)$. Because each core consists of geodesic triangles, our decomposition effectively partitions P into geodesic triangles.

As will be seen later, when our decomposition is used to answer visibility queries, the property that the gate d_v separates P_v from $P \setminus P_v$ plays a crucial role. We refer to this as the *gate property*. Moreover, for any node v and any ancestor u of v, since $P_v \subseteq P_u \subseteq P$, the gate property also implies that d_v separates P_v from $P_u \setminus P_v$.

Section 4.3 presents the detailed construction of the decomposition Ψ. The following Theorem 8 is our main result for answering visibility queries using Ψ. Our results in Sections 5 and 6 will build upon this theorem.

▶ **Theorem 8.** *A data structure of size $O(n^2 \log^4 n)$ can be constructed in $O(n^2 \log^5 n)$ time for the decomposition Ψ such that, for any query point q, if q lies in the core $P_{core}(u)$ of a node $u \in \Psi$, then $Vis(P_u, q)$ can be computed in $O(\log n)$ time; and if q lies in P_v for a child v of u, then $Vis(P_u \setminus P_v, q)$ can be computed in $O(\log n)$ time.*

Proof. We only sketch the proof here; full details can be found in the full paper.

Consider the root v^* of Ψ. For each geodesic triangle $\triangle$ in the core $P_{\text{core}}(v^*)$, we build the data structure of Lemma 7 for $\triangle$ in P_{v^*} (which equals P). We can show that this construction requires $O(n^2 \log^4 n)$ space and $O(n^2 \log^5 n)$ preprocessing time. Afterward, for any query point $q \in P$, if $q \in P_{\text{core}}(v^*)$, we can compute $\text{Vis}(P_{v^*}, q)$ in $O(\log n)$ time; and if $q \in P_v$ for a child v of v^*, we can compute $\text{Vis}(P_{v^*} \setminus P_v, q)$ in $O(\log n)$ time.

In general, for each node $v \in \Psi$, and for each geodesic triangle $\triangle$ in $P_{\text{core}}(v)$, we build the data structure of Lemma 7 for $\triangle$ within P_v. This requires $O(n_v^2 \log^4 n_v)$ space and $O(n_v^2 \log^5 n_v)$ preprocessing time, where $n_v = |P_v|$. Since the sizes of the subpolygons P_v decrease geometrically along every root-to-leaf path, and since the subpolygons P_v of nodes at the same level of Ψ are interior disjoint, the total space over all nodes of Ψ is $O(n^2 \log^4 n)$, and the total preprocessing time is $O(n^2 \log^5 n)$. After this preprocessing, the queries can be answered as stated in the theorem. ◀

4.3 The construction of the decomposition Ψ

We present the details of our decomposition and the decomposition tree Ψ.

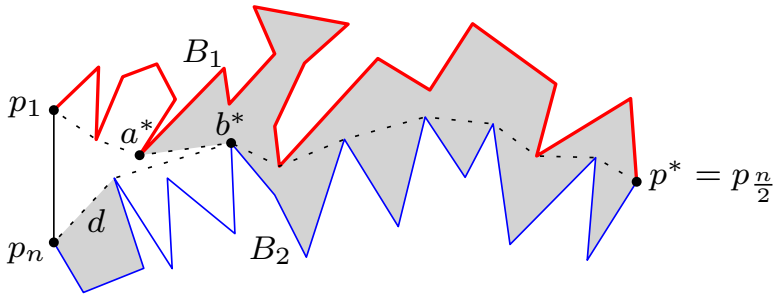

Figure 6 The shaded region bounded by d is a pocket $P_d(\triangle^*)$. $\overline{a^*b^*}$ is the bride d^* on $\partial\triangle^* \setminus d_{v^*}$. The bigger shaded region bounded by d^* is $P_{d^*}(\triangle^*)$.

During the discussion, instead of directly arguing the gate property, we will establish two other properties that together guarantee the gate property as shown in the following lemma.

▶ **Lemma 9.** *Suppose that the following two properties hold for each non-root node $v \in \Psi$: (1) d_v separates P_v from $P_u \setminus P_v$, where u is the parent of v; (2) the gate d_u of u is not in P_v. Then, the gate property holds for v, i.e., d_v separates P_v from $P \setminus P_v$.*

Proof. We prove the lemma by induction. Initially, if v is a child of the root v^*, then since d_v separates P_v from $P_u \setminus P_v$, $u = v^*$, and $P_{v^*} = P$, it is vacuously true that d_v separates P_v from $P \setminus P_v$.

We now inductively assume that d_u separates P_u from $P \setminus P_u$. We next argue that d_v separates P_v from $P \setminus P_v$. Indeed, consider two points p and p' with $p \in P_v$ and $p' \in P \setminus P_v$. It suffices to show that the geodesic path $\pi(p', p)$ must intersect d_v. If p' is in P_u, then since d_v separates P_v from $P_u \setminus P_v$, then $\pi(p', p)$ must intersect d_v. If $p' \notin P_u$, then $p' \in P \setminus P_u$. In this case, since $p \in P_v \subseteq P_u$, $d_u \not\subseteq P_v$, and d_u separates P_u from $P \setminus P_u$, if we traverse from p' to p along $\pi(p', p)$, we must first cross d_u and then d_v. The lemma thus follows. ◀

We next describe our decomposition. We will focus on explaining the children v of v^*. Specifically, we will define the core $P_{\text{core}}(v^*)$, the subpolygons P_v, and the gate d_v for every child v of v^*. In particular, we will argue that the following *three key properties* hold for each v: (1) P_v is separated from $P \setminus P_v$ by d_v; (2) d_{v^*} is not in P_v; (3) P_v has no more than $n/2$ edges other than the gate d_v. Note that the first two properties are those required in Lemma 9. To define the whole tree Ψ, we simply apply the same decomposition recursively on P_v for each child v of v^*.

