Segment Visibility Counting Queries in Polygons

Kevin Buchin
Department of Computer Science, TU Dortmund, Germany

Bram Custers
Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands

Ivor van der Hoog
Department of Applied Mathematics and Computer Science, DTU, Copenhagen, Denmark

Maarten Löffler
Department of Information and Computing Sciences, Utrecht University, The Netherlands

Aleksandr Popov
Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands

Marcel Roeloffzen
Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands

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

Abstract

Let P be a simple polygon with n vertices, and let A be a set of m points or line segments inside P. We develop data structures that can efficiently count the objects from A that are visible to a query point or a query segment. Our main aim is to obtain fast, $\mathcal{O}(\text{polylog}\, nm)$, query times, while using as little space as possible.

In case the query is a single point, a simple visibility-polygon-based solution achieves $\mathcal{O}(\log nm)$ query time using $\mathcal{O}(nm^2)$ space. In case A also contains only points, we present a smaller, $\mathcal{O}(n + m^{2+\varepsilon} \log n)$-space, data structure based on a hierarchical decomposition of the polygon.

Building on these results, we tackle the case where the query is a line segment and A contains only points. The main complication here is that the segment may intersect multiple regions of the polygon decomposition, and that a point may see multiple such pieces. Despite these issues, we show how to achieve $\mathcal{O}(\log n \log nm)$ query time using only $\mathcal{O}(nm^{2+\varepsilon} + n^2)$ space. Finally, we show that we can even handle the case where the objects in A are segments with the same bounds.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Visibility, Data Structure, Polygons, Complexity

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.58

Related Version *Full Version*: `https://arxiv.org/abs/2201.03490` [7]

Funding *Bram Custers*: Supported by the Dutch Research Council (NWO) under the project number 628.011.005.
Ivor van der Hoog: Supported by the Dutch Research Council (NWO) under the project number 614.001.504.
Maarten Löffler: Partially supported by the Dutch Research Council (NWO) under the project numbers 614.001.504 and 628.011.005.
Aleksandr Popov: Supported by the Dutch Research Council (NWO) under the project number 612.001.801.
Marcel Roeloffzen: Supported by the Dutch Research Council (NWO) under the project number 628.011.005.

1 Introduction

Let P be a simple polygon with n vertices, and let A be a set of m points or line segments inside P. We develop efficient data structures for *visibility counting queries* in which we wish to report the number of objects from A visible to some (constant-complexity) query object Q. An object X in A is *visible* from Q if there is a line segment connecting X and Q contained in P; other objects in A do not block visibility. We focus on the case when Q is a point or a line segment. We aim to obtain fast, $\mathcal{O}(\operatorname{polylog} nm)$, query times, using as little space as possible. Our work is motivated by problems in movement analysis where we have sets of moving entities, for example, an animal species and their predators, and we wish to determine if there is mutual visibility between the entities of different sets. We also want to quantify "how much" the sets can see each other. Given measurements at certain points in time, solving the mutual visibility problem between two such times reduces to counting visibility between line segments (for moving entities) and points (for static objects). This visibility counting problem is also of general interest.

Related work. Computing visibility in polygons is a classical problem in computational geometry [19, 30]. Algorithms for efficiently testing visibility between a pair of points, for computing visibility polygons [18, 27, 29], and for constructing visibility graphs [31] have been a topic of study for over thirty years. There is even a host of work on computing visibility on terrains and in other three-dimensional environments [20, Chapter 33.8]. For many of these problems, the data structure version of the problem has also been considered. In these versions, the polygon is given up front, and the task is to store it so that we can efficiently query whether a pair of points p, q is mutually visible [13, 23, 26], or report the visibility polygon $V(q)$ of q [4]. In particular, when P is a simple polygon with n vertices, the former type of queries can be answered optimally – in $\mathcal{O}(\log n)$ time using linear space [26]. The latter type of queries can be answered in $\mathcal{O}(\log^2 n + |V(q)|)$ time using $\mathcal{O}(n^2)$ space [4]. The visibility polygon itself has complexity $\mathcal{O}(n)$ [18]. Visibility polygons inherit structure from the boundaries of P, so the approaches that use it do not transfer to our setting.

Computing the visibility polygon of a line segment has been considered, as well. When the polygon modelling the environment is simple, the visibility polygon, called a *weak visibility polygon,* denoted $V(\overline{pq})$ for a line segment $\overline{pq}$, still has linear complexity, and can be computed in $\mathcal{O}(n)$ time [23]. Chen and Wang [15] consider the data structure version of the problem: they describe a linear-space data structure that can be queried in $\mathcal{O}(|V(\overline{pq})| \log n)$ time, and an $\mathcal{O}(n^3)$-space data structure that can be queried in $\mathcal{O}(\log n + |V(\overline{pq})|)$ time.

Computing the visibility polygon of a line segment $\overline{pq}$ allows us to answer whether an entity moving along $\overline{pq}$ can see a particular fixed point r, i.e. there is a time at which the moving entity can see r if and only if r lies inside $V(\overline{pq})$. If the point r may also move, it is not necessarily true that the entity can see r if the trajectory of r intersects $V(\overline{pq})$. Eades et al. [17] present data structures that can answer such queries efficiently. In particular, they present data structures of size $\mathcal{O}(n \log^5 n)$ that can answer such a query in time $\mathcal{O}(n^{3/4} \log^3 n)$. They present results even in case the polygon has holes. Aronov et al. [4] show that we can also efficiently maintain the visibility polygon of an entity as it is moving.

Visibility counting queries have been studied before, as well. Bose et al. [6] studied the case where, for a simple polygon and a query point, the number of visible polygon edges is reported. The same problem has been considered for weak visibility from a query segment [8]. For the case of a set of disjoint line segments and a query point, approximation algorithms

Table 1 Results in this paper. ● and / denote points and line segments, respectively.

A	Q	Data structure Space	Preprocessing	Query	Section
●	●	$\mathcal{O}(nm^2)$	$\mathcal{O}(nm \log n + nm^2)$	$\mathcal{O}(\log nm)$	3.1
●	●	$\mathcal{O}(n+m^{2+\varepsilon} \log n)$	$\mathcal{O}(n + m \log^2 n + m^{2+\varepsilon} \log n)$	$\mathcal{O}(\log n \log nm)$	3.2
/	●	$\mathcal{O}(nm^2)$	$\mathcal{O}(nm \log n + nm^2)$	$\mathcal{O}(\log nm)$	3.1
●	/	$\mathcal{O}(n^2 + nm^{2+\varepsilon})$	$\mathcal{O}(n^2 \log m + nm^{2+\varepsilon})$	$\mathcal{O}(\log n \log nm)$	4
/	/	$\mathcal{O}(n^2 + nm^{2+\varepsilon})$	$\mathcal{O}(n^2 \log m + nm^{2+\varepsilon})$	$\mathcal{O}(\log n \log nm)$	5

exist [3, 21, 32]. In contrast to these settings, we wish to count visible line segments with visibility obstructed by a simple polygon (other than the line segments). Closer to our setting is the problem of reporting all pairs of visible points in a simple polygon [5].

Results and organisation. Our goal is to efficiently count the objects, in particular, line segments or points, in a set A that are visible to a query object Q. We denote this count by $C(Q, A)$. Given P, A, and Q, we can easily compute $C(Q, A)$ in optimal $\mathcal{O}(n+m \log n)$ time (see Lemma 1). We are mostly interested in the data structure version of the problem, in which we are given the polygon P and the set A in advance, and we wish to compute $C(Q, A)$ efficiently once we are given the query object Q. We show that we can indeed answer such queries efficiently, that is, in polylogarithmic time in n and m. The exact query times and the space usage and preprocessing times depend on the type of the query object and the type of objects in A. See Table 1 for an overview. Here and in the rest of the paper, $\varepsilon > 0$ denotes an arbitrarily small constant.

