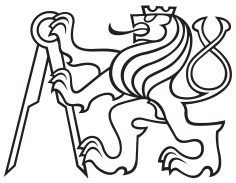
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of cybernetics

Fast Computation of Visibility Polygons

Bc. Jakub Rosol

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Field of study: Cybernetics and Robotic
January 2023

Acknowledgements

I would first like to thank my thesis advisor RNDr. Miroslav Kulich, Ph.D. of the Intelligent mobile robotics group at Czech technical university. Mr Kulich advised me whenever I ran into trouble or had a question about writing and kept me motivated in my research.

I would also like to thank Ing. Jan Mikula, for introducing me to the initial state of the problem and giving me helpful implementation advice.

Finally, I must express my gratitude to my mother and sister for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Jakub Rosol

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 10. January 2023, Jakub Rosol

Abstract

The computation of visibility regions has been an important part of computational geometry since 1979. It finds its use in many fields, including robot control, where computational time is crucial. We introduce one new algorithm for the computation of visibility from a point and one algorithm for the computation of visibility from a segment. The state-of-the-art *Triangular expansion algorithm* (TEA) is used as the benchmark for the computation of visibility regions for points. At first, we describe an algorithm presented in the *Polyanya* project that extends TEA from triangles to polygons enhancing the query performance by 20%. Further, we introduce the first solution (PEA-E) for continuous computation of visibility regions for segments (until now, they were computed by sampling the segment with points). At last, we present an algorithm for point visibility that builds on PEA-E and has a twice as fast response to query than TEA. Proposed algorithms provide significant advances in the field of visibility computation.

Keywords: Visibility Region, Triangular Expansion Algorithm, Iron Harvest Benchmarks, Polygonal Expansion Algorithm, Visibility from Segment, Computational geometry

Supervisor: RNDr. Miroslav Kulich, Ph.D.
Czech institute of informatics, Intelligent mobile robotics B-322,
Jugoslávských partyzánů 1580,
Praha 6

Abstrakt

Výpočet oblastí viditelnosti je důležitou součástí výpočetní geometrie již od roku 1979. Své využití nachází v mnoha oblastech včetně řízení robotů, kde je čas výpočtu rozhodující. Představujeme nový algoritmus pro výpočet viditelnosti z bodu a nový algoritmus pro výpočet viditelnosti z úsečky. Momentálně nejlepší algoritmus *triangular expansion algorithm* (TEA) je použit jako výchozí srovnávací standard pro výpočet oblastí viditelnosti bodů. Nejprve popisujeme algoritmus prezentovaný v projektu *Polyanya*, který rozšiřuje TEA z trojúhelníků na polygony a zvyšuje výkon o 20%. Dále představujeme první řešení (PEA-E) pro spojitý výpočet oblasti viditelnosti z úsečky (dosud byly úsečky nahrazovány konečnou množinou bodů). Nakonec představujeme algoritmus pro viditelnost z bodů, který staví na PEA-E a má dvakrát rychlejší odezvu na dotaz než TEA. Představené algoritmy poskytují nezanedbatelný posun v oblasti výpočtu viditelnosti.

Klíčová slova: Oblast viditelnosti, Triangular Expansion Algorithm, Iron Harvest Benchmark, Polygonal Expansion Algorithm, Viditelnost z úsečky, Výpočetní geometrie

Překlad názvu: Rychlý výpočet polygonu viditelnosti

Contents

Figures

Tables

Chapter 1

Introduction

The visibility problem has been a high-interest issue in geometrical computing for a few decades. It has applications in simulations, games, robot control and many others. Although, in the real world, visibility is not computed but obtained by sensor systems such as LIDAR, cameras, or eyes, in the case of humans, in the virtual world, what is visible has to be computed.

In many games, visibility is essential. For example, in shooter games, it is needed for the evaluation of hits. Strategy games sometimes use *fog of war*, which prevents players from seeing parts of the map which are not visible to their troops. Visibility must then be computed for all units in the game. The necessity of visibility evaluation can be expressed by a warning known to many players: "Target is not in the line of sight."

Simulations of complex systems may require visibility not only for image reasons since it does affect properties other than just visibility. For example, the strength of a wireless signal can be approximated by computing distances from the source. However, obstacles between the transmitter and the receiver may have a dire effect on the actual result. The visibility cannot be omitted in simulations of autonomous agents whose decisions are based on visual input.

The usage for robot control is well described in work by J. Mikula [MK22]. Consider a mobile robot with an omnidirectional vision that has to search a known building floor. To be sure the robot searched the whole floor, it has to compute its visibility. The path has to be planned so that the robot sees each part of the floor at least once. Visibility can also be used in robot

localization which is one of the bases for robot control (a robot cannot decide where to go if it does not know where it is). Consider a real mobile robot with a LIDAR sensor that measures the distances from obstacles in 360 degrees around the robot. The robot is placed arbitrarily on a known building floor. To compute where it is, the algorithm may compare readings from the LIDAR with simulated visibility in the virtual map of the floor. The robot is most likely where the simulated vision corresponds to the actual measurement.

The contribution of this thesis lies in a reduction of the *Polyanya* [CHG17] project to a standalone library for the computation of visibility regions from points, in introducing the first algorithm for the computation of visibility regions for segments, and in presenting a new algorithm for visibility from points. We stripped *Polyanya* source from all path-finding related parts and introduced robust orientation tests to it. The implementation represents the state-of-the-art expansion algorithm (for triangles in 2014 [BHH+14], expanded to polygons in 2017 [CHG17]). Further, we introduce *polygonal expansion algorithm for edges* (PEA-E) which is the first algorithm able to compute visibility from a segment without sampling it with points. Our last contribution lies in presenting a new algorithm *EdgeVis* for the computation of visibility from a point. *EdgeVis* uses PEA-E to precompute visibility regions for all edges in a mesh and uses those regions as a reduction of the searched area. We also present experiments which describe the basic properties and performance of all discussed algorithms. The *EdgeVis* enhances current state-of-the-art by 33%. Additionally, we compare robust orientation tests by R. J. Shewchuk with naive implementation with limited precision.

1.1 Problem definition

The problem of visibility computation can be solved in varying spaces, but this paper is focused on visibility in two-dimensional space. Let $\mathbf{W}$ be a subset of R^2 that represents the environment without any mobility and visibility restrictions. Then, $\mathbf{O} = \mathbf{W}^c$ is the rest of the two-dimensional space, and it is considered an **o**bstacle that cannot be passed or seen through. Point P is visible from point Q if $\overline{PQ} \subset \mathbf{W}$.

One of the most common representations of $\mathbf{W}$ is a polygon (with polygonal holes). Its boundaries represent a transition between observable and unobservable space. Edges of the polygon exterior and all hole boundaries must form a closed chain without intersecting itself or other boundaries. Some current algorithms compute visibility graphs that define visibility relations between vertices in the environment, while others compute visibility region

from a query point anywhere in $\mathbf{W}$. The visibility graph $G(\mathbf{W})$ is a structure storing information about visibility between all vertices in $\mathbf{W}$ in a set of all edges that can be defined by two vertices that see each other. Visibility region $\mathbf{V}(Q)$ of a query point Q is a subset of $\mathbf{W}$ where all points are visible from Q:

$$\mathbf{V}(Q) = \{\forall X \in \mathbf{W} \mid \overline{QX} \subset \mathbf{W}\}. \tag{1.1}$$

This work focuses on revising current approaches for the computation of visibility regions from a query and trying to improve them.

1.2 Related work

In 1979 visibility regions for points in a single polygon were introduced by L. S. Davis, and M. L. Benedikt [DB79]. The time complexity of the proposed algorithm is $O(n^2)$. In the early '80s, linear $O(n)$ algorithms were presented by H. ElGindy, and D. Avis [EA81] and by D. T. Lee[Lee83], but in 1987 B. Joe and R. B. Simpson [JS87] showed that these algorithms might fail and presented first correct $O(n)$ algorithm for polygons without holes. These algorithms emulate a sweeping line which follows the boundary vertex by vertex. The boundary is sorted counter-clockwise, and the line follows its vertices. If the line moves counter-clockwise, the vertex can be visible and added to the stack; if the line moves clockwise, some already added vertices must be swept away.