Let $p^* = p_{n/2}$, the middle point of ∂P (recall that vertices of P are indexed from p_1 to p_n clockwise around ∂P). We consider the geodesic triangle $\triangle^*$ formed by the three geodesic paths $\pi(p^*, p_1)$, $\pi(p^*, p_n)$, and $\pi(p_1, p_n)$. Note that $\pi(p_1, p_n)$ is the edge $d_{v^*} = \overline{p_1 p_n}$. We first consider the general case where $\triangle^*$ is not empty, and we will discuss later that the empty case can be treated similarly. We add $\triangle^*$ to the core $P_{\text{core}}(v^*)$.

Note that d_{v^*} is one side of $\triangle^*$ and thus p_1 and p_n are two apexes of $\triangle^*$. Let b^* be the third apex (see Figure 6). Note that $\partial\triangle^* \setminus d_{v^*}$ is the union of the other two sides of $\triangle^*$.

Let B_1 denote the boundary of P from p_1 clockwise to p^* and B_2 the boundary of P from p^* clockwise to p_n. We let both B_1 and B_2 contain p^*. Hence, $\partial P = B_1 \cup B_2 \cup d_{v^*}$. We say that a diagonal is a *bridge* if one vertex of it is in $B_1 \setminus \{p^*\}$ while the other is in $B_2 \setminus \{p^*\}$. By definition, if $b^* = p^*$, then $\partial\triangle^* \setminus d_{v^*}$ has no bridge; otherwise, $\partial\triangle^* \setminus d_{v^*}$ has exactly one bridge, denoted by d^*, which must have b^* as a vertex (see Figure 6, where $d^* = \overline{a^*b^*}$).

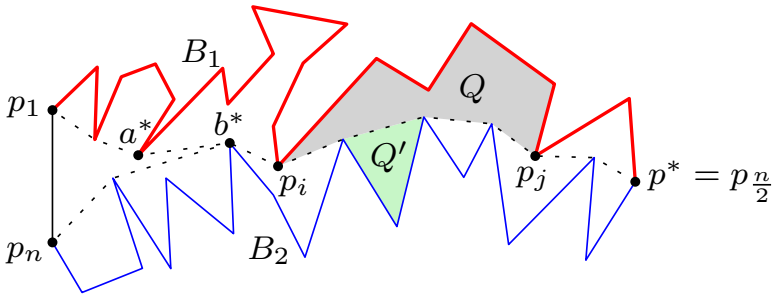

Figure 7 The gray region Q is an open B_1-bounded subpocket while the green region Q' is a closed B_2-bounded subpocket.

For each diagonal d of $\partial\triangle^* \setminus d_{v^*}$, if d is not a bridge, then it connects two vertices either both on B_1 or both on B_2. Recall we defined in Section 4.1 that the notation $P_d(\triangle^*)$ refer to the pocket of P separated by d from $\triangle^*$ (see Figure 6). Observe that the edges of $P_d(\triangle^*)$ other than d are all in B_1 or all in B_2. Hence, $P_d(\triangle^*)$ has no more than $n/2$ edges other than d. We create a child v for v^* in Ψ with $P_v = P_d(\triangle^*)$ and the gate $d_v = d$. In this way, d_v separates P_v from $P \setminus P_v$ and $d_{v^*} = \overline{p_1 p_n}$ is not an edge of P_v. Hence, all three key properties hold for v.

If d is a bridge, i.e., $d = d^*$, then $b^* \neq p^*$. In this case, $P_{d^*}(\triangle^*)$ contains vertices from both B_1 and B_2 (see Figure 6), and therefore, $P_{d^*}(\triangle^*)$ may have more than $n/2$ edges other than d^*. We therefore cannot simply create a child for v^* as above and need to resort to other approaches as follows.

Decomposing $P_{d^*}(\triangle^*)$ into subpockets. Let a^* be the vertex of d^* other than b^* (see Figure 6). Then, $\pi(p^*, a^*)$ is a subpath of either $\pi(p^*, p_1)$ or $\pi(p^*, p_n)$. To simplify the notation, let $\mathbb{Q} = P_{d^*}(\triangle^*)$ and $\pi^* = \pi(a^*, p^*)$.

The geodesic path π^* decomposes the pocket $\mathbb{Q}$ into *subpockets* each of which is bounded by a subpath of π^* and a sequence of edges of B_1 or B_2. If a subpocket is bounded by a subpath of π^* and a sequence of edges of B_1 (resp., B_2), we call it a *B_1-bounded subpocket* (resp., *B_2-bounded subpocket*); see Figure 7. For each subpocket Q, the subpath of π^* on the boundary of Q is called the *base* of Q. We say that Q is *closed* if its base is a single edge; otherwise, it is *open* (see Figure 7).

For a closed subpocket Q, since its edges other than its base edge are either all from B_1 or all from B_2, the number of its edges other than its base is at most $n/2$, and Q is separated from $P \setminus Q$ by its base edge. Hence, we create a child v for v^* in Ψ with $P_v = Q$ and d_v as the base edge of Q. Clearly, d_{v^*} is not in P_v. Hence, all three key properties hold for v.

If Q is open, then we will need to further decompose it. The situation in this case becomes significantly more complicated and our main effort is to handle this case. Without loss of generality, we assume that Q is B_1-bounded.

Decomposing an open subpocket Q. Define $Q^* = P \setminus \mathbb{Q}$. We consider Q^* a *special subpocket*. Following our above definition, Q^* is a closed subpocket since it has only one edge of π^*. To differentiate, when we say "subpockets of P", we refer to all subpockets of $\mathbb{Q}$ and also Q^*; when we say "subpockets of $\mathbb{Q}$", we only refer to the subpockets inside $\mathbb{Q}$. Unless otherwise stated, a subpocket of P refers to any subpocket of $\mathbb{Q} \cup \{Q^*\}$.