In Section 3, we consider the case where the query object is a point. We show how to answer queries efficiently using the arrangement of all (weak) visibility polygons. As Bose et al. [6, Section 6.2] argued, such an arrangement has complexity $\Theta(nm^2)$ in the worst case. We then show that if the objects in A are points, we can do significantly better. We argue that we do not need to construct the visibility polygons of all points in A, avoiding an $\mathcal{O}(nm)$ term in the space and preprocessing time. We use a hierarchical decomposition of the polygon and the fact that the visibility of a point $a \in A$ in a subpolygon into another subpolygon is described by a single constant-complexity cone. Aronov et al. [4] also use hierarchical decomposition, but the rest of their approach cannot be efficiently used in our setting, since it uses the cyclic ordering of the vertices of a visibility polygon.

In Section 4, we turn our attention to the case where the query object is a line segment $\overline{pq}$ and the objects in A are points. One possible solution in this scenario would be to store the visibility polygons for the points in A so that we can count such polygons stabbed by the query segment. However, since these visibility polygons have total complexity $\mathcal{O}(nm)$ and the query may have an arbitrary orientation, a solution achieving polylogarithmic query time will likely use at least $\Omega(n^2m^2)$ space [1, 2, 24]. So, we again use an approach that hierarchically decomposes the polygon to limit the space usage. Unfortunately, testing visibility between the points in A and the query segment is more complicated in this case. Moreover, the segment can intersect multiple regions of the decomposition, so we have to avoid double counting. All of this makes the problem significantly harder. We manage to overcome these difficulties using careful geometric arguments and an inclusion–exclusion-style counting scheme. Our result in Table 1 saves at least a linear factor compared to an approach based on stabbing visibility polygons. We then show that we can extend these arguments even further and solve the scenario where the objects in A are also line segments. Surprisingly, this does not impact the space or time complexity of the data structure.

Finally, in Section 5, we discuss some extensions of our results. In particular, we show that just testing if the count $C(Q, A)$ is non-zero (i.e. if Q is visible from any of the objects) is easier, and that we can compute the pairwise visibility of two sets of objects – that is, solve the problem that motivated this work – in time subquadratic in the number of objects.

2 Preliminaries

In this section, we briefly review some basic tools we use to build our data structures. We omit some common material and proofs; we refer the reader to the full version for those.

Visibility in a simple polygon. Denote the (weak) visibility polygon in a simple polygon P of a point p (resp. line segment $\overline{pq}$) by $V(p)$ (resp. $V(\overline{pq})$). A *cone* is a subset of the plane that is enclosed by two rays starting at some point p, called the *apex* of the cone; the angle between any two rays in the cone is acute. We refer to the two bounding rays as the *left* and the *right* ray, so that moving clockwise from the left to the right ray traverses the cone. A *subcone* of some cone C is a cone with the same apex as C that is a subset of C. For a segment $\overline{rs} \subset P$, define the *visibility cone* of a point $p \in P$ *through* $\overline{rs}$, denoted $V(p, \overline{rs})$, as a region consisting of the rays from p that intersect $\overline{rs}$ before properly crossing the boundary of P.[1] Define a visibility cone $C(p)$ *into* a subpolygon U for $p \in P \setminus U$ as the visibility cone through the diagonal of P that separates U from the subpolygon containing p.

▶ **Lemma 1.** *Let P be a simple polygon with n vertices, and let A be a set of m points or line segments in P. For a point or a line segment Q, we can find $C(Q, A)$ in time $\mathcal{O}(n + m \log n)$.*

Proof. If A is a set of points, it suffices to compute the visibility polygon of Q and preprocess it for $\mathcal{O}(\log n)$-time point location queries. Both preprocessing steps take linear time [23, 28], and querying takes $\mathcal{O}(m \log n)$ time in total. In case A consists of line segments, we can similarly test if one of the endpoints of each segment of A is visible, thus making the segment visible. We also need to count the number of visible segments whose endpoints lie outside $V(Q)$. This can be done in $\mathcal{O}(n + m \log n)$ time by computing a sufficiently large bounding box B of $V(Q)$, and constructing an $\mathcal{O}(\log n)$-time ray shooting data structure on $B \setminus V(Q)$. This allows us to test if a segment intersects $V(Q)$ in $\mathcal{O}(\log n)$ time. The polygon $B \setminus V(Q)$ has only a single hole, so we can connect the boundary of $V(Q)$ to the boundary of B with a line segment $\overline{rs}$ and cut $B \setminus V(Q)$ along $\overline{rs}$ to obtain a simple polygon. We can then build a ray shooting structure [26] on this simple polygon, and answer a query by $\mathcal{O}(1)$ ray shooting queries. In particular, for any segment in A that does not cross $\overline{rs}$, we get the result directly; and for any segment that crosses $\overline{rs}$, we detect that the ray hits $\overline{rs}$ and do a second query on the other side of the cut. Either way, we use $\mathcal{O}(1)$ ray shooting queries. ◀

▶ **Lemma 2.** *Given a visibility polygon $V(p) \subseteq P$ for some point $p \in P$ and a line segment $\overline{rs} \subset P$, either $\overline{rs}$ and $V(p)$ do not intersect, or their intersection is a line segment.*

▶ **Corollary 3.** *The intersection between the line segment $\overline{rs} \subset P$ and the visibility cone $V(p, \overline{rs})$ for some $p \in P$ is either empty, or a line segment.*

[1] In case $p \in \overline{rs}$ holds, this definition yields $\mathbb{R}^2$. We handle such cases separately.

Cutting trees. A *cutting tree* [11, 14, 16] is a data structure commonly used for efficient half-plane range queries. Nesting multiple cutting trees in levels allows to efficiently perform simplex range searching and solve other related queries; we make extensive use of this. See Figure 1 for an illustration. We now discuss this data structure in more detail.

Suppose we want to preprocess a set $\mathcal{L}$ of m lines in the plane so that given a query point q, we can count the number of lines below the query point. Let $r \in [1, m]$ be a parameter; then a $({}^{1}\!/\!{}_{r})$-*cutting* of $\mathcal{L}$ is a subdivision of the plane with the property that each cell is intersected by at most ${}^{m}\!/\!{}_{r}$ lines [11]. If q lies in a certain cell of the cutting, we know, for all lines that do not cross the cell, whether they are above or below q, and so we can store the count with the cell, or report the lines in a precomputed *canonical subset;* for the lines that cross the cell, we can recurse. The data structure that performs such a query is called a *cutting tree;* it can be constructed in $\mathcal{O}(m^{2+\varepsilon})$ time, uses $\mathcal{O}(m^{2+\varepsilon})$ space, and supports answering the queries in time $\mathcal{O}(\log m)$, for any constant $\varepsilon > 0$. Intuitively, the parameter r here determines the trade-off between the height of the recursion tree and the number of nodes for which a certain line in $\mathcal{L}$ is relevant. If we pick $r = m$, the $({}^{1}\!/\!{}_{r})$-cutting of $\mathcal{L}$ is just the arrangement of $\mathcal{L}$. The bounds above are based on picking $r \in \mathcal{O}(1)$, so the height of the recursion tree is $\mathcal{O}(\log m)$. This approach follows the work of Clarkson [16], with Chazelle [11] obtaining the bounds above by improving the cutting construction.