In realistic scenarios, polygons representing an environment are usually complex, with holes representing untraversable areas. T. Asano first proposed an algorithm for polygons with h holes with time complexity $O(n \log h)$[Asa85]. Instead of sweeping the polygon based on the boundary, the algorithm sweeps the plane with a ray intersecting all edges. It starts with computing polar coordinates of all boundary vertices relative to the query and sorting them. Then as the ray rotates, the algorithm computes whether a new vertex is closer to the query point than all currently intersected edges. If the point is the closest, new points for the final visibility region are generated (usually the new point and an intersection). In 1995 an optimal $O(n + h \log h)$ algorithm was introduced by P. J. Heffernan and J. S. B. Mitchell [HM95]. A summary of algorithms proposed since 1979 was published in a book by S. K. Ghosh in 2007 [Gho07].

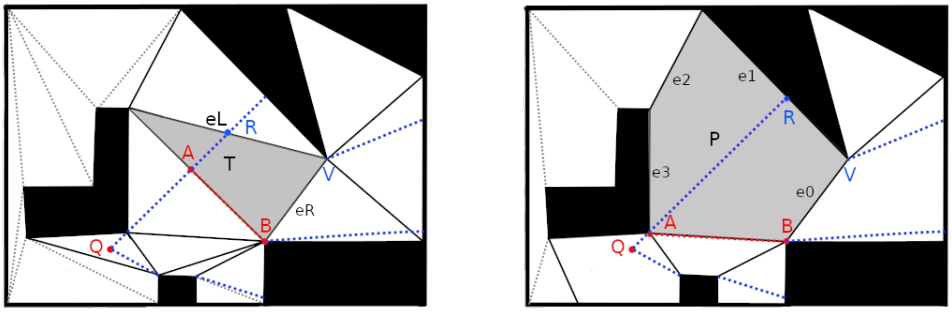
Many algorithms presented after 2007 were based on tradeoffs between a preprocessing time and the query's computation time. For example, A. Zarei and M. Ghodsi proposed an algorithm with a $O(n^3 \log n)$ preprocessing and $O((1 + \min(h, k) \log n + k)$ query time, where new parameter k represents

the number of vertices defining visibility region from the query. During preprocessing, the algorithm slices the polygon with holes into a set of simple polygons on which the visibility region can be computed more efficiently. A better algorithm on a similar principle was later proposed by R. Inkulu with S. Kapoor [IK09].

In 2014, F. Bungiu et al. introduced the *triangular expansion algorithm* [BHH+14] (TEA). This algorithm has time complexity $O(n^2)$ in theory, but it computes visibility region for a query very efficiently in practice (faster by two orders of magnitude than *Asano's* algorithm). During preprocessing, the polygon is converted to a triangular mesh. When a query is given, its visibility is gradually expanded from the initial polygon to others. This algorithm evaluates only visible parts of the polygon, which is why it performs well in realistic scenarios. Visibility regions in complex environments are much smaller than the environment itself. Chapter 2 describes TEA in more detail.

In 2022, our colleague J. Mikula showed that the query performance of TEA is affected by triangulation [MK22]. He proposed a new triangulation *MinTV* on which TEA query performance increased by 5-28% compared to commonly used CDT. He also introduced *d*-TEA modification for computing visibility regions with a limited range of vision. His work presents a state-of-the-art version of TEA.

In 2017 M. Cui, D. D. Harabor, and A. Grastien presented a new state-of-the-art algorithm *polyanya* for path-finding based on visibility in convex polygons [CHG17]. They are computing the visibility by expanding TEA to polygons. Since it is only a part of the *polyanya* project, the authors do not pay much attention to it. We remove unnecessary code in *polyanya* and describe it as a stand-alone algorithm *polygonal expansion algorithm* (PEA) introduced in Chapter 3. An example that shows a different number of expansions for TEA and PEA on the same query is shown in Fig. 1.1.



(a) : TEA - entering triangle T. **(b) :** PEA - entering polygon P.

Figure 1.1: Expansion algorithms for computation of visibility region from point Q.

Chapters 4 and 5 describe two proposed algorithms. Chapter 4 presents *polygonal expansion algorithm for edges* (PEA-E), which computes visibility regions for segments in polygonal mesh (without sampling the segment). Chapter 5 describes new algorithm *EdgeVis*, that uses PEA-E precomputed structure of visibility regions from edges in the mesh for fast computation of visibility from a query point.

All algorithms implemented in this paper first need to subdivide polygon $\mathbf{W}$ representing the environment into a polygonal mesh $\mathbf{M}$ composed of convex polygons $\mathbf{P}_i$ (resp. $\mathbf{T}_i$ for triangular mesh). The creation of the mesh can affect query performance [MK22]. However, the goal of this work is not to find the optimal division of $\mathbf{W}$, so the generation of triangular and polygonal mesh from *polyanya* project [CHG17] is used. The triangulation is done with *constrained Delaunay triangulation* implemented in Fade2D library [Lee19], and triangles can be further merged to a mesh of convex polygons.

Every algorithm since 1979 must evaluate the position of a point to an oriented line (also called orientation or winding). The theoretical approach to compute the position of P to line $\overrightarrow{AB}$ is to compute cross-product $O = (B - A) \times (P - B)$. P is counter-clockwise from $\overrightarrow{AB}$ if $O > 0$ and clockwise if $O < 0$. $O = 0$ only if $P \in \overrightarrow{AB}$. Roundoff errors produced by the finite precision of computers may produce an incorrect result, which may produce unwanted behaviour of the final application. To eliminate this error effectively, R. J. Shewchuk produced algorithms and implementations for quick and correct tests of orientation [She97]. Presented algorithms are implemented with robust tests from R. J. Shewchuk and elementary tests where rounding errors are resolved with tolerance constant ϵ.

Chapter 2

Triangular expansion algorithm

2.1 Motivation

The computation of visibility regions for a query point has been an important part of computational geometry since 1979. Most of the algorithms introduced since then evaluate all the points describing the environment. In 2014 Bungiu et al. presented an expansion algorithm that evaluates only the visible area [BHH+14]. Even though this algorithm is not optimal in theory, experiments have shown that it performs very well in realistic scenarios where most queries see only a tiny portion of the environment. *Triangular expansion algorithm* (TEA) is implemented as a benchmark algorithm for this thesis and we describe it in the following paragraphs.

2.2 Definition

The algorithm works on triangular mesh, so two-dimensional representation of the environment $\mathbf{W}$ has to be divided into a set of triangles $\mathbf{T}_i$. Commonly $\mathbf{W}$ is defined as a polygon with holes that can be transformed to a set of triangles for example with *constrained Delaunay triangulation*.

2.3 Methodology

The algorithm gradually expands the visibility region over visible and traversable edges, until no expandable edges are available. Usually, it means that the algorithm reached an obstacle in each direction, but a distance from the query can also limit it.

Algorithm 1 TEA

1: **Inputs:**
2: Q: Point for which visibility is computed.
3: W: Polygon representing environment.
4: **Output:**
5: *visibility*: Visibility region of Q.
6: **Initialization:** $mesh \leftarrow CDT(W)$
7: $expandable \leftarrow getInitialEdges(Q, mesh)$
8: **for** e in expandable **do**
9: $visibility \leftarrow$ **Expand**$(Q,\ e,\ mesh)$
return *visibility*

On algorithm startup (Algorithm 1), before any query is given, the mesh is loaded or computed from **W** (line 6). For every given query Q, the algorithm first finds the initial triangle $\mathbf{T}_0$ so that $Q \in \mathbf{T}_0$. Whole $\mathbf{T}_0$ is visible from Q, so all its edges can be used as initial edges for expansion (line 7). Every initial edge e is then recursively expanded to new triangles until the end condition is met (lines 8 & 9).

Algorithm 2 TEA-E

1: **procedure Expand**
2: **Inputs:**
3: Q: Point for which visibility is computed.
4: e: Segment through which is visibility expanded.
5: *mesh*: Environment represented as triangular mesh.
6: **Output:**
7: *visibility*: Points defining area in mesh visible from s over e.
8: **if** e is not traversable **then**
9: **return** $visibility \leftarrow e$
10: $newExpandable \leftarrow getNewSegments(e,s,mesh)$
11: **for** newE in newExpandable **do**
12: **push:** $visibility \leftarrow$ **Expand**$(node)$ *# Next level of recursion*

The recursive expansion procedure in Alg. 2 stops if expanded edge e is not traversable (does not lead to a new triangle) (lines 8 & 9). If it is traversable, the algorithm gets two unexplored edges e_{left} and e_{right} from the triangle behind e. Visible parts of new edges have to be defined (line 7). New visible edges or their sub-segments can then be expanded further (line 9). A straightforward approach to saving visibility is to push end points of edges leading to obstacles to a list. The resulting list will describe the final visibility region if segments are expanded in a consistent order. The commonly used order is e_{right} first, and e_{left} second (counter-clockwise - CCW).