To simplify the discussion, we assume that each edge e of π^* is a diagonal. Indeed, this is the most general case because if e is an edge of P then we can assume there is an infinitely small subpocket separated by e. With this assumption, e is on the boundaries of

two subpockets of P. In particular, if p^* is a vertex of e, then one subpocket bounded by e is closed while the other is open. In this case, we consider e a bridge in the open subpocket but not a bridge in the closed one. In this way, we have the following observation.

▶ **Observation 10.** *Each open subpocket Q of $\mathbb{Q}$ has exactly two bridges, which are the first and last edges of its base (see Figure 7).*

Proof. To see this, since the base of Q is a geodesic path (because it is a subpath of π^*) and Q is a simple polygon, the base of Q must be a concave chain, meaning that if we traverse on it, either we always make right turns or we always make left turns. Hence, the two end vertices of the base of Q must be in B_1 while all other vertices are in B_2 because Q is B_1-bounded (see Figure 7). This implies that among all edges on the base of Q, only the first and last edges are bridges. For other edges of Q not on its base, by definition, each of them connects two vertices of B_1, meaning that it is not a bridge. ◀

We say that two subpockets of P are *neighboring* if they share a common edge of π^*. By definition, for two neighboring subpockets of $\mathbb{Q}$, one of them is B_1-bounded while the other is B_2-bounded. Also, a closed subpocket has exactly one neighboring subpocket while an open one has multiple. For any open subpocket Q, since only the first and last edges of its base are bridges, Q has at most two neighboring open subpockets. The following simple observation implies that to compute $\mathrm{Vis}(q)$ for a point $q \in Q$, it suffices to consider Q and its neighboring subpockets.

▶ **Observation 11.** *For any point $q \in Q$, if a point $p \in P$ is visible to q, then p is either in Q or in a neighboring subpocket of Q.*

Proof. Assume to the contrary p is in other subpockets. Then, $\overline{pq}$ must cross two edges of π^*. Since π^* is a geodesic path in P, no line segment in P can cross two edges of π^*. We thus obtain contradiction. ◀

With the above concepts, we next discuss how to further decompose an open subpocket Q (again assume that Q is B_1-bounded).

Let $\pi(Q)$ denote the base of Q. Let p_i be the end vertex of $\pi(Q)$ that is closer to a^* along $\pi^* = \pi(a^*, p^*)$ and p_j the other end vertex (see Figure 7). By definition, $i < j$. Note that if we traverse $\pi(Q)$ from p_i to p_j, we always make right turns. For simplicity of discussion, we consider the order of vertices of $\pi(Q)$ from p_i to p_j as the left-to-right order so that we can use "left" and "right" to refer to the directions of $\pi(Q)$. Note that $\pi(Q)$ could be spiral. Let $B_1(Q)$ denote the portion of B_1 in Q. Hence, p_i is the first vertex and p_j is the last vertex of $B_1(Q)$ if we order vertices of $B_1(Q)$ along B_1 clockwise (and we also consider it the left-to-right order of $B_1(Q)$). Also, the union of $B_1(Q)$ and $\pi(Q)$ forms the boundary ∂Q of Q.

We pick the middle vertex p_k of $B_1(Q)$ with $k = (i + j)/2$, and connect p_k to p_i and p_j by geodesic paths, which yields a geodesic triangle, denoted by $\triangle_k$. Since the base $\pi(Q)$ is a geodesic path and p_k is not on π, $\triangle_k$ cannot be empty. One side of $\triangle_k$ is on $\pi(Q)$ and we call it the *base side*; the other two sides are called the *non-base sides*. Among the two non-base sides, the one that connects the leftmost vertex of the base side is called the *left side* while the other is called the *right side* (recall that we have a left-to-right order of $\pi(Q)$; since the base side is a subpath of $\pi(Q)$, we can talk about the "leftmost" and "rightmost" vertices). We add $\triangle_k$ to the core $P_{\mathrm{core}}(v^*)$ of the root v^* of Ψ.

Consider the left side of $\triangle_k$, denoted by π_k^l. Let a be the leftmost vertex of the base side of $\triangle_k$. Depending on whether $a = p_i$, there are two cases.

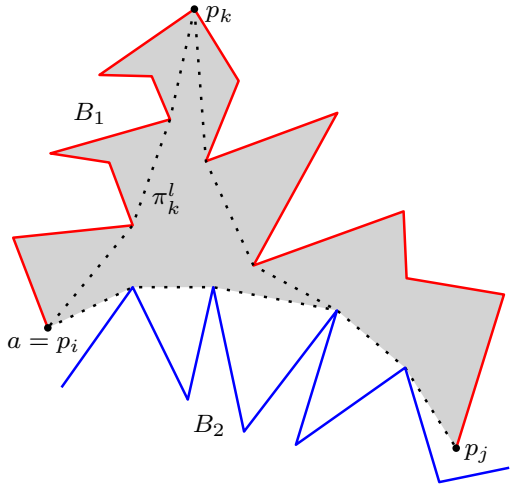

Figure 8 Illustrating the case $a = p_j$. The gray region is Q. The left dotted polygonal chain is π_k^l. The red solid polygonal chain is in B_1 and the blue solid polygonal chain is in B_2.