An obvious benefit of this approach over just constructing the arrangement on $\mathcal{L}$ and doing point location in that arrangement is that using cuttings, we can obtain $\mathcal{O}(\log m)$ canonical subsets and perform nested queries on them without an explosion in storage required; the resulting data structure is called a *multilevel cutting tree.* Specifically, we can query with k points and a direction associated with each point (above or below) and return the lines of $\mathcal{L}$ that pass on the correct side (above or below) of all k query points. If we pick $r \in \mathcal{O}(1)$ and nest k levels in a k-level cutting tree, we get the same construction time and storage bounds as for a regular cutting tree; but the query time is now $\mathcal{O}(\log^k m)$. Chazelle et al. [14] show that if we set $r = n^{\varepsilon/2}$, each level of a multilevel cutting tree is a constant-height tree, so the answer to the query can be represented using only $\mathcal{O}(1)$ canonical subsets and the query time is reduced to $\mathcal{O}(\log m)$. The space used and the preprocessing time remains $\mathcal{O}(m^{2+\varepsilon})$.

▶ **Lemma 4** ([14]). *Let $\mathcal{L}$ be a set of m lines and let k be a constant. Suppose we want to answer the following query: given k points and associated directions (above or below), find the lines in $\mathcal{L}$ that lie on the correct side of all k points. In time $\mathcal{O}(m^{2+\varepsilon})$, we can construct a data structure using $\mathcal{O}(m^{2+\varepsilon})$ storage that supports such queries. The lines are returned as $\mathcal{O}(1)$ canonical subsets, and the query time is $\mathcal{O}(\log m)$.*

Dualising the problem in the usual way, we can alternatively report or count points from the set A that lie in a query half-plane; or in the intersection of several half-planes, using a multilevel cutting tree.

▶ **Lemma 5** ([14]). *Let A be a set of m points and let k be a constant. In time $\mathcal{O}(m^{2+\varepsilon})$, we can construct a data structure using $\mathcal{O}(m^{2+\varepsilon})$ storage that returns $\mathcal{O}(1)$ canonical subsets with the points in A that lie in the intersection of the k query half-planes in time $\mathcal{O}(\log m)$.*

▶ **Lemma 6.** *Let A be a set of m arbitrary points in P, with each $a \in A$ an apex of some cone C_a. At query time, we get the point q, an apex of a cone C_q. In time $\mathcal{O}(m^{2+\varepsilon})$, we can construct a data structure using $\mathcal{O}(m^{2+\varepsilon})$ space that returns a representation of the points in $A' \subseteq A$, so that for any $p \in A'$, we have $q \in C_p$ and $p \in C_q$. The points are returned as $\mathcal{O}(1)$ canonical subsets, and the query time is $\mathcal{O}(\log m)$; they can be counted in the same time.*

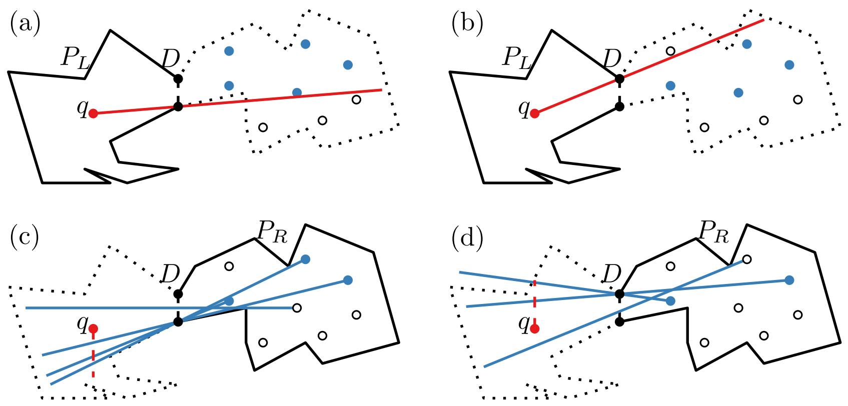

Figure 1 A query in a multilevel cutting tree, top left to bottom right. The query point is red; the selected points of A are blue. Black outline shows the relevant part of the polygon. (a, b) We select points in A above (resp. below) the right (resp. left) cone boundary of q. (c, d) We refine by taking points whose left (resp. right) cone boundary is below (resp. above) q.

▶ **Lemma 7.** *Let L be a vertical line and let A be a set of m cones starting left of L and whose left and right rays intersect L. In time $\mathcal{O}(m^{2+\varepsilon})$, we can construct two two-level cutting trees for A of total size $\mathcal{O}(m^{2+\varepsilon})$, so that for a query segment $\overline{pq}$ that is fully to the right of L, we can count the cones that contain or intersect $\overline{pq}$ in $\mathcal{O}(\log m)$ time.*

▶ **Lemma 8.** *Let $\mathcal{L}$ be a set of m lines and $\overline{pq}$ a query line segment. We can store $\mathcal{L}$ in a multilevel cutting tree, using $\mathcal{O}(m^{2+\varepsilon})$ space and preprocessing time, so that we can count the lines in $\mathcal{L}$ intersected by $\overline{pq}$ in time $\mathcal{O}(\log m)$.*

Polygon decomposition. For a simple polygon P on n vertices, Chazelle [9] shows that we can construct a balanced hierarchical decomposition of P by recursively splitting the polygon into two subpolygons of roughly equal size, using only diagonals (segments between two vertices of the polygon). The recursion stops when reaching triangles. The decomposition can be computed in $\mathcal{O}(n)$ time and stored using $\mathcal{O}(n)$ space in a balanced binary tree [9, 10, 22].

Hourglasses and the shortest path data structure. An *hourglass* for two diagonals $\overline{pq}$ and $\overline{rs}$ in a simple polygon P is the union of geodesic shortest paths in P from points on $\overline{pq}$ to points on $\overline{rs}$ [22]. Such an hourglass is bounded by two diagonals and two inward convex chains. Call one of the diagonals the *left* diagonal and the other the *right* diagonal. By following the hourglass boundary in a clockwise manner, starting from the left diagonal, we visit the *upper convex chain,* the right diagonal, and the *lower convex chain.* If the upper chain and lower chain of an hourglass share vertices, it is *closed,* otherwise it is *open.* We only explicitly use open hourglasses in our constructions.[2] A *visibility glass* is a subset of the hourglass, restricted to the line segments between points on $\overline{pq}$ and points on $\overline{rs}$ [17].

Guibas and Hershberger [22, 25] describe a data structure to compute shortest paths in a simple polygon P. They use the polygon decomposition by Chazelle [9] and also store hourglasses between the splitting diagonals of the decomposition. The data structure uses $\mathcal{O}(n)$ storage and preprocessing time and can answer the following queries in $\mathcal{O}(\log n)$ time:

[2] If an hourglass is closed, the two chains are only inward convex from the endpoints of a diagonal until they meet.

Segment location query. Given a segment $\overline{pq}$, return the two leaf triangles containing p and q in the decomposition and the $\mathcal{O}(\log n)$ pairwise disjoint open hourglasses so that the triangles and hourglasses fully cover $\overline{pq}$. We call this structure the *polygon cover* of $\overline{pq}$.

Shortest path query. Given points $p, q \in P$, return the geodesic shortest path between p and q in P as a set of $\mathcal{O}(\log n)$ nodes of the decomposition. The shortest path between p and q is a concatenation of subcurves of the polygonal chains of the appropriate boundaries of the (open or closed) hourglasses in these $\mathcal{O}(\log n)$ nodes together with at most $\mathcal{O}(\log n)$ segments connecting two consecutive subcurves.