The algorithm proceeds as a depth-first search as shown in (Fig. 2.1). Because the whole edge e is visible from Q, the visibility region will start at point A and end at G. Calls of the procedure *Expand* are marked with numbered arrows. Blue arrows represent calls that result in further recursion, and green ones represent an end of the recursion branch because the obstacle was reached. If the endpoints of non-traversable edges are saved, and endpoints of e are used, the resulting list will be $\{A, A, B, B, C, C, D, D, E, E, F, G\}$. After removing duplicate points, the resulting visibility region from expansion over e is $\{A, B, C, D, E, F, G\}$.

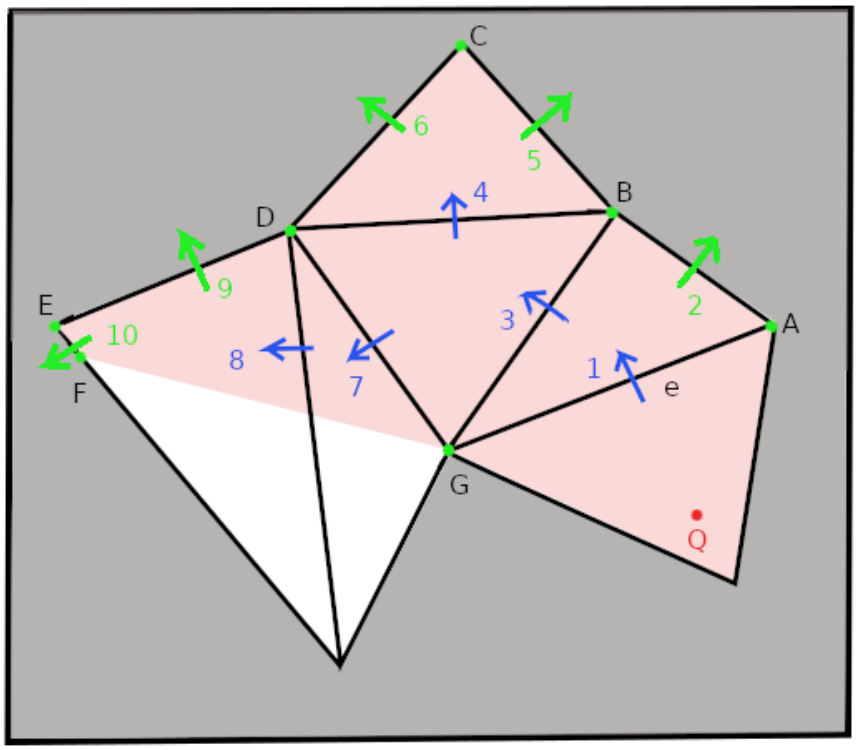


Figure 2.1: Full expansion of TEA over initial edge e.

2.3.1 Initial edges

This section describes the search for the initial triangle and its edges implemented in line 7 of Alg. 1. Because triangles are convex polygons, and convexity guarantees visibility, all edges of the triangle in which the query point lies are wholly visible and can be used as initial edges. The problem is finding the initial triangle.

Lets assume **T** is a triangle defined with three points in 2D space

$$T = A, B, C, \tag{2.1}$$

and Q is an arbitrary point. To decide whether $Q \in T$, orientation of Q to **oriented** triangle edges $\overline{AB}$, $\overline{BC}$, $\overline{CA}$ has to be evaluated. $Q \in T$ only if orientation of Q to all edges is the same. Figure 2.2 shows the resulting orientation in different areas around triangle A, B, C. The oriented edges must define an oriented closed loop for this to apply.

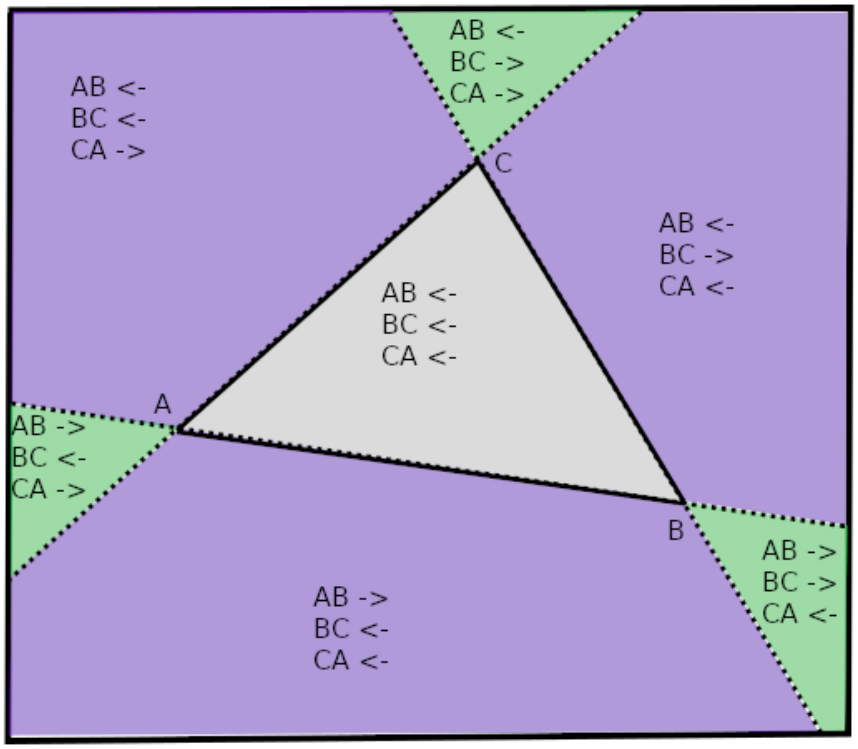


Figure 2.2: Orientation to oriented edges of a triangle.

A naive approach to finding the initial triangle is to loop through all triangles and check the position of Q as above. The worst-case complexity

of such initialization is $O(t)$, where t is the number of triangles in the mesh. Finding the initial polygon is needed for every query, and this naive approach would harshly affect the performance of TEA on complex maps.

An improvement in query time can be introduced with additional preprocessing of the mesh. The environment can be overlayed with a uniform rectangular grid. Unlike triangles in the mesh, the cells in the grid can be sorted by x and y coordinates.

The cell containing Q can be accessed directly. Assume a map with size (w, h) in x, y coordinate system. An overlaying grid with dimensions $n_w \times n_h$ have resolution $r_w = \frac{w}{n_w}$ and $r_h = \frac{h}{n_h}$. The i_x and i_y index of the cell containing point $A = \{x_A, y_A\}$ can be computed as

$$i_x = round(\frac{x_A}{r_w}), \tag{2.2}$$

$$i_y = round(\frac{y_A}{r_h}). \tag{2.3}$$

The result has to be rounded down if the indexing starts at 0 and rounds up if the indexing starts at 1.

On the triangular mesh, a single cell of overlaying grid overlaps only a limited number of triangles. Because the initial cell can be accessed directly, the evaluation if Q lies in a triangle is limited to triangles overlapped by the initial cell. This information can be preprocessed.

2.3.2 New segments

The main procedure of TEA for the expansion of the visibility region implemented in line 9 in Alg. 1 and line 12 in Alg. 2 is described in this section. Because the algorithm expands visibility over edges or segments, visibility region from a point Q over segment s must be defined:

$$\mathbf{V}^t(Q, s) = \{\forall X \in \mathbf{W} | \overline{QX} \subset \mathbf{W} \wedge \overline{QX} \cap s \neq \varnothing\}. \tag{2.4}$$

The only difference between the definition 2.4 and the general definition 1.1 for visibility region is that the line connecting two points must not only be in the open area but also it has to intersect segment s. The visible area from point Q over segment $s = \overline{AB}$ is limited by boundary lines $\overrightarrow{QA}$ and $\overrightarrow{QB}$ as in Fig. 2.3.