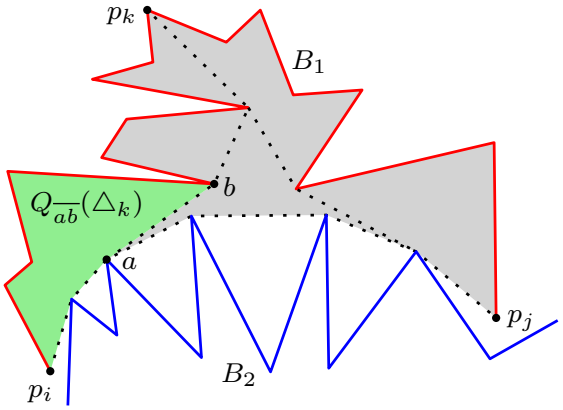

Figure 9 Illustrating the case $a \neq p_j$. The green region is $Q_{\overline{ab}}(\triangle_k)$. The union of the green region and the gray region is Q. The red solid polygonal chain is in B_1 and the blue solid polygonal chain is in B_2.

- If $a = p_i$, then no edge of π_k^l is a bridge and every edge of π_k^l connects two vertices of B_1; see Figure 8. In this case, for each diagonal d of π_k^l, all vertices of $Q_d(\triangle_k)$ are in $B_1(Q)$, where $Q_d(\triangle_k)$ is the subpolygon of Q divided by d that does not contain $\triangle_k$. Hence, $Q_d(\triangle_k)$ has no more than $n/2$ edges other than d. We create a child v for v^* in Ψ with $P_v = Q_d(\triangle_k)$ and gate $d_v = d$. Since P_v does not contain d_{v^*}, all three key properties hold for v. In this case, the processing of the left side π_k^l is done.
- If $a \neq p_i$, then a is in B_2 and the edge of π_k^l incident to a must be a bridge (see Figure 9); let b denote the other vertex of the edge. Note that none of the other edges of π_k^l is a bridge. For each diagonal d of π_k^l with $d \neq \overline{ab}$, we create a child v for v^* in the same way as above. For $\overline{ab}$, the pocket $Q_{\overline{ab}}(\triangle_k)$ now also contains vertices of B_2 (see Figure 9), and thus we cannot create a child for v^* in the same way as above. Instead, we process $Q_{\overline{ab}}(\triangle_k)$ recursively. The reason we can do this is the following. By definition, the union of $\pi(p_i, a)$, which is the portion of $\pi(Q)$ between p_i and a, and $\overline{ab}$ is a subpath of $\pi(p_i, p_k)$ and thus it is the geodesic path $\pi(p_i, b)$. In fact, $\overline{ab}$ is tangent to $\pi(Q)$ at a.

Since $Q_{\overline{ab}}(\triangle_k)$ is a simple polygon with $\pi(p_i, b)$ on its boundary, $\pi(p_i, b)$ is also a concave chain and we can consider $\pi(p_i, b)$ as the base of $Q_{\overline{ab}}(\triangle_k)$ (note that $\pi(p_i, b)$ also contains exactly two bridges at the two ends). Hence, $Q_{\overline{ab}}(\triangle_k)$ has the same "structure" as Q and thus we can apply the above processing recursively. Note that the number of vertices of B_1 in $Q_{\overline{ab}}(\triangle_k)$ is at most $|B_1(Q)|/2$.

The right side of $\triangle_k$ can be processed symmetrically.

The above process can be better represented by a tree T_Q. Each node ν of T_Q corresponds to a subpolygon $Q(\nu)$ of Q and also a non-empty geodesic triangle $\triangle_\nu$, which is added to the core $P_{\text{core}}(v^*)$ of v^* for Ψ. If ν is the root, then we have $Q(\nu) = Q$ and $\triangle_\nu = \triangle_k$. In the above processing of π_k^l, if π_k^l has a bridge $\overline{ab}$, then the root of T_Q has a left subtree by processing $Q_{\overline{ab}}(\triangle_k)$ recursively (if ν is the left child of the root, then $Q(\nu) = Q_{\overline{ab}}(\triangle_k)$). The right subtree is defined similarly. The tree height is $O(\log n)$, or more precisely $O(\log |B_1(Q)|)$ since every time we pick the middle vertex of the portion of B_1 in the current subpolygon $Q(\nu)$ to further decompose $Q(\nu)$. Note that as the base case, if $B_1(Q)$ contains a single vertex p_k other than p_i and p_j, then the two non-base sides of $\triangle_k$ are line segments $\overline{p_k p_i}$ and $\overline{p_k p_j}$, which are edges of P, and therefore the recursive step stops at Q.

Summary. The above defines the children v of v^* in the decomposition tree Ψ. As mentioned before, the entire tree Ψ can be obtained by applying the same decomposition recursively on P_v for every child v of the root v^*. Since our decomposition does not introduce any new vertices and no two edges of the geodesic paths in the decomposition cross each other, the total size of the decomposition is $O(n)$.

The above discusses the case where the geodesic triangle $\triangle^*$ is not empty. If it is empty, then it becomes a special case and can be handled similarly. Indeed, in this case, either p_n is in $\pi(p^*, p_1)$ or p_1 is in $\pi(p^*, p_n)$. We let π^* be $\pi(p^*, p_1)$ in the first case and $\pi(p^*, p_n)$ in the second case. Hence, $d_{v^*} = \overline{p_1 p_n} \subseteq \pi^*$ holds in either case. We can simply treat the entire P as $\mathbb{Q}$, and then apply the above decomposition. Since d_{v^*} is in π^*, according to our decomposition, it must lie on the base side of a geodesic triangle that is added to the core $P_{\text{core}}(v^*)$. Hence, d_{v^*} can never be in P_v for any child v of v^*. This ensures the second key property. The other two key properties follow from the same analysis.

As computing a geodesic path in P can be done in linear time [18], a straightforward algorithm can decompose each open subpocket Q in $O(|Q| \log n)$ time since the height of T_Q is $O(\log n)$. As such, decomposing all subpockets of $\mathbb{Q}$ takes $O(|\mathbb{Q}| \log n)$ time, and thus computing all the children of the root v^* of the decomposition tree Ψ can be done in $O(n \log n)$ time. As the height of Ψ is $O(\log n)$ and the subpolygons P_v of all nodes v in the same level of Ψ are interior disjoint, the total time for computing Ψ can be bounded by $O(n \log^2 n)$.