Cone query. Given a point s and a line segment $\overline{pq}$ in P, return $V(s, \overline{pq})$. This can be done by computing the shortest paths from s to p and to q and taking the first segments of each path starting at s to extend them into the bounding rays of a cone.

3 Point Queries

In this section, given a set A of m points in a simple polygon P on n vertices, we count the points of A that are in the visibility polygon of a query point $q \in P$. We present two solutions: (i) an arrangement-based approach that also applies in the case where A contains line segments, which achieves low query time at the cost of large storage and preprocessing time; and (ii) a cutting-tree-based approach with query times slower by a factor of $\mathcal{O}(\log n)$, but with much better storage requirements and preprocessing time.

3.1 Point Location in an Arrangement

The approach relies on the fact that the number of objects in A visible to a query point q is equal to the number of (weak) visibility polygons of the objects in A stabbed by q. We construct all (weak) visibility polygons of the objects in A and compute the arrangement $\mathcal{A}$ of the edges of these polygons. For each cell C in the arrangement, we store the number of visibility polygons that contain C. Then a point location query for q yields the number of visible objects in A. Computing the visibility polygons takes $\mathcal{O}(nm)$ time, and constructing the arrangement using an output-sensitive line segment intersection algorithm takes $\mathcal{O}(nm \log nm + |\mathcal{A}|)$ time [12], where $|\mathcal{A}|$ is the number of vertices of $\mathcal{A}$. Building a point location structure on $\mathcal{A}$ for $\mathcal{O}(\log|\mathcal{A}|)$-time point location queries takes $\mathcal{O}(|\mathcal{A}|)$ time [28]; the space used is $\mathcal{O}(|\mathcal{A}|)$. As Bose et al. [6] show, the worst-case complexity of $\mathcal{A}$ is $\Theta(nm^2)$.

▶ **Theorem 9.** *Let P be a simple polygon with n vertices, and let A be a set of m points or line segments in P. In $\mathcal{O}(nm^2 + nm \log nm)$ time, we can build a data structure of size $\mathcal{O}(nm^2)$ that can report the number of points or segments in A visible from a query point q in $\mathcal{O}(\log nm)$ time.*

3.2 Hierarchical Decomposition

To design a data structure that uses less storage than that of Section 3.1, we observe that if we subdivide the polygon, we can count the visible objects by summing up the number of visible objects in the cells of the subdivision. To efficiently compute these counts, we use the polygon decomposition approach (see Section 2). With each split in our decomposition, we store data structures that can efficiently count the visible objects in the associated subpolygon.

Cone containment. Let us solve the following problem first. We are given a simple polygon P and a (w.l.o.g.) vertical diagonal D that splits it into two simple polygons P_L and P_R. Furthermore, we are given a set A of m points in P_L. Given a query point q in P_R, we want to count the points in A that see q. We base our approach on the following observation.

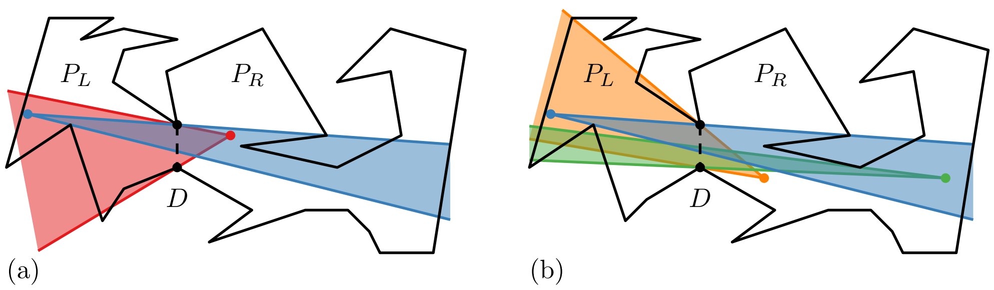

Figure 2 Visibility cones (coloured regions) of (coloured) points w.r.t. some diagonal D. (a) Blue and red are mutually visible. (b) Green and blue cannot see each other, nor can orange and blue.

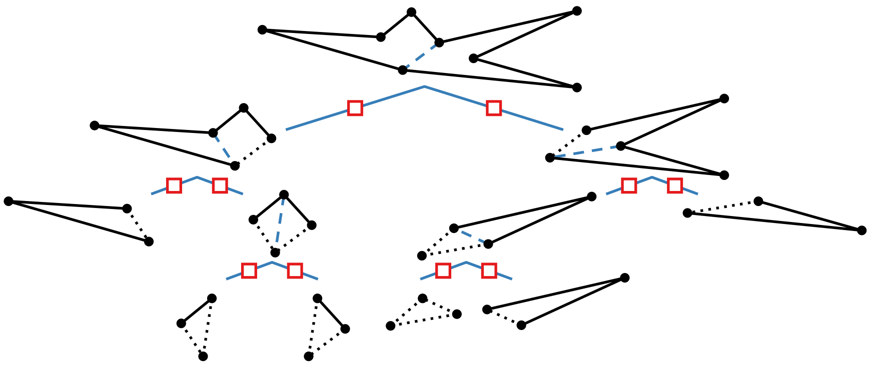

Figure 3 Augmented polygon decomposition following the approach by Chazelle [9]. Each node corresponds to the splitting diagonal (blue dashed line). Along the tree edges (blue lines), we store the multilevel cutting tree (red box) for the polygon in the child using the diagonal of the parent.

▶ **Lemma 10.** *Given a simple polygon P, split into two simple polygons P_L and P_R by a diagonal D between two vertices; and given two points $p \in P_L$ and $q \in P_R$, consider the visibility cones $V(p, D)$ and $V(q, D)$, i.e. the cones from p and q through D into the other subpolygons. Point p sees q in P if and only if $q \in V(p, D)$ and $p \in V(q, D)$.*

Lemma 10 shows that to count the points in A that see q, it suffices to construct the cones from all points in A through D and the cone from q through D and count the points in A satisfying the condition of Lemma 10 (see Figure 2). The cones $V(p, D)$ from all $p \in A$ can be precomputed (we shall handle this later), so only the cone $V(q, D)$ needs to be computed at query time. The query of this type can be realised using a multilevel cutting tree (Lemma 6).

Decomposition. Let us return to the original problem. To solve it, we can use the balanced polygon decomposition [9] (see Section 2). Following Guibas and Hershberger [22, 25], we represent it as a binary tree (see Figure 3). Observe that as long as there is some diagonal D separating our query point from a subset of points of A, we can use the approach above.

Every node of the tree is associated with a diagonal, and the two children correspond to the left and the right subpolygons. With each node, we store two data structures described above: one for the query point to the left of the diagonal and one for the query to the right.

The query then proceeds as follows. Suppose the polygon P is triangulated, and the triangles correspond to the leaves in the decomposition. Given a query point q, find the triangle it belongs to; then traverse the tree bottom up. In the leaf, q can see all the points of A that are in the same triangle, so we start with that count. As we proceed up the tree,

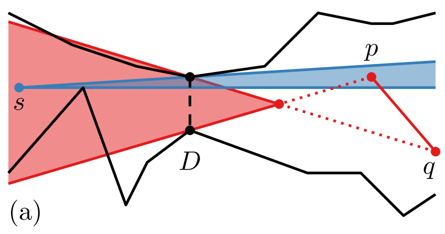

Figure 4 (a) For the cone that describes visibility of $\overline{pq}$ through D, Lemma 10 does not hold – there can be visibility without visibility between the apices of the cones. (b) The segment $\overline{pq}$ intersects the cone of s, and s is in the cone of $\overline{pq}$, but they cannot see each other, so testing intersection between the objects and the cones also does not work directly.

we query the correct associated data structure – if q is to the right of the diagonal, we want to count the points to the left of the diagonal in the current subpolygon that see q. It is easy to see that this way we end up with the total number of points in A that see q, since the subsets of A that we count are disjoint as we move up the tree and cover the entire set A.