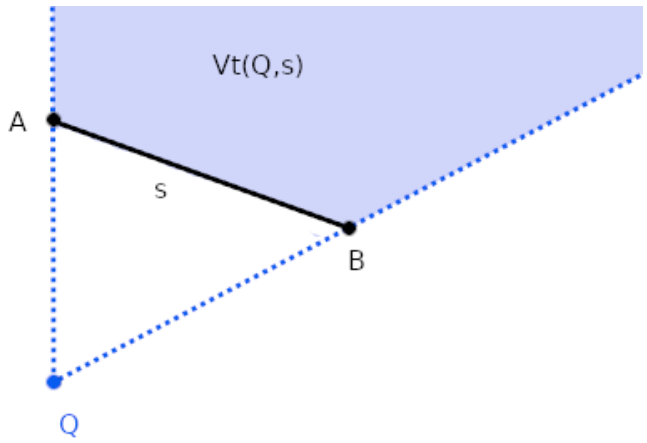


Figure 2.3: Visibility region from point Q over segment s.

During expansion to new triangle T, the visibility is expanded over a segment that lies on one of the triangle's edges (or is equal to one). The algorithm knows which part of the expanded edge is visible (it looks over it) and computes the visibility of the rest edges. The new visible segments can be expanded further. In figure 2.4 visibility from query point, Q is expanded to triangle **T** over edge $\overline{AB}$. Because an obstacle blocks vision, the visibility is expanded only over segment $s \subset \overline{AB}$ defined with points A and B'. Before computing which parts of the remaining edges $\overline{BC}$ and $\overline{CA}$ are visible, the algorithm must decide whether they can be visible at all. Both edges are at least partially visible if vertex $C \in \mathbf{V}^t(Q, s)$. If C is not visible, one edge is not visible at all, and the second one must be visible partially. In the figure, C is visible, so the algorithm looks for a visible segment on both edges. Visibility of A guarantees visibility of whole edge $l = \overline{CA}$. On the right side, the vision was obstructed, and vertex B is not visible. The right visible segment is defined as $r = \overline{B_{new}C}$, where B_{new} is a intersection of edge $\overline{BC}$ and boundary $\overrightarrow{QB'}$. New segments r and l will be expanded to adjacent triangles **RT** and **LT**, respectively.

The code for this is in Alg. 3. This procedure assumes that expanded segment $s \subset \overline{AB}$ and points A, B, C are sorted counter-clockwise. The algorithm first defines boundaries (lines 7 & 8) and computes the position of C to them (lines 9 & 10). C is in the bounded area if two new segments are created (lines 11-14). If C is left to the visibility region, only one new segment will be created on $\overline{BC}$, and if C is right the segment will be created on $\overline{CA}$ (lines 15-20). Intersections on lines 13, 14, 17 and 20 must be evaluated only when relevant point A or B is not visible (otherwise A or B define the boundary and the intersections are equal to A or B). The procedure always returns at least one new visible segment.

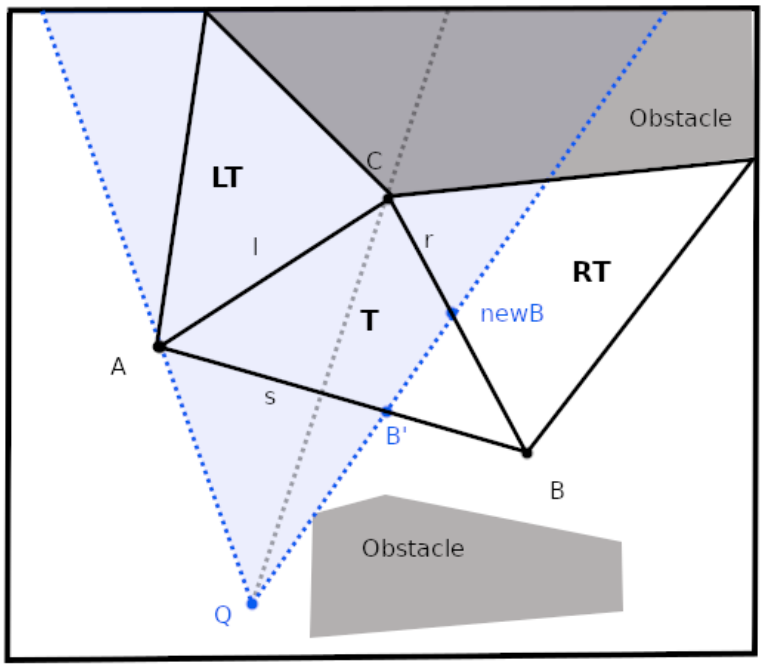


Figure 2.4: Expansion of visibility from Q to triangle T over segment s

Algorithm 3 TEA new segments

Inputs:
    Q: Query point.
    e: Expanded segment $e(L_e, R_e)$.
    T: Expanded triangle $T(A, B, C)$.
Output:
    S: Set of new expandable segments.

$rightBoundary \leftarrow \overrightarrow{QR_e}$
$leftBoundary \leftarrow \overrightarrow{QL_e}$

$rightOrient \leftarrow$ **winding**$(rightBoundary, C)$
$leftOrient \leftarrow$ **winding**$(leftBoundary, C)$

if $rightOrient == CCW$ **and** $leftOrient == CW$ **then**
    // C is visible
    $leftSegment \leftarrow [leftBoundary \cap \overline{CA}, C]$
    $rightSegment \leftarrow [rightBoundary \cap \overline{BC}, C]$

if $rightOrient == CCW$ **and** $leftOrient == CWW$ **then**
    // C is too left
    $rightSegment \leftarrow [rightBoundary \cap \overline{BC}, leftBoundary \cap \overline{BC}]$

if $rightOrient == CW$ **and** $leftOrient == CW$ **then**
    // C is too right
    $leftSegment \leftarrow [leftBoundary \cap \overline{CA}, rightBoundary \cap \overline{CA}]$

return leftSegment, rightSegment

2.4 Summary

The *triangular expansion algorithm* was described in this chapter. The algorithm is designed to compute visibility regions from a query point. Even though its time complexity is $O(n^2)$, in realistic scenarios, it performs in sub-linear time.

Chapter 3

Polygonal expansion algorithm

3.1 Motivation

The query computation time of TEA has a linear dependency on the number of searched triangles. The idea of *polyanya* authors for improving TEA is to merge triangles into convex polygons and expand visibility more efficiently. *Polygonal expansion algorithm* (PEA) works on a polygonal mesh, performs better than TEA, and opens the possibility of research for finding the best polygonal mesh.

The algorithm is the same as TEA except for expanding to a new polygon with up to n new edges. The initial polygon and edges can be obtained the same as the initial triangle in Section 2.3.1. Implementation of PEA is the same as in TEA Alg. 1 and Alg. 2 with the only difference in computation of new expandable segments (line 10 Alg. 2). This chapter describes only this procedure.

3.2 New expandable segments

The rules for visibility from point Q over segment s are in Section 2.3.2. When expanding over a segment, visibility is expanded only to the single polygon behind the segment. What is visible in this **expanded polygon** is computed from the intersection of the polygon geometry and visibility boundary. Two boundary lines of visibility given by looking from query point Q over the **expanded segment** are given as $\overrightarrow{QA'}$ and $\overrightarrow{QB'}$. The polygon's geometry is then divided into four categories of visibility. Every vertex of the expanded polygon can be located between boundary lines, in a counter-clockwise direction from both lines, in a clockwise direction from both lines, or it can be located directly on one of the boundary lines (further **Visible, Right, Left, Bordering** as in Fig. 3.1).

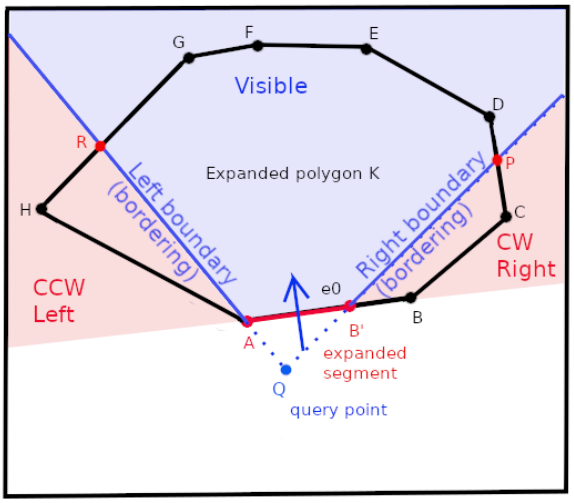


Figure 3.1: Visibility states defined by bounded visibility when looking over a segment $\overline{AB}$.