5 A data structure of optimal query time

In this section, we present our visibility query data structure of $O(n^{2+\epsilon}, n^{2+\epsilon}, \log n)$ complexity, based on the new decomposition introduced in Section 4.

Preprocessing. We first construct the decomposition tree Ψ and build the data structure of Theorem 8. This requires $O(n^2 \log^4 n)$ space and $O(n^2 \log^5 n)$ preprocessing time.

For each node $v \in \Psi$, recall that all edges of the subpolygon P_v are edges of P, except for the gate d_v. For simplicity, let $|P_v|$ denote the number of edges of P that belong to P_v.

Let r be a parameter with $1 \leq r \leq n$. A node v of Ψ is called a *border node* if $|P_{v_p}| \geq n/r$ and $|P_v| < n/r$, where v_p denotes the parent of v. By definition, each leaf-to-root path in Ψ contains at most one border node. This further implies that the subpolygons P_v corresponding to all border nodes v are edge-disjoint.

The border nodes, together with all their ancestors in Ψ, form a subtree Ψ_b of Ψ. Note that Ψ_b includes the root and has all border nodes as its leaves (hence, the border nodes indeed form the "border" of Ψ_b). We use $B(\Psi)$ to denote the set of all border nodes of Ψ. We further have the following lemma.

▶ **Lemma 12.** *The number of internal nodes of Ψ_b is $O(r)$.*

Proof. Let Ψ'_b denote the subtree of Ψ_b obtained by removing all its leaves. Our goal is to show that Ψ'_b contains $O(r)$ nodes.

By definition, $|P_v| \geq n/r$ holds for every node v of Ψ'_b. Since the edges of P_v for all leaves $v \in \Psi'_b$ are disjoint, Ψ'_b has $O(r)$ leaves. According to our decomposition, $|P_v| \leq |P_{v_p}|/2$ holds for every non-root node $v \in \Psi$.

For any value m, let V_m denote the set of nodes $v \in \Psi'_b$ satisfying $m/2 < |P_v| \leq m$. For any two nodes $u, v \in V_m$, it is impossible for one to be an ancestor of the other. Indeed, suppose to the contrary that u is an ancestor of v. Then we would have $|P_u| \leq |P_v|/2 \leq m/2$, contradicting the condition $m/2 < |P_u|$. Hence, P_u and P_v are interior disjoint, which implies that $|V_m| \leq 2n/m$.

The total number of nodes in Ψ'_b is therefore bounded by

$$|V_n| + |V_{n/2}| + |V_{n/4}| + \cdots + |V_{n/r}| \leq 2 + 4 + 8 + \cdots + 2r = O(r).$$

The lemma thus follows. ◀

We further perform the following preprocessing. For each internal node $v \in \Psi_b$, we construct a data structure $\mathcal{D}_v$ with respect to the gate d_v such that, for any point $q \in P_v$, $\mathrm{Vis}(P \setminus P_v, q)$ can be computed in $O(\log n)$ time. This corresponds to a diagonal-separated subproblem and requires $O(n^2)$ space and $O(n^2 \log n)$ preprocessing time [2], as described in Section 3.1. Since Ψ_b has $O(r)$ internal nodes by Lemma 12, constructing the data structures for all these nodes takes $O(n^2 r)$ space and $O(n^2 r \log n)$ preprocessing time in total.

For each leaf $v \in \Psi_b$, we recursively preprocess P_v and the subtree of Ψ rooted at v. This completes the preprocessing phase.

We now analyze the overall space. Let $S(n)$ denote the total space of the preprocessing (excluding the data structure of Theorem 8). We have the following recurrence:

$$S(n) = \sum_{v \in B(\Psi)} S(|P_v|) + O(n^2 r).$$

Since $|P_v| < n/r$ for every node $v \in B(\Psi)$, and the subpolygons P_v for all $v \in B(\Psi)$ are edge-disjoint (implying that $\sum_{v \in B(\Psi)} |P_v| \leq n$), by setting $r = O(n^\epsilon)$ for any constant $\epsilon > 0$, the recurrence solves to $S(n) = O(n^{2+\epsilon})$.

Similarly, the preprocessing time is also $O(n^{2+\epsilon})$. Including the data structure of Theorem 8, the total preprocessing requires $O(n^{2+\epsilon})$ space and time.

Queries. Recall that the data structure of Theorem 8 includes a point location structure for the cores $P_{\mathrm{core}}(v)$ of the nodes $v \in \Psi$, which together form a decomposition of P.

Given a query point q, we first perform a point location query to determine the node v of Ψ whose core $P_{\mathrm{core}}(v)$ contains q. Then, by Theorem 8, we compute $\mathrm{Vis}(P_v, q)$ in $O(\log n)$ time. Depending on whether v is an internal node of Ψ_b, we distinguish two cases.