▶ **Theorem 11.** *Let P be a simple polygon with n vertices, and let A be a set of m points inside P. In $\mathcal{O}(n + m^{2+\varepsilon} \log n + m \log^2 n)$ time, we can build a data structure of size $\mathcal{O}(n + m^{2+\varepsilon} \log n)$ that can report the number of points from A visible from a query point q in $\mathcal{O}(\log n \log m + \log^2 n)$ time.*

Proof. The correctness follows from the considerations above; it remains to analyse the time and storage requirements. For the query time, we do point location of the query point q in the triangulation of P and make a single pass up the decomposition tree, making queries in the associated multilevel cutting trees. Clearly, the height of the tree is $\mathcal{O}(\log n)$. At every level of the decomposition tree, we need to construct the visibility cone from the query point to the current diagonal; this can be done in $\mathcal{O}(\log n)$ time with a cone query (see Section 2). Then we need to query the associated data structure, except at the leaf, where we simply fetch the count. The query then takes time $\mathcal{O}(\log n \log m + \log^2 n)$. For the storage requirements, we need to store the associated data structures on in total m points at every level of the tree, as well as a single copy of the shortest path data structure, yielding overall $\mathcal{O}(n + m^{2+\varepsilon} \log n)$ storage. Finally, we analyse the preprocessing time. Triangulating a simple polygon takes $\mathcal{O}(n)$ time [10]. Constructing the decomposition can be done in additional $\mathcal{O}(n)$ time [22]. Constructing the associated data structures takes time $\mathcal{O}(m^{2+\varepsilon})$ per level, so $\mathcal{O}(m^{2+\varepsilon} \log n)$ overall, after determining the visibility cones for the points of A to all the relevant diagonals, which can be done in time $\mathcal{O}(m \log^2 n)$, as each point of A occurs a constant number of times per level of the decomposition, and constructing the cone takes $\mathcal{O}(\log n)$ time. Overall we need $\mathcal{O}(n + m^{2+\varepsilon} \log n + m \log^2 n)$ time. ◀

▶ Remark 12. While this approach uses many of the ideas needed to tackle the setting with segment queries, Lemma 10 does not apply – see Figure 4.

4 Segment Queries

In this section, we are given a simple polygon P and a set A of stationary entities (points) in P. We construct a data structure to count the points in A that see a query segment $\overline{pq}$. We cannot reuse the approach of Section 3.1, as the query $\overline{pq}$ may intersect multiple arrangement cells. Thus, we construct a new data structure using the insights of the hierarchical decomposition of Section 3.2. For the complete argument we refer to the full version.

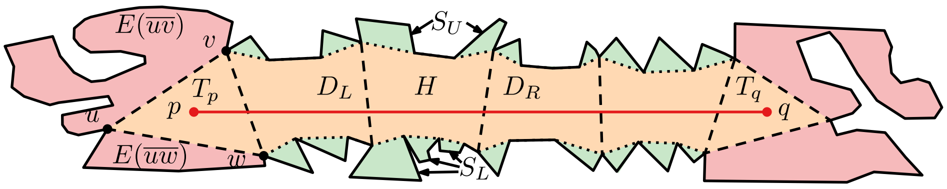

Figure 5 Partitioning of the polygon based on the polygon cover of $\overline{pq}$. The hourglasses and triangles are shown in orange; the side polygons in green; and the end polygons in red.

High-level overview. We use the data structure by Guibas and Hershberger [22] (abbreviated GHDS) on P as the basis and augment the elements of GHDS with data structures that allow us to perform the queries. Recall from Section 2 that we can obtain the polygon cover for a query segment $\overline{pq}$ using the GHDS, consisting of $\mathcal{O}(\log n)$ hourglasses and two triangles T_p, T_q containing p and q. For a given query $\overline{pq}$, the polygon cover partitions P into regions of three types (see Figure 5):

1. Hourglasses and triangles T_p, T_q intersecting $\overline{pq}$ and containing p or q, respectively.
2. Side polygons, each incident to the upper or lower chain of an hourglass.
3. End polygons that are adjacent to T_p and T_q.

For each type, we now count the points in A in a region of that type that see $\overline{pq}$.

4.1 Counting Points from A in Triangles and Hourglasses

We count the points in A contained in a region of Type 1 that see $\overline{pq}$. Triangles are convex, so any point in a triangle can see $\overline{pq}$. Similarly, each hourglass is completely traversed by $\overline{pq}$, so every point inside an hourglass can see $\overline{pq}$. During preprocessing, for every region in the GHDS (a triangle T or an hourglass H), we count the points from A in the region. Specifically, we construct a half-plane range query data structure in $\mathcal{O}(m^{2+\varepsilon})$ time and, for a triangle T, count the points in $A \cap T$ with three consecutive half-plane range queries. The total complexity of the hourglasses in the GHDS is $\mathcal{O}(n \log^3 n)$ [17, Lemma 7]. We triangulate each hourglass H to compute $|A \cap H|$ in $\mathcal{O}(n \log^3 n \log m)$ total time. Given $\overline{pq}$, we compute the sum over all $\mathcal{O}(\log n)$ Type 1 regions in $\mathcal{O}(\log n)$ time.

▶ **Theorem 13.** *We can construct an $\mathcal{O}(n)$-size data structure in $\mathcal{O}(m^{2+\varepsilon} + n \log^3 n \log m)$ time, so that we can count all points in Type 1 regions that see $\overline{pq}$ in $\mathcal{O}(\log n)$ time.*

4.2 Counting Points from A in Side Polygons

The points in a Type 2 region (side polygon) do not always see $\overline{pq}$. Let H be an hourglass with two diagonals D_L, D_R and two chains π_U, π_L. Let S_U be the side polygon bounded by the upper chain π_U; the analysis for S_L bounded by the lower chain π_L is symmetrical. A point $a \in A \cap S_U$ can see $\overline{pq}$ if its visibility cone $C(a)$ into $P \setminus S_U$ is not empty. Furthermore, we discern three mutually exclusive types of cones (Figure 6):

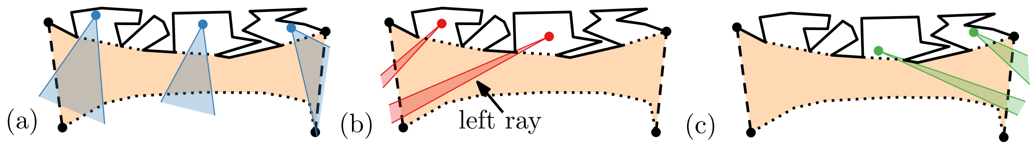

Figure 6 The three cases (a), (b), and (c) for cones originating from a side polygon S_U.