At first, the visibility state of each vertex is determined with the computation of orientation predicates to visibility boundary lines. Vertices are *visible* if they are oriented CCW to one line and CW to the other one. Vertices in the same direction to both lines must be either *left* (CCW) or *right* (CW) from the visibility region over the expanded segment. Vertices that are collinear with a boundary line (lie on the line) are visible but do not necessarily produce new expandable segments (as vertex A in Fig. 3.1). That is the reason for having *Bordering* visibility state instead of including such vertices in *Visible* state.

For example, vertices of polygon K from Fig. 3.1 are split into visible set $V(G, F, E)$, left set $L(I)$, right set $R(B, C, D)$ and bordering set $B(A)$.

After determining the visibility state of all vertices, new expandable segments can be produced. All edges of the expanded polygon except the edge containing the expanded segment are evaluated by the following set of rules:

- If both vertices defining the edge are *visible*, the whole edge is passed as expandable. (Edges $\overline{EF}$ and $\overline{FG}$ in Fig. 3.1.)
- If one vertex is *visible* and the second one is *bordering*, the whole edge is also passed as expandable.
- If one vertex is *visible* and the second one is *right*, the new segment is defined by the *visible* vertex and the intersection between the right boundary and the edge. (In Fig. 3.1, this applies to segment $\overline{PE}$.)
- If one vertex is *visible* and the second one is *left*, the segment is defined by the *visible* one and the intersection between the left boundary and the edge. (As segment $\overline{GR}$ in Fig. 3.1.)
- If one vertex is *left* and the second one is *right*, the new segment is defined by intersections of boundary lines with the edge.
- In case both vertices are *left* or *right*, the edge is not visible, and any part of it cannot be expanded.
- In case one vertex is *bordering* and the second one is *left* or *right*, the edge will be expanded only if the first vertex lies on the opposite boundary. (This is shown in Fig. 3.2 and applies for segment $\overline{HA}$ in Fig. 3.1.)

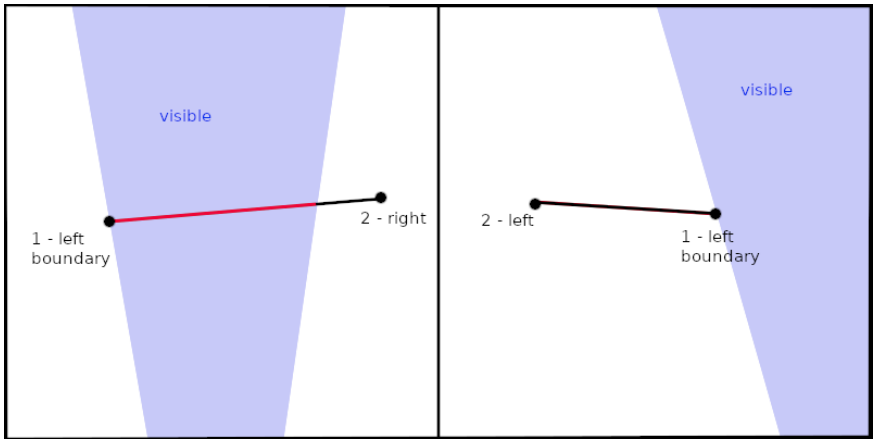


Figure 3.2: When one vertex is bordering and the second is not visible, the creation of a new segment depends on their mutual position.

If the expanded segment is traversable (is not an obstacle), at least one new expandable segment is always created.

3.2.1 Implementation

In the provided implementation, vertices of the expanded polygon are sorted counter-clockwise, starting with the right vertex of the edge on which the expanded segment (**expanded edge**) lies. Edges are also sorted CCW starting with the expanded edge. The indexing is shown in Fig. 3.3.

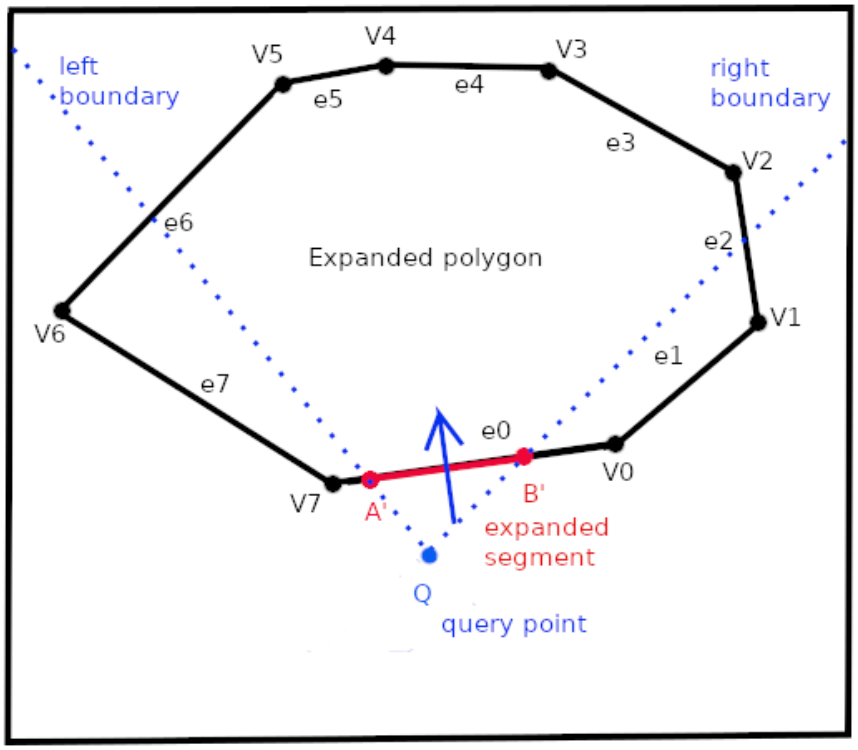


Figure 3.3: Indexing and nomenclature for forward expansion implementation.

Forward expansion can be implemented as described in Alg. 4. Boundaries of visibility region $\mathbf{V}^t(Q, s = (L_e, R_e))$ are $\overrightarrow{QL_e}$ and $\overrightarrow{QR_e}$ (lines 7 and 8). With these boundaries, the visibility state of each vertex in an expanded polygon is computed (lines 9 to 12). The set of rules listed above creates new expandable edges or their sub-segments, which are then pushed to a list passed for further expansion (lines 13 to 16).

Algorithm 4 PEA new segments

Inputs:
    Q: Query point.
    e: Expanded segment $e(L_e, R_e)$.
    P: Expanded polygon $P(V_0, V_1...V_n;\ e_0, e_1...e_n)$.
Output:
    S: Set of new expandable segments.

$rightBoundary \leftarrow \overrightarrow{QR_e}$
$leftBoundary \leftarrow \overrightarrow{QL_e}$

for $\forall V_i \in P$ **do**
    $rightOrient \leftarrow$ **winding**$(rightBoundary, V_i)$
    $leftOrient \leftarrow$ **winding**$(leftBoundary, V_i)$
    $V_i.state \leftarrow$ **getVisibilityState**$(rightOrient, leftOrient)$

for $\forall e_i \in P,\ i > 0$ **do**
    $visible \leftarrow$ **getVisibleSubsegment**(V_i, V_{i-1})
    if $\exists visible$ **then**
        $push : S \leftarrow visible$
return S

3.3 Summary

In this chapter, *polygonal expansion algorithm* is derived from *triangular expansion algorithm*. The PEA works on a polygonal mesh instead of a triangular one which allows passing large areas in one expansion instead of expanding to all triangles filling that area. In theory, PEA on triangles should perform the same as TEA, but the additional code needed for PEA slows it down in practice. Experiments show that PEA on polygons is at least 20% faster than TEA.

Chapter 4

Visibility from segment

4.1 Motivation

Computing visibility region from a segment opens new possibilities for multiple research areas. In robot localization and control, it provides a more precise sensor model. For example, if a robot drives in a sequence of linear motions, its observations do not have to be sampled to points. Instead, whole trajectory segments can be computed at once without approximation.

Visibility from a segment is also beneficial for estimating an ideal mesh pre-processing for TEA in the MinVT algorithm [MK22], where visibility from an edge is estimated by sampling the edge with points and computing the visibility region for each of them. Computed visibility from the segment entirely removes this need for approximation by sampling.

In surveillance applications, computation of visibility from a segment can simplify decisions where to put cameras to see the whole entrance or a wall with art pieces.

Last but not least, the new computation of visibility from a point introduced in this paper uses visibility from segments as its pre-processed structure.