- If v is an internal node of Ψ_b, then using the data structure $\mathcal{D}_v$, we compute $\mathrm{Vis}(P \setminus P_v, q)$ in $O(\log n)$ time. Finally, by Lemma 5, we merge $\mathrm{Vis}(P_v, q)$ and $\mathrm{Vis}(P \setminus P_v, q)$ in $O(\log n)$ time to obtain $\mathrm{Vis}(q)$. Thus, in this case, computing $\mathrm{Vis}(q)$ takes $O(\log n)$ time in total.
- If v is not an internal node of Ψ_b, then by following the path from v to the root of Ψ, we find a border node v'. Using the recursively constructed data structure for $P_{v'}$, we compute $\mathrm{Vis}(P_{v'}, q)$ recursively. Let u be the parent of v'. By definition, u is an internal node of Ψ_b. By Theorem 8, we compute $\mathrm{Vis}(P_u \setminus P_{v'}, q)$ in $O(\log n)$ time, and by Lemma 5, we merge $\mathrm{Vis}(P_{v'}, q)$ and $\mathrm{Vis}(P_u \setminus P_{v'}, q)$ in $O(\log n)$ time to obtain $\mathrm{Vis}(P_u, q)$. Finally, using the data structure $\mathcal{D}_u$, we compute $\mathrm{Vis}(P \setminus P_u, q)$ in $O(\log n)$ time and merge it with $\mathrm{Vis}(P_u, q)$, again in $O(\log n)$ time, to obtain $\mathrm{Vis}(q)$.
  Let $Q(n)$ denote the query time. Since $|P_{v'}| < n/r$, the recursive computation of $\mathrm{Vis}(P_{v'}, q)$ takes $Q(n/r)$ time. Hence, the recurrence relation is

$$Q(n) = Q(n/r) + O(\log n),$$

  which solves to $Q(n) = O(\log n)$ for $r = n^\epsilon$, where ϵ is a constant.

The following theorem summarizes our result.

▶ **Theorem 13.** *Given a simple polygon P with n vertices, a data structure of size $O(n^{2+\epsilon})$ can be constructed in $O(n^{2+\epsilon})$ time such that the visibility polygon $Vis(q)$ can be computed in $O(\log n)$ time for any query point $q \in P$.*

▶ Remark. One may wonder whether the balanced decomposition $\mathcal{T}(P)$ from Section 3.3 could replace our decomposition Ψ to achieve the same result. This, however, appears challenging because $\mathcal{T}(P)$ does not satisfy the gate property. Indeed, each subpolygon in $\mathcal{T}(P)$ may connect to the "outside world" through up to $O(\log n)$ diagonals. This limitation is precisely why we introduce a new decomposition for P.

6 A quadratic-space data structure

In this section, we present our data structure of complexity $O(n^2 \log n, n^2, \log n \log\log n)$. We will first give a data structure of $O(n^2 \log^5 n, n^2 \log^4 n, \log n \log\log n)$ using our decomposition Ψ. By integrating it with the framework in Section 3.3, we will reduce the space to $O(n^2)$ and reduce the preprocessing time to $O(n^2 \log n)$, while keeping the same query time.

In the preprocessing phrase, we compute the decomposition tree Ψ and the data structure of Theorem 8. This takes $O(n^2 \log^4 n)$ space and $O(n^2 \log^5 n)$ preprocessing time.

Super-levels. We partition Ψ into $O(\log\log n)$ *super-levels*. Each super-level consists of a collection of subtrees of Ψ, and we refer to each such subtree as a *component subtree* of the super-level. The i-th super-level Ψ_i, with $1 \le i \le O(\log\log n)$, is defined as follows.

We assume that the set R_i of the roots of all component subtrees of Ψ_i is already known and that the following *algorithm invariants* hold for R_i: (1) $|P_v| \le 2n/2^{2^i}$ for every node $v \in R_i$; (2) the subpolygons P_v for all $v \in R_i$ are edge-disjoint; and (3) for each node $v \in R_i$ with $i > 0$, its parent lies in the $(i-1)$-th super-level. Initially, when $i = 0$, the set R_0 consists of a single node that is the root of Ψ, and thus all invariants hold for R_0.

We now define Ψ_i. Consider a node $v \in R_i$. Let $\Psi(v)$ denote the subtree of Ψ rooted at v. We borrow several concepts from Section 5. A non-root node u of $\Psi(v)$ is called a *border node* of $\Psi(v)$ if $|P_{u_p}| \ge |P_v|/2^{2^i}$ and $|P_u| < |P_v|/2^{2^i}$, where u_p is the parent of u. The border

nodes, together with all their ancestors in $\Psi(v)$, form a subtree $\Psi_b(v)$ of $\Psi(v)$. Let $\Psi_i(v)$ denote the subtree of $\Psi_b(v)$ obtained by removing all its leaves, and let B_v denote the set of border nodes of $\Psi(v)$. We then define

$$\Psi_i = \bigcup_{v \in R_i} \Psi_i(v) \quad \text{and} \quad R_{i+1} = \bigcup_{v \in R_i} B_v.$$

We have the following observation.

▶ **Observation 14.** *All algorithm invariants hold for R_{i+1}.*

Proof. Consider a node $u \in B_v$ for some $v \in R_i$. By definition, $|P_u| \leq |P_v|/2^{2^i}$. Since $|P_v| \leq 2n/2^{2^i}$, we obtain $|P_u| \leq 2n/2^{2^{i+1}}$. This proves the first invariant.

For the second invariant, by definition, P_u of all nodes $u \in B_v$ are edge disjoint (this is because the path from any leaf to the root of $\Psi(v)$ has exactly one border node). Since P_v for all nodes $v \in R_i$ are edge disjoint, we obtain that P_u for all nodes $u \in R_{i+1}$ are edge disjoint. This proves the second invariant.

By definition, the parent of each node of R_{i+1} is in the i-th super-level Ψ_i. Therefore, the third invariant also holds. ◀

Finally, note that if $|P_v| = O(2^{2^i})$ for any $v \in R_i$, we simply set $\Psi_i(v) = \Psi(v)$, which marks the last super-level. Since $|P_v| \leq 2n/2^{2^i}$, this occurs for $i = O(\log \log n)$, implying that the total number of super-levels is $O(\log \log n)$.

Preprocessing. With the super-levels defined above, we perform the following preprocessing.