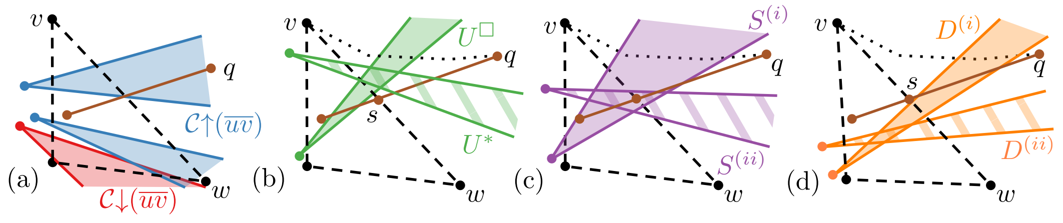

Figure 7 (a) We partition $\mathcal{C}_{\overline{uv}}$ based on vertex w into sets $\mathcal{C}\!\uparrow\!(\overline{uv})$ that pass above w or contain it and $\mathcal{C}\!\downarrow\!(\overline{uv})$ that pass below w. (b–d) We partition $\mathcal{C}\!\uparrow\!(\overline{uv})$ based on $s = \overline{pq} \cap \overline{vw}$ into sets that pass above s (U), below s (D), or contain it (S).

(a) $C(a)$ intersects the lower chain π_L of H;
(b) $C(a)$ intersects *only* the left diagonal D_L of H; and
(c) $C(a)$ intersects *only* the right diagonal D_R of H.
Cones of Type (a) see $\overline{pq}$, since $\overline{pq}$ separates the upper and the lower chain (we count them during preprocessing). Types (b) and (c) are symmetrical; we discuss Type (b) for a fixed side polygon S_U and, without loss of generality, we assume that the directed segment $\overline{pq}$ crosses D_L before D_R. At preprocessing, we store $A_L \subseteq A \cap S_U$ where for all $a \in A_L$, visibility cone $C(a)$ intersects only D_L; and we store $|A_L|$. To count the visible cones of Type (b) at query time, we subtract from $|A_L|$ the number of elements in A_L that do *not* see $\overline{pq}$.

▶ **Lemma 14.** *A point $a \in A_L$ is not visible to $\overline{pq}$ if and only if the left ray of $C(a)$ intersects the shortest path from p to the top of D_L.*

On a high level, we count the points $a \in A_L$ for which the left ray of $C(a)$ intersects the shortest path from p to the top vertex v of D_L as follows. For each hourglass in GHDS, we store the shortest path map SPM(v) rooted at v [23], and for each edge e in SPM(v), we store the number of points $a \in A_L$ for which the left ray of $C(a)$ intersects e. We construct the SPMs and the cutting trees for the $\mathcal{O}(n)$ hourglasses in the GHDS using $\mathcal{O}(nm^{2+\varepsilon} + n^2)$ space and $\mathcal{O}(nm^{2+\varepsilon} + n^2 \log m)$ time. At query time, we obtain the shortest path $\pi(p, v)$ from p to v as a single segment $\overline{pu}$, followed by a path $\pi(u, v)$ in SPM(v) for some vertex u. We count the points $a \in A_L$ where the left ray of $C(a)$ intersects $\overline{pu}$ in $\mathcal{O}(\log m)$ time, and those where the left ray of $C(a)$ intersects $\pi(u, v)$ in $\mathcal{O}(\log n)$ time, thus avoiding double counting. Thus, we count the elements in A_L that do *not* see $\overline{pq}$ in $\mathcal{O}(\log nm)$ time per side polygon. Given $\overline{pq}$, we apply this strategy for all $\mathcal{O}(\log n)$ hourglasses.

▶ **Theorem 15.** *We can construct an $\mathcal{O}(nm^{2+\varepsilon} + n^2)$-size data structure in time $\mathcal{O}(nm^{2+\varepsilon} + n^2 \log m)$, so we can count all points in Type 2 regions that see $\overline{pq}$ in $\mathcal{O}(\log n \log nm)$ time.*

4.3 Counting Points from A in End Polygons

Type 3 regions (end polygons) are bounded by triangles in the decomposition T_p or T_q, containing p or q, respectively. Let $T_p = uvw$; it is incident to at most three end polygons. We describe the data structure for the end polygon $E(\overline{uv})$ incident to T_p through the edge $\overline{uv}$ (see Figure 7). All other data structures are symmetrical. We count the points in $A \cap E(\overline{uv})$ that see $\overline{pq}$. Denote the set of all visibility cones $V(a, \overline{uv})$ by $\mathcal{C}_{\overline{uv}}$ over all $a \in A \cap E(\overline{uv})$.

Consider the special case where $\overline{pq}$ is fully contained in a triangle ($T_p = T_q$). A triangle is convex, so we can immediately apply Lemma 8. Now assume $T_p \neq T_q$. The query segment $\overline{pq}$ intersects some boundary of T_p. We construct a data structure under the assumption that $\overline{pq}$ intersects edge $\overline{vw}$ of T_p; for $\overline{pq}$ intersecting $\overline{uw}$, we construct a symmetrical structure. So, given a triangle T_p, where $\overline{pq}$ intersects $\overline{vw}$, we want to count the cones $C(a) \in \mathcal{C}_{\overline{uv}}$ whose apex a sees $\overline{pq}$. Our argument is a multilevel case distinction on $\mathcal{C}_{\overline{uv}}$. We first partition $\mathcal{C}_{\overline{uv}}$ during preprocessing (Figure 7a) into two sets:

1. $\mathcal{C}\downarrow(\overline{uv})$ are the cones in $\mathcal{C}_{\overline{uv}}$ that pass entirely below the vertex w; and
2. $\mathcal{C}\uparrow(\overline{uv})$ are the cones in $\mathcal{C}_{\overline{uv}}$ that pass entirely above or contain the vertex w.

Counting cones in $\mathcal{C}\downarrow(\overline{uv})$ whose apex sees $\overline{pq}$. For any cone $C(a) \in \mathcal{C}\downarrow(\overline{uv})$, its apex a sees $\overline{pq}$ if and only if p lies below the supporting line of the left ray of $C(a)$. We construct a cutting tree on these lines to count such cones in $\mathcal{O}(\log m)$ time.

Counting cones in $\mathcal{C}\uparrow(\overline{uv})$ whose apex sees $\overline{pq}$. We partition $\mathcal{C}\uparrow(\overline{uv})$ with an elaborate double case distinction. Observe that this conceptual case distinction can only be made at query time when we have access to $s = \overline{pq} \cap \overline{vw}$ and q (Figures 7 and 8).

(b) $U \subseteq \mathcal{C}\uparrow(\overline{uv})$ are the cones where both boundary rays intersect $\overline{vs}$ (but not s):
- $U^* \subseteq U$ are the cones whose apices lie above the supporting line of $\overline{sq}$;
- $U^\square \subseteq U$ is the set $U \setminus U^*$.

(c) $S \subseteq \mathcal{C}\uparrow(\overline{uv})$ are the cones which contain $s = \overline{pq} \cap \overline{vw}$:
- $S^{(i)} \subseteq S$ are the cones whose right ray intersects $\overline{sq}$;
- $S^{(ii)} \subseteq S$ is the set $S \setminus S^{(i)}$.

(d) $D \subseteq \mathcal{C}\uparrow(\overline{uv})$ are the cones where both boundary rays intersect $\overline{sw}$ (but not s):
- $D^{(i)} \subseteq D$ are the cones whose right ray intersects $\overline{sq}$;
- $D^{(ii)} \subseteq D$ is the set $D \setminus D^{(i)}$.