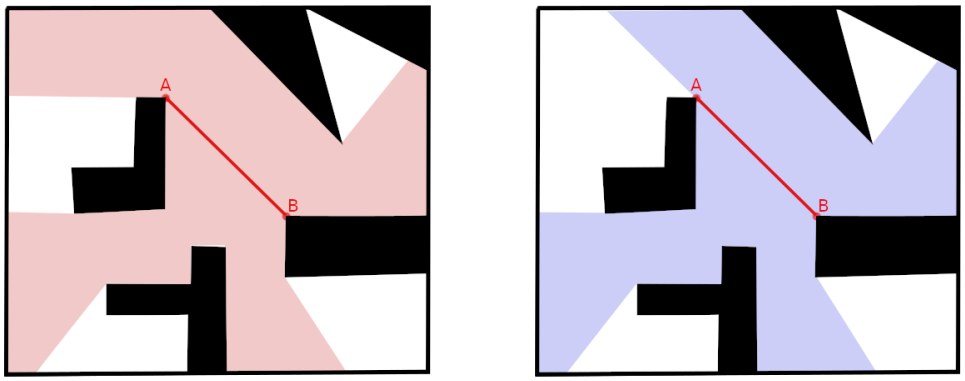
4.2 Definition

The visibility region of a segment can be separated into two categories. The first one can either look for all points that are visible at least from one point on the segment (sensor model during motion) and the second one looks only for points that are visible from all points on the segment (surveillance). The focus of this research is on the former.

Visible region V from closed line segment s in polygon $P \subset R^2$ is defined as a set of all points that are visible from at least one point on the segment:

$$\forall X \in V(s) : \exists Y \in s, \overline{YX} \subset P \tag{4.1}$$

.



(a) : Visibility of segment $\overline{AB}$. **(b) :** Reduced visibility of segment $\overline{AB}$.

Figure 4.1: Visibility of segment $\overline{AB}$ by two definitions.

A visibility region of a segment can be seen in Fig. 4.1a. The visibility region of a point on a segment changes continuously as the point moves along the segment (even when the visibility expands to a new area, the new visible area is only a line at first and then continuously grows). This continuity can be broken only if an obstacle touches the line. In the example in Fig. 4.2 the continuity is broken in the middle of a segment, but a similar situation can occur on segment endpoints as in Fig. 4.1. We provide an alternative definition Eq. 4.2 which handles this discrete behaviour for the endpoints. If the touch is between the endpoints, the segment can be split into two at the point of touch.

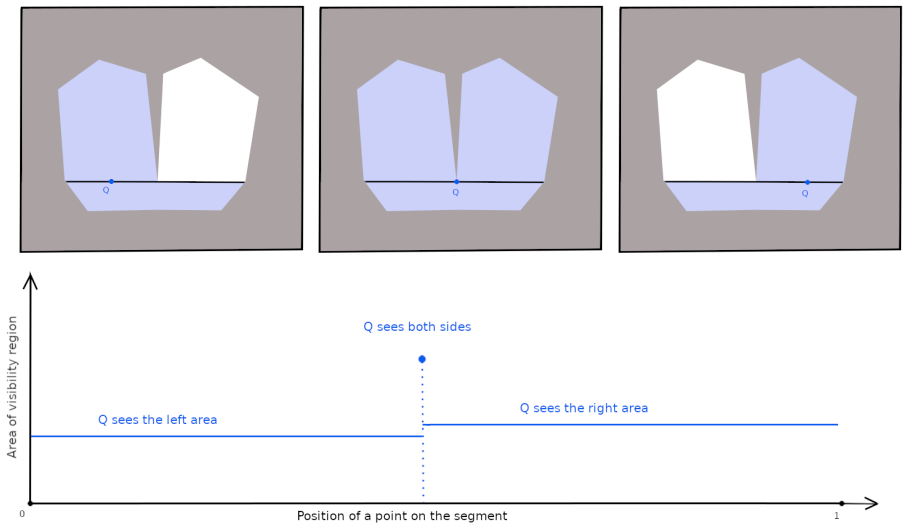


Figure 4.2: If an obstacle is touching the segment, the area of visibility region of that segment changes discretely in the point of touch.

The alternative visibility is shown in Fig. 4.1b. Reduced visibility region V' of closed segment s in polygon $P \subset R^2$ is defined as a set of all points that are visible from at least one point on the open segment s':

$$s = [A, B], s' = (A, B),$$
$$\forall X \in V'(s) : \exists Y \in s', \overline{YX} \subset P \tag{4.2}$$

. The algorithm introduced in this chapter uses the reduced visibility region definition.

4.3 PEA-E introduction

The new proposed algorithm computes reduced visibility from segments in polygon subdivided into a set of convex polygons (**polygonal mesh**). It is called the polygonal expansion algorithm for edges (**PEA-E**). The algorithm gradually expands visibility through edges in mesh until it reaches obstacles in all directions.

The general schema of the PEA-E is presented in Alg. 5. At first, the polygon representing the environment is transformed into a polygonal mesh (line 6). Then, after a **mother segment** for which visibility should be computed is given, initial edges from the mesh are prepared for the expansion (line 7).

Algorithm 5 PEA-E

1: **Inputs:**
2: s: Segment for which visibility is computed.
3: P: Polygon representing environment.
4: **Output:**
5: *visibility*: Reduced visibility region of s.
6: **Initialization:** $mesh \leftarrow CDT(P)$
7: $expandable \leftarrow getInitialEdges(s, mesh)$
8: **for** e in expandable **do**
9: $visibility \leftarrow$ **Expand**$(s,\ e,\ mesh)$
return *visibility*

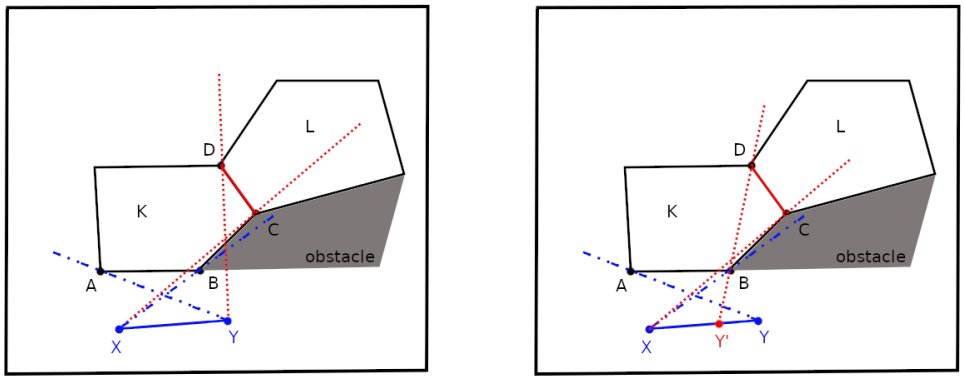
The initial edges are all edges of the mesh that are visible from the mother segment without looking over any segment in the mesh (they are directly visible and will be the first segment to look over). Then, for each initial edge, a recursive procedure expanding further until it hits an obstacle is launched (lines 8 & 9).

Algorithm 6 PEA-E

1: **procedure Expand**
2: **Inputs:**
3: s: Segment for which visibility is computed.
4: e: Segment through which is visibility expanded.
5: *mesh*: Environment represented as polygonal mesh.
6: **Output:**
7: *visibility*: Points defining area in mesh visible from s over e.
8: **if** e is not traversable **then**
9: **return** $visibility \leftarrow e$
10: *newExpandable* $\leftarrow$ *Forward(e,s,mesh)*
11: **for** newE in newExpandable **do**
12: Backward(newE)
13: **push:** $visibility \leftarrow$ **Expand**$(node)$ *# Next level of recursion*

Alg. 6 describes a recursive procedure that gradually expands through the polygonal mesh and results in a visibility region from the segment over an initial edge in the mesh. Before expanding, the algorithm checks whether the expanded segment is traversable (line 8). Non-traversable segments lead to obstacles, so they define the end of the visibility region (line 9). Traversable segments lead to unexplored polygons in mesh and produce at least one new segment for further expansion (line 10). Unlike using expansion algorithms for visibility from a point, this algorithm computes backward visibility (**root**) of new segments (line 11).

If this step was skipped, the algorithm would try to look over expanded segments from points that cannot see the segment. The cause of this is the absence of mutual visibility. By definition of reduced visibility Eq. 4.2, visibility of the whole area from a segment does not guarantee the visibility of the whole segment from the area. Fig. 4.3 shows that without computation of the root, the algorithm could look through obstacles.