Consider the i-th super-level Ψ_i. For each node $v \in R_i$ and each node $u \in \Psi_i(v)$, note that v is an ancestor of u in Ψ. By the gate property, P_u is separated from $P_v \setminus P_u$ by the gate d_u of u. We construct a data structure $\mathcal{D}_u(v)$ for P_u with respect to d_u in P_v, such that, for any query point $q \in P_u$, $\mathrm{Vis}(P_v \setminus P_u, q)$ can be computed in $O(\log n)$ time. This is a diagonal-separated subproblem and such a data structure can be constructed in $O(|P_v|^2)$ space and $O(|P_v|^2 \log n)$ preprocessing time [2], as described in Section 3.1.

We now analyze the total space required for the data structures in Ψ_i. By the same argument as in Lemma 12, we have $|\Psi_i(v)| = O(2^{2^i})$ since $\Psi_i(v)$ consists of the internal nodes of $\Psi_b(v)$. Hence, the total space for all data structures $\mathcal{D}_u(v)$, over all nodes $u \in \Psi_i(v)$ and all $v \in R_i$, is $O(\sum_{v \in R_i} |P_v|^2 \cdot 2^{2^i})$. Because the subpolygons P_v for all $v \in R_i$ are edge-disjoint, we have $\sum_{v \in R_i} |P_v| = O(n)$. Moreover, since $|P_v| \leq 2n/2^{2^i}$, it follows that $\sum_{v \in R_i} |P_v|^2 = O(n^2/2^{2^i})$. Therefore, the total space of all data structures in the i-th super-level is $O(n^2)$, and the total preprocessing time is $O(n^2 \log n)$.

Hence, building the data structures for all $O(\log \log n)$ super-levels takes $O(n^2 \log \log n)$ space and $O(n^2 \log n \log \log n)$ preprocessing time. Including the preprocessing for Theorem 8, the total space becomes $O(n^2 \log^4 n)$ and the total preprocessing time is $O(n^2 \log^5 n)$.

Queries. Given a query point q, by a point location query, we find the node u of Ψ whose core $P_{\mathrm{core}}(u)$ contains q. We then compute $\mathrm{Vis}(P_u, q)$ in $O(\log n)$ time using Theorem 8.

Assume that u belongs to the i-th super-level for some i, i.e., $u \in \Psi_i$. Let v be the node in R_i such that $u \in \Psi_i(v)$. Using the data structure $\mathcal{D}_u(v)$, we compute $\mathrm{Vis}(P_v \setminus P_u, q)$ in $O(\log n)$ time. Since P_u is separated from $P_v \setminus P_u$ by the gate d_u, we can obtain $\mathrm{Vis}(P_v, q)$ by merging $\mathrm{Vis}(P_u, q)$ and $\mathrm{Vis}(P_v \setminus P_u, q)$ in $O(\log n)$ time using Lemma 5.

If $i = 0$, then $P_v = P$, and thus $\mathrm{Vis}(q) = \mathrm{Vis}(P_v, q)$ is already computed. Otherwise, let u' be the parent of v. By the algorithm invariants, u' lies in the $(i-1)$-th super-level. By

Theorem 8, we can compute $\mathrm{Vis}(P_{u'} \setminus P_v, q)$ in $O(\log n)$ time. Since P_v is separated from $P_{u'} \setminus P_v$ by the gate d_v, we can merge $\mathrm{Vis}(P_v, q)$ and $\mathrm{Vis}(P_{u'} \setminus P_v, q)$ in $O(\log n)$ time to obtain $\mathrm{Vis}(P_{u'}, q)$. As u' lies in the $(i-1)$-th super-level, we repeat the same procedure until reaching the 0-th super-level, after which $\mathrm{Vis}(q)$ is obtained.

Since there are $O(\log \log n)$ super-levels and each super-level requires $O(\log n)$ time, the total query time is $O(\log n \log \log n)$. For reference, we summarize the result below.

▶ **Lemma 15.** *We can construct a data structure of size $O(n^2 \log^4 n)$ in $O(n^2 \log^5 n)$ time for P such that $Vis(q)$ can be computed in $O(\log n \log \log n)$ time for any query point $q \in P$.*

We further reduce the preprocessing complexities in the following theorem.

▶ **Theorem 16.** *Given a simple polygon P with n vertices, a data structure of size $O(n^2)$ can be constructed in $O(n^2 \log n)$ time such that the visibility polygon $Vis(q)$ can be computed in $O(\log n \log \log n)$ time for any query point $q \in P$.*

Proof. We follow the framework of Section 3.3 to construct the balanced decomposition $\mathcal{T}(P)$ and solve the diagonal-separated subproblems using the quadratic-space method of [2], as reviewed in Section 3.1. This requires $O(n^2)$ space and $O(n^2 \log n)$ preprocessing time.

Let $\mathcal{T}_t(P)$ denote the subtree of $\mathcal{T}(P)$ consisting of its top t levels, where t is the smallest integer such that the subpolygon P_v at every node v of the t-th level has size at most $n/\log^5 n$. By the definition of $\mathcal{T}(P)$, we have $t = O(\log \log n)$.

For each leaf $v \in \mathcal{T}_t(P)$, we construct a data structure $\mathcal{D}_v$ as in Lemma 15 for P_v. Each such data structure requires $O(|P_v|^2 \log^4 n)$ space and $O(|P_v|^2 \log^5 n)$ preprocessing time. Hence, the total preprocessing time for all data structures $\mathcal{D}_v$ of the leaves $v \in \mathcal{T}_t(P)$ is $O(\sum_{v \in L} |P_v|^2 \log^5 n)$, where L denotes the set of leaves of $\mathcal{T}_t(P)$. Since the subpolygons P_v for all leaves $v \in L$ form a partition of P, we have $\sum_{v \in L} |P_v| = O(n)$. Furthermore, because $|P_v| \le n/\log^5 n$ for all $v \in L$, it follows that $\sum_{v \in L} |P_v|^2 = O(n^2/\log^5 n)$. Thus, the total preprocessing time of all $\mathcal{D}_v$ is $O(n^2)$, and the total space is also $O(n^2)$.