Counting cones in U whose apex sees $\overline{pq}$. Both U^* and $U^\square$ may contain cones whose apex sees $\overline{pq}$. We show how to count these for both sets using a data structure *only* on $\mathcal{C}\uparrow(\overline{uv})$. We show that cones in $U^\square$ are visible if and only if they intersect the segment $\overline{ps}$; we can test this in $\mathcal{O}(\log m)$ time. We count the cones in U^* whose apices see $\overline{pq}$ via an inclusion–exclusion argument. Specifically, we observe that all cones in $\mathcal{C}\uparrow(\overline{uv}) \setminus U^*$ have one of two mutually exclusive properties:

(i) For all cones in $S^{(i)}$, $D^{(i)}$, and $U^\square$, their right ray intersects $\pi(v, q)$ and lies above s.

(ii) For all cones in $S^{(ii)}$ and $D^{(ii)}$, their right ray lies below s and does not intersect $\pi(v, q)$.

Cones in U^* never have property (ii). Moreover, they are not visible if and only if their right ray intersects $\pi(v, q)$, i.e. invisible cones have property (i). Thus, the number of cones in U^* whose apices see $\overline{pq}$ is equal to $|\mathcal{C}\uparrow(\overline{uv})|$ minus all cones with property (i) or (ii). We count cones with property (i) using a shortest path map in $\mathcal{O}(\log n)$ time, identically to Section 4.2. We count cones with property (ii) using half-plane range queries in $\mathcal{O}(\log m)$ time.

Counting cones in S and D whose apex sees $\overline{pq}$. The apices of all cones in S see $\overline{pq}$. We identify these in $\mathcal{O}(\log m)$ time using a stabbing query on $\mathcal{C}\uparrow(\overline{uv})$. For cones in D, we can make a symmetrical case distinction, creating the sets D^* and $D^\square$, and we can handle them through an identical data structure.

▶ **Theorem 16.** *We can construct an $\mathcal{O}(nm^{2+\varepsilon} + n^2)$-size data structure in time $\mathcal{O}(nm^{2+\varepsilon} + n^2 \log m)$, so we can count all points in Type 3 regions that see $\overline{pq}$ in $\mathcal{O}(\log n \log nm)$ time.*

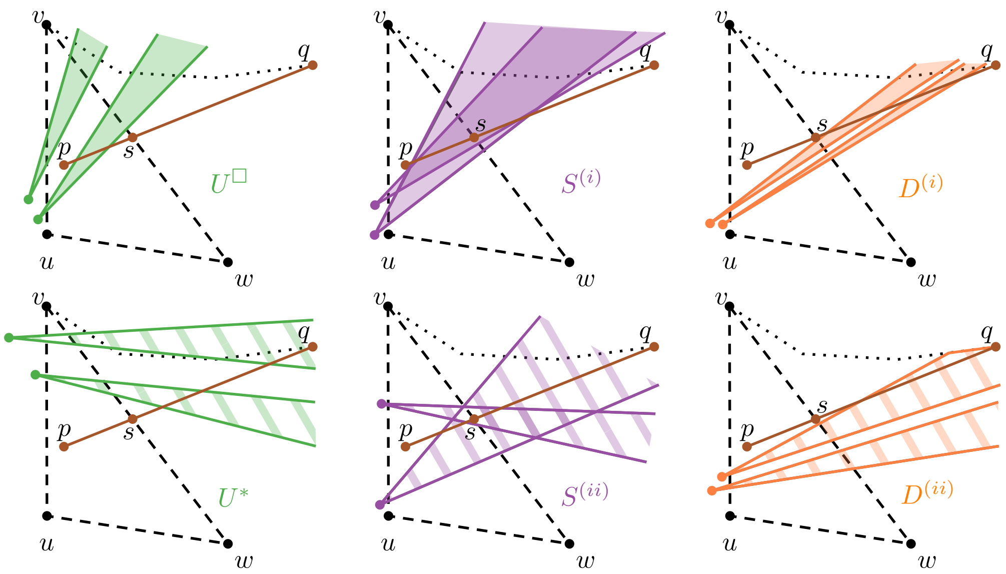

Figure 8 We partition $\mathcal{C}\uparrow(\overline{uv})$ into six sets: U^*, $U^{\square}$, $S^{(i)}$, $S^{(ii)}$, $D^{(i)}$, and $D^{(ii)}$.

Finalising the argument. In the full version, we show a lemma that allows us to efficiently construct all visibility cones from points of A in Type 2 and Type 3 regions. We store all points and visibility cones in the data structures of Theorems 13, 15, and 16 to show the main result of this section.

▶ **Theorem 17.** *Let P be a simple polygon with n vertices, and let A be a set of m points inside P. In time $\mathcal{O}(nm^{2+\varepsilon} + n^2 \log m)$, we can build a data structure of size $\mathcal{O}(nm^{2+\varepsilon} + n^2)$ to count the points from A visible from a query segment $\overline{pq}$ in $\mathcal{O}(\log n \log nm)$ time.*

5 Extensions and Future Work

We consider further variants of the problem. In particular, we extend the approach of Section 4 to the cases where A contains line segments, or even constant-complexity simple polygons, with the same bounds. Furthermore, we consider a version of the problem where we only want to test if at least one object in A sees the query. Finally, we tackle the question of subquadratic counting, where given two sets A and B with m points each, we wish to count the pairs from $A \times B$ that see each other in the simple polygon in time subquadratic in m. We briefly describe these results here and further refer the reader to the full version.

Extending to a set of segments or polygons. A natural extension to our GHDS would be to consider the case where A is a set of segments. As it turns out, there is a straightforward way to reuse our GHDS. Observe that any line segment that is *not* entirely contained in a single end polygon or side polygon intersects a triangle or an hourglass and thus is visible to $\overline{pq}$. Hence, it suffices to count the *invisible* line segments inside the individual side and end polygons and subtract that count from the total. Instead of the visibility cones, we use the *visibility glasses,* i.e. the collections of all visibility lines between two line segments (Figure 9). By intersecting the bitangents of the visibility glass, we get a new cone that describes the potentially visible region on the other side of the diagonal. The preprocessing time for GHDS construction dominates, and we do not add extra query time or storage. To use the same approach with constant-complexity simple polygons, we simply show that we can construct visibility glasses for them, as well. See the full version for details.

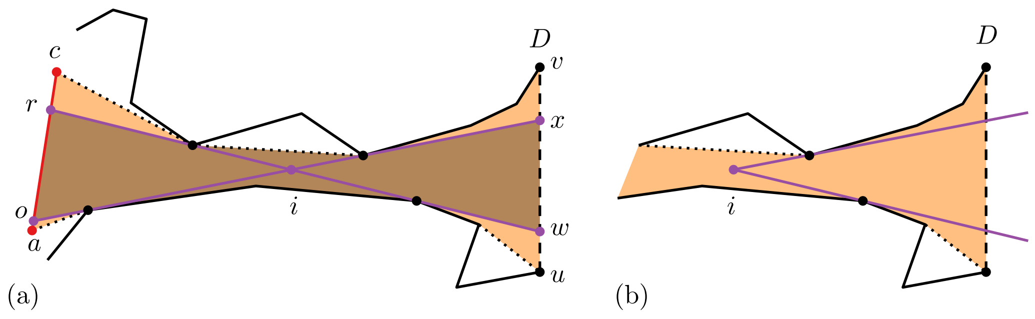

Figure 9 (a) The visibility glass (dark region) inside the hourglass (orange region) from segment $\overline{ac}$ to diagonal $D = \overline{uv}$. (b) The intersection point i and the two rays from i through w and x form a new visibility region in the subpolygon to the right of D.