(a) : Without computed roots. **(b) :** With computed roots.

Figure 4.3: Example of expansion visibility region from segment with and without recomputed roots.

Once the new segment has its root computed, it can be recursively expanded (line 13). Fig. 4.4 shows that the algorithm proceeds as a depth-first search where leaves are segments leading to obstacles. The algorithm first expands the blue path visible through initial blue edge $\overline{AD}$ to the end (5) and then proceeds over initial green edge $\overline{BC}$ which splits to three branches. Red arrows show where the algorithm evaluates non-traversable segments (Alg. 6: line 8,9).

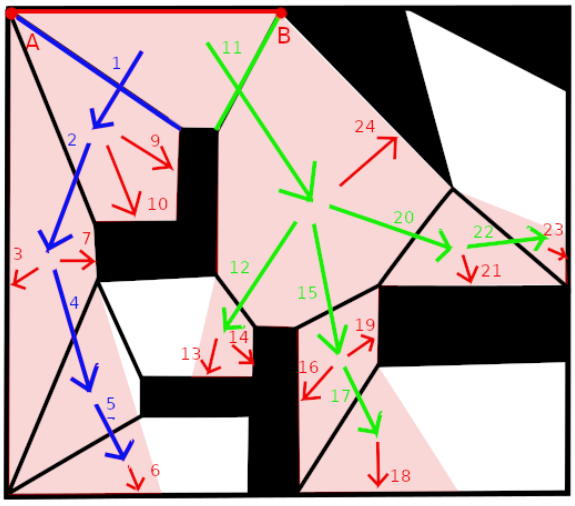
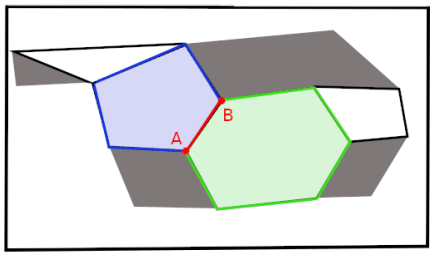


Figure 4.4: PEA-E proceeds as depth-first search.

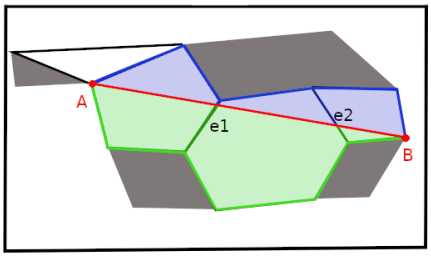
4.4 Initial edges

This section describes a search for initial edges which can be expanded further (line 7 in Alg. 5).
To start the expansion, the algorithm first needs to find first expandable edges. Those are all edges wholly visible from the mother segment. Also, these new edges can't intersect the mother segment except by sharing the endpoints. Any other intersection would cause some segments to look over themselves during forward expansion or during computation of root which is not defined.

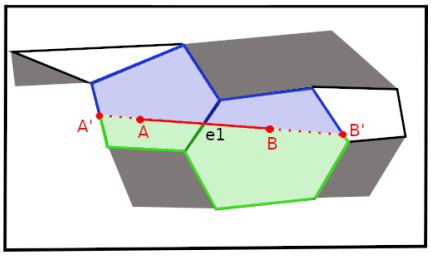
The computation of initial edges depends on the definition of the mother segment. The easiest case is mother segments defined with an edge of the mesh because then the directly visible area is just the neighbouring polygons (Fig. 4.5a). If we define the mother segment with mesh vertices, the situation is more complicated because it can pass through multiple polygons (Fig. 4.5b). The hardest case is when the mother segment is completely arbitrary because it must be somehow connected to the mesh structure (Fig. 4.5c). We've implemented the first two cases, but in this thesis only the first and least complicated one is relevant.



(a) : Mother segment defined with a mesh edge.



(b) : Mother segment defined with two mesh vertices.



(c) : Mother segment defined with arbitrary points.

Figure 4.5: Possible definitions of mother segment in a polygonal mesh.

The algorithm computes the visibility region from mother segment m separately on each half-plane defined by line b, where

$$m \subset b. \tag{4.3}$$

Both half-planes include boundary line b, so resulting visibility regions might share a visible segment.

If m is given by an edge on the mesh, it can have up to two neighbouring polygons. All polygons in the mesh are convex which guarantees visibility between all points in the mesh including its edges. Initial expandable edges are then all edges e_i of the neighbouring polygons for which

$$m \not\subseteq e_i. \tag{4.4}$$

With definition Eq. 4.2, where the mother segment endpoints are omitted rule Eq. 4.4 changes to

$$\forall X \in m : X \notin e_i. \tag{4.5}$$

The common initial root of these edges is whole m.

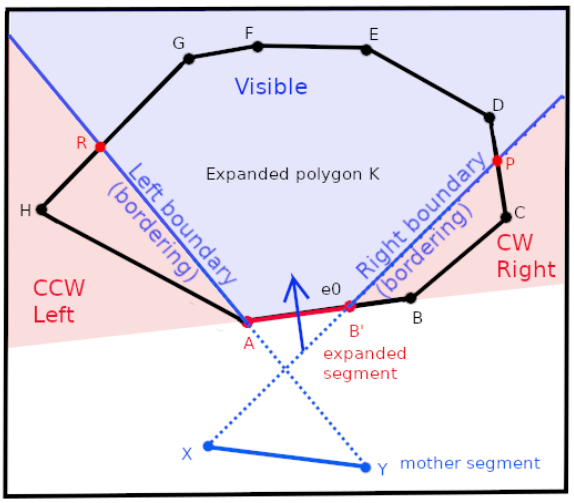
Mother segments defined by two vertices on the mesh can pass through multiple polygons as in Fig. 4.5b. All edges of polygons crossed by m that fulfil rule Eq. 4.5 are defined as initial edges (edges that do not share any points with the mother segment). Initial roots can differ and must be computed by evaluating the visibility of the initial edge on crossed polygons (the root of every edge is a subset of m visible from the edge).

Initialization for an arbitrary segment was not tested, so we give only a proposal. The segment is prolonged until it touches a mesh edge. A new temporary vertex is created in this virtual intersection splitting the edge in two. Initial edges and roots are then obtained the same as when the mother segment is defined by vertices (virtual ones in this case). Initial roots are then constrained to the original segment. The creation of virtual vertices is in Fig.4.5c.

Once all initial edges and their roots are prepared, the algorithm can expand visibility over them.

4.5 Forward expansion

Forward expansion for PEA-E (line 10 in Alg. 6) is derived from the same procedure for PEA in Section 3.2 (For PEA it is the only core procedure, but PEA-E also recomputes roots). The only difference in the whole process of obtaining new visible segments is that looking over segment s from mother segment m creates different visibility boundaries than looking from point Q over s. The similarity is clear when comparing Fig. 3.1 with Fig. 4.6.



.

Figure 4.6: Visibility states defined by bounded visibility when looking from mother segment $\overline{XY}$ over a segment $\overline{AB}$

At first, the visibility boundaries are computed in a specific way for visibility from one segment over another segment. Then PEA-E determines visibility states for all vertices in the expanded polygon and by the same rules as PEA defines new visible segments. This section describes how visibility boundaries should be obtained and what changed in the implementation.

4.5.1 Visibility boundaries

Visibility boundaries are needed for evaluating which parts of the expanded polygon are visible.

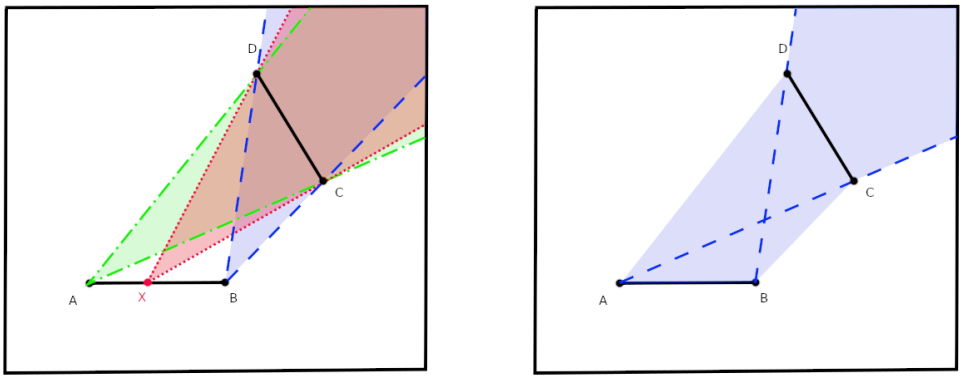
Two points X, Y in polygon P are mutually visible through segment s if

$$\overline{XY} \subset P \wedge \overline{XY} \cap s \neq \varnothing. \tag{4.6}$$

From Eq. 4.6 and Eq. 4.1 follows that visibility from one segment s over another segment e ($V^t(s,e)$) can be defined as

$$\forall X \in V^t(s,e) : \exists Y \in s, \overline{XY} \subset P \wedge \overline{XY} \cap s \neq \varnothing. \tag{4.7}$$

This definition also applies to visibility from a point over an edge since segment s can degenerate to a single point. If all points on segment s see at least one point on another segment e, visibility from s over e is a union of visibility regions from all points $Y_i \in s$ over e.