This completes the preprocessing, which requires $O(n^2)$ space and $O(n^2 \log n)$ time in total.

For a query point $q \in P$, we first locate the leaf node $v \in L$ such that P_v contains q, which takes $O(\log n)$ time using a point location query. Then, by Lemma 15, we compute $\mathrm{Vis}(P_v, q)$ in $O(\log n \log \log n)$ time using $\mathcal{D}_v$. Next, as described in Section 3.3, we use the diagonal-separated data structure to compute $\mathrm{Vis}(P_u, q)$ in $O(\log n)$ time for the parent u of v. In general, by following the path from v to the root in $\mathcal{T}(P)$, $\mathrm{Vis}(q)$ can be computed in an additional $O(\log n \log \log n)$ time since the path has $O(\log \log n)$ nodes. Therefore, the total query time for computing $\mathrm{Vis}(q)$ is $O(\log n \log \log n)$. ◀

References

1 Pankaj K. Agarwal and Micha Sharir. Ray shooting amidst convex polygons in 2D. *Journal of Algorithms*, 21(3):508–519, 1996. doi:10.1006/jagm.1996.0056.

2 Boris Aronov, Leonidas J. Guibas, Marek Teichmann, and Li Zhang. Visibility queries and maintenance in simple polygons. *Discrete and Computational Geometry*, 27(4):461–483, 2002. doi:10.1007/s00454-001-0089-9.

3 Prosenjit Bose, Anna Lubiw, and J. Ian Munro. Efficient visibility queries in simple polygons. *Computational Geometry: Theory and Applications*, 23(3):313–335, 2002. doi:10.1016/S0925-7721(01)00070-0.

4 Mojtaba Nouri Bygi and Mohammad Ghodsi. Weak visibility queries in simple polygons. In *Proceedings of the 23rd Canadian Conference on Computational Geometry (CCCG)*, 2011. URL: https://www.cccg.ca/proceedings/2011/papers/paper95.pdf.

5 Timothy M. Chan and Da Wei Zheng. Simplex range searching revisited: How to shave logs in multi-level data structures. In *Proceedings of the 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1493–1511, 2023. doi:10.1137/1.9781611977554.ch54.

6 Bernard Chazelle. A theorem on polygon cutting with applications. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 339–349, 1982. doi:10.1109/SFCS.1982.58.

7 Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete and Computational Geometry*, 6:485–524, 1991. doi:10.1007/BF02574703.

8 Bernard Chazelle. Cutting hyperplanes for divide-and-conquer. *Discrete and Computational Geometry*, 9:145–158, 1993. doi:10.1007/BF02189314.

9 Bernard Chazelle, Herbert Edelsbrunner, Michelangelo Grigni, Leonidas J. Guibas, John Hershberger, Micha Sharir, and Jack Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12:54–68, 1994. doi:10.1007/BF01377183.

10 Bernard Chazelle and Leonidas J. Guibas. Visibility and intersection problems in plane geometry. *Discrete and Computational Geometry*, 4:551–589, 1989. doi:10.1007/BF02187747.

11 Danny Z. Chen and Ovidiu Daescu. Maintaining visibility of a polygon with a moving point of view. *Information Processing Letters*, 65:269–275, 1996. doi:10.1016/S0020-0190(97)00211-1.

12 Danny Z. Chen and Haitao Wang. Visibility and ray shooting queries in polygonal domains. *Computational Geometry: Theory and Applications*, 48:31–41, 2015. doi:10.1016/j.comgeo.2014.08.003.

13 Danny Z. Chen and Haitao Wang. Weak visibility queries of line segments in simple polygons. *Computational Geometry: Theory and Applications*, 48:443–452, 2015. doi:10.1016/j.comgeo.2015.02.001.

14 James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

15 Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986. doi:10.1137/0215023.

16 Hossam A. ElGindy and David Avis. A linear algorithm for computing the visibility polygon from a point. *Journal of Algorithms*, 2(2):186–197, 1981. doi:10.1016/0196-6774(81)90019-5.

17 Leonidas J. Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, 1989. doi:10.1016/0022-0000(89)90041-X.

18 Leonidas J. Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987. doi:10.1007/BF01840360.

19 Leonidas J. Guibas, Rajeev Motwani, and Prabhakar Raghavan. The robot localization problem in two dimensions. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 259–268, 1992. doi:10.5555/139404.139460.

20 John Hershberger and Subhash Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18:403–431, 1995. doi:10.1006/jagm.1995.1017.

21 Rajasekhar Inkulu and Sanjiv Kapoor. Visibility queries in a polygonal region. *Computational Geometry: Theory and Applications*, 42(9):852–864, 2009. doi:10.1016/j.comgeo.2009.02.004.

22 David G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983. doi:10.1137/0212002.

23 Lin Lu, Chenglei Yang, and Jiaye Wang. Point visibility computing in polygons with holes. *Journal of Information and Computational Science*, 8(16):4165–4173, 2011. URL: https://www.researchgate.net/publication/282763094_Point_visibility_computing_in_polygons_with_holes.

24 Jiří Matoušek. Efficient partition trees. *Discrete and Computational Geometry*, 8(3):315–334, 1992. doi:10.1007/BF02293051.

25 Michel Pocchiola. Graphics in flatland revisited. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 85–96, 1990. doi:10.1007/3-540-52846-6_80.

26 Neil Snarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986. doi:10.1145/6138.6151.

27 Alireza Zarei and Mohammad Ghodsi. Query point visibility computation in polygons with holes. *Computational Geometry: Theory and Applications*, 39(2):78–90, 2008. doi:10.1016/j.comgeo.2007.02.005.