Testing for visibility. If all we need to compute is if any of the objects from A can see Q we can avoid computing $C(Q, A)$, and answer queries more efficiently. We explicitly compute the union U of all (weak) visibility polygons of all m objects in A, and build point location and ray shooting data structures on it. Irrespective of the type of objects in A (points or segments) and the type of Q, we argue that U has complexity $\mathcal{O}(m(m+n))$, and that we can store using $\mathcal{O}(m(m+n))$ space so that we can answer queries in $\mathcal{O}(\log(m+n))$-time.

Subquadratic counting. Given two sets of points or line segments A and B, each of size m, in a simple polygon P with n vertices, we want to count the pairs in $A \times B$ that see each other. Using the work by Eades et al. [17], we can solve this problem by checking the visibility for all pairs, which is optimal for $n \gg m$. If at least one set consists of points, it runs in time $\mathcal{O}(n + m^2 \log n)$. When $m \gg n$, we want to avoid the m^2 factor, using our data structures. Suppose we have a data structure for visibility counting queries with query time $Q(m, n)$ and preprocessing time $P(m, n)$. Pick $k = m^s$ with $0 \leq s \leq 1$. We split the set A into sets $A_1, \dots, A_k$, with $^m/_k$ objects each; then we construct a data structure for each set. Finally, with each point in B, we query these k data structures and sum up the counts. The time that this approach takes is $\mathcal{O}\big(m^s \cdot P(m^{1-s}, n) + m^{1+s} \cdot Q(m^{1-s}, n)\big)$. Picking s to minimise it, we obtain algorithms subquadratic in m in all settings.

Moving points and query variations. We can interpret a line segment as a trajectory traced by a point moving with constant velocity. The results from Sections 3 and 4 directly apply in this setting, too. When A consists of line segments as above, we solve a different problem. Solving the visibility counting problem for moving points seems non-trivial. Alternatively, one could report the visible points, or ask to quantify the visible portions of segments. All of these would be highly exciting directions for future work.

References

1 Pankaj K. Agarwal and Micha Sharir. Applications of a new space-partitioning technique. *Discrete & Computational Geometry*, 9:11–38, 1993. doi:10.1007/BF02189304.

2 Pankaj K. Agarwal and Marc J. van Kreveld. Connected component and simple polygon intersection searching. *Algorithmica*, 15:626–660, 1996. doi:10.1007/BF01940884.

3 Sharareh Alipour, Mohammad Ghodsi, Alireza Zarei, and Maryam Pourreza. Visibility testing and counting. *Information Processing Letters*, 115(9):649–654, 2015. doi:10.1016/j.ipl.2015.03.009.

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

5 Boaz Ben-Moshe, Olaf Hall-Holt, Matthew J. Katz, and Joseph S. B. Mitchell. Computing the visibility graph of points within a polygon. In Jack S. Snoeyink and Jean-Daniel Boissonnat, editors, *Proceedings of the 20th Annual Symposium on Computational Geometry (SoCG 2004)*, pages 27–35, New York, NY, USA, 2004. ACM. doi:10.1145/997817.997825.

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

7 Kevin Buchin, Bram Custers, Ivor van der Hoog, Maarten Löffler, Aleksandr Popov, Marcel Roeloffzen, and Frank Staals. Segment visibility counting queries in polygons, 2022. arXiv:2201.03490.

8 Mojtaba Nouri Bygi, Shervin Daneshpajouh, Sharareh Alipour, and Mohammad Ghodsi. Weak visibility counting in simple polygons. *Journal of Computational and Applied Mathematics*, 288:215–222, 2015. doi:10.1016/j.cam.2015.04.018.

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

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

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

12 Bernard Chazelle and Herbert Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39(1):1–54, 1992. doi:10.1145/147508.147511.

13 Bernard Chazelle and Leonidas J. Guibas. Visibility and intersection problems in plane geometry. *Discrete & Computational Geometry*, 4:551–581, 1989. doi:10.1007/BF02187747.

14 Bernard Chazelle, Micha Sharir, and Emo Welzl. Quasi-optimal upper bounds for simplex range searching and new zone theorems. *Algorithmica*, 8:407–429, 1992. doi:10.1007/BF01758854.

15 Danny Ziyi Chen and Haitao Wang. Weak visibility queries of line segments in simple polygons. *Computational Geometry: Theory & Applications*, 48(6):443–452, 2015. doi:10.1016/j.comgeo.2015.02.001.

16 Kenneth L. Clarkson. New applications of random sampling in computational geometry. *Discrete & Computational Geometry*, 2:195–222, 1987. doi:10.1007/BF02187879.

17 Patrick Eades, Ivor van der Hoog, Maarten Löffler, and Frank Staals. Trajectory visibility. In Susanne Albers, editor, *Proceedings of the 17th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2020)*, number 162 in Leibniz International Proceedings in Informatics (LIPIcs), pages 23:1–23:22, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.SWAT.2020.23.

18 Hossam El Gindy 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.

19 Subir Kumar Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, Cambridge, UK, 2007. doi:10.1017/CBO9780511543340.

20 Jacob E. Goodman, Joseph O'Rourke, and Csaba D. Tóth, editors. *Handbook of Discrete and Computational Geometry*. Chapman and Hall/CRC, New York, NY, USA, 3rd edition, 2017. doi:10.1201/9781315119601.

21 Joachim Gudmundsson and Pat Morin. Planar visibility: Testing and counting. In David G. Kirkpatrick and Joseph S. B. Mitchell, editors, *Proceedings of the 26th Annual Symposium on Computational Geometry (SoCG 2010)*, pages 77–86, New York, NY, USA, 2010. ACM. doi:10.1145/1810959.1810973.

22 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.

23 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*, 2:209–233, 1987. doi:10.1007/BF01840360.

24 Prosenjit Gupta, Ravi Janardan, and Michiel H. M. Smid. Further results on generalized intersection searching problems: Counting, reporting, and dynamization. *Journal of Algorithms*, 19(2):282–317, 1995. doi:10.1006/jagm.1995.1038.

25 John Hershberger. A new data structure for shortest path queries in a simple polygon. *Information Processing Letters*, 38(5):231–235, 1991. doi:10.1016/0020-0190(91)90064-O.

26 John Hershberger and Subhash Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18(3):403–431, 1995. doi:10.1006/jagm.1995.1017.

27 Barry Joe and Richard B. Simpson. Corrections to Lee's visibility polygon algorithm. *BIT Numerical Mathematics*, 27:458–473, 1987. doi:10.1007/BF01937271.

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

29 Der-Tsai Lee. Visibility of a simple polygon. *Computer Vision, Graphics, and Image Processing*, 22(2):207–221, 1983. doi:10.1016/0734-189X(83)90065-8.

30 Joseph O'Rourke. *Art Gallery Theorems and Algorithms*, volume 3 of *The International Series of Monographs on Computer Science*. Oxford University Press, Oxford, UK, 1987. URL: http://www.science.smith.edu/~jorourke/books/ArtGalleryTheorems/art.html.

31 Mark H. Overmars and Emo Welzl. New methods for computing visibility graphs. In Herbert Edelsbrunner, editor, *Proceedings of the 4th Annual Symposium on Computational Geometry (SoCG 1988)*, pages 164–171, New York, NY, USA, 1988. ACM. doi:10.1145/73393.73410.

32 Subhash Suri and Joseph O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In Alok Aggarwal, editor, *Proceedings of the 2nd Annual Symposium on Computational Geometry (SoCG 1986)*, pages 14–23, New York, NY, USA, 1986. ACM. doi:10.1145/10515.10517.