(a) : Looking over the segment from points.

(b) : Looking over the segment from a segment.

Figure 4.7: Visibility regions from points A, B, X and from segment $\overline{AB}$ when looking over segment $\overline{CD}$.

In Fig 4.7a, both boundary lines for visibility from a single point X over segment $\overline{CD}$ begin with X and go through points C and D. These boundaries define visible cone $\measuredangle\overleftrightarrow{CXD}$.
To satisfy Eq. 4.7, the cone part in the same half-plane as X (defined with line $\overline{CD}$) should not be included. However, introduced algorithms work only in the other half-plane the segment so these simple boundaries are kept.

Moving with point X pivots boundary lines around points C and D. If segments $\overline{AB}$ and $\overline{CD}$ can define convex tetragon $t(A,B,C,D)$, unifying visibility from $\forall X \in \overline{AB}$ over $\overline{CD}$ results in visibility from $\overline{AB}$ over $\overline{CD}$. This can be shown by starting with $X = A$ and moving X to B. This movement causes boundary line $\overrightarrow{XC}$ to rotate only in a counter-clockwise direction. When moving X from B to A, boundary $\overrightarrow{XD}$ can rotate only in a clockwise direction. Combining both observations results in boundaries $\overrightarrow{AC}$ and $\overrightarrow{BD}$ for visibility from segment $\overline{AB}$ as in Fig. 4.7b.

4.5.2 Implementation

In the provided implementation, vertices and edges of the expanded polygon are sorted counter-clockwise, starting with the right vertex of the edge on which lies the expanded segment as in PEA. Because the mother and expanded segments have to define convex tetragon (Section 4.5.1), we can define the right and the left point of both segments relative to the direction of expansion (in Fig. 4.8 points A' and X are left and points B' and Y are right.

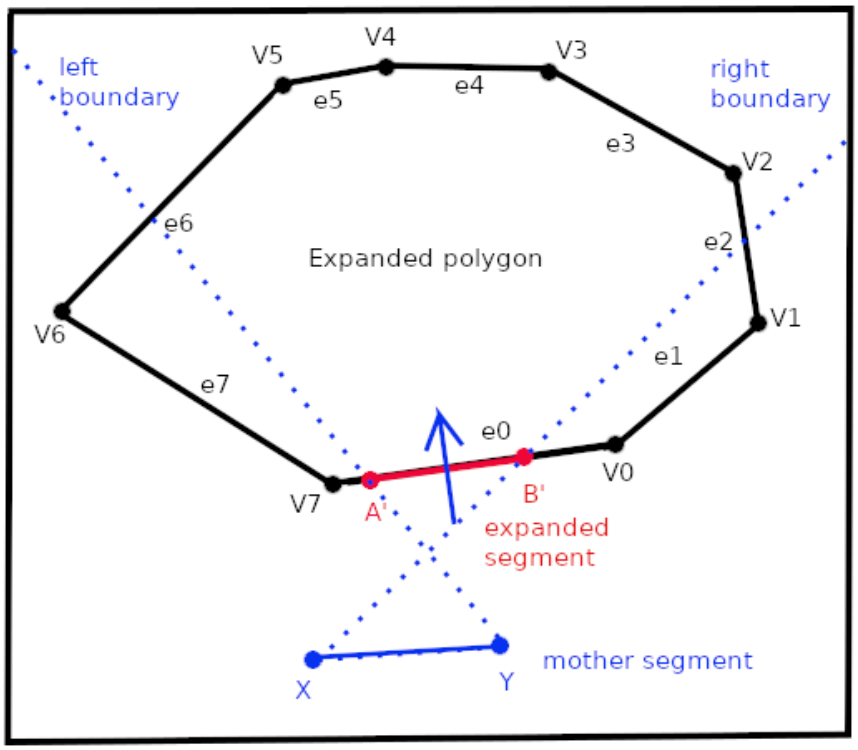


Figure 4.8: Indexing and nomenclature for forward expansion implementation.

In comparison with PEA (Alg. 4), PEA-E has only two different lines defining the boundary lines (lines 7 & 8 in Alg. 7. Before expanding the new segments, their backward visibility has to be computed to satisfy the rule at the beginning of Section 4.5.1. This rule demands that all points from the mother segment see at least one point from the expanded segment. Newly generated expandable segments do not always satisfy this rule.

Algorithm 7 PEA-E Forward Expansion

Inputs:
    s: Mother segment $s(L_s, R_s)$.
    e: Expanded segment $e(L_e, R_e)$.
    P: Expanded polygon $P(V_0, V_1 ... V_n;\ e_0, e_1 ... e_n)$.
Output:
    S: Set of new expandable segments.

$rightBoundary \leftarrow \overline{L_s R_e}$
$leftBoundary \leftarrow \overline{R_s L_e}$

for $\forall V_i \in P$ **do**
    $rightOrient \leftarrow$ **winding**$(rightBoundary, V_i)$
    $leftOrient \leftarrow$ **winding**$(leftBoundary, V_i)$
    $V_i.state \leftarrow$ **getVisibilityState**$(rightOrient, leftOrient)$

for $\forall e_i \in P,\ i > 0$ **do**
    $visible \leftarrow$ **getVisibleSubsegment**(V_i, V_{i-1})
    if $\exists visible$ **then**
        $push : S \leftarrow visible$
return S

4.6 Backward visibility

This section describes the computation of roots for new segments generated in forward expansion. Root r_s is a part of mother edge m which is wholly visible from segment s (line 12 in Alg. 6):

$$r_s = \{Y \in m | Y \in \mathbf{V}(s)\}. \tag{4.8}$$

Visibility of point X from point Q guarantees visibility of Q from X. In other words, they are mutually visible. Due to this, all points in $\mathbf{V}(Q)$ are mutually visible with Q:

$$\forall X \in \mathbf{V}(Q) : Q \in \mathbf{V}(X). \tag{4.9}$$

Because segments are a set of points, Eq. 4.9 should change

$$\forall X \in \mathbf{V}(s) : s \subset \mathbf{V}(X). \tag{4.10}$$

Definitions of visibility from a segment in Section 4.2 do not guarantee that all points on s see point $X \in \mathbf{V}(s)$, so there may be point $Y \in s$ such that $X \notin \mathbf{V}(Y)$. From Eq. 4.9 can be deduced that if X is not visible from Y, Y cannot be visible from X. If such point Y exists, it breaks Eq. 4.10. To be valid, Eq. 4.10 has to be modified to

$$\forall X \in \mathbf{V}(s) : \exists r \subset s, r \subset \mathbf{V}(X), \tag{4.11}$$

where r is the same as root r_s in Eq. 4.8. In other words, visibility between a point and a segment is not mutual. (It is partially mutual, meaning there has to be mutually visible subsegment, but we have to compute it.)

Fig. 4.3a shows that without computation of roots, boundary $\overrightarrow{YD}$ would intersect the obstacle. The visibility region $\mathbf{V}^t(\overline{XY}, \overline{CD}$ is then computed larger than it should be and the algorithm will produce a wrong visibility region. Correct expansion over the new segment with new boundary $\overrightarrow{Y'D}$ is shown in Fig. 4.3b.

4.6.1 Computing root

According to Eq. 4.7, new segment e is an edge or segment visible from at least one point on segment m over segment s. Eq. 4.8 for the root of e can be interpreted as a set of all points on m from which e is visible. It follows that new root r_e is visible from e through s:

$$e \subset V^t(m, s) \Longrightarrow r_e \subset V^t(e, s), \tag{4.12}$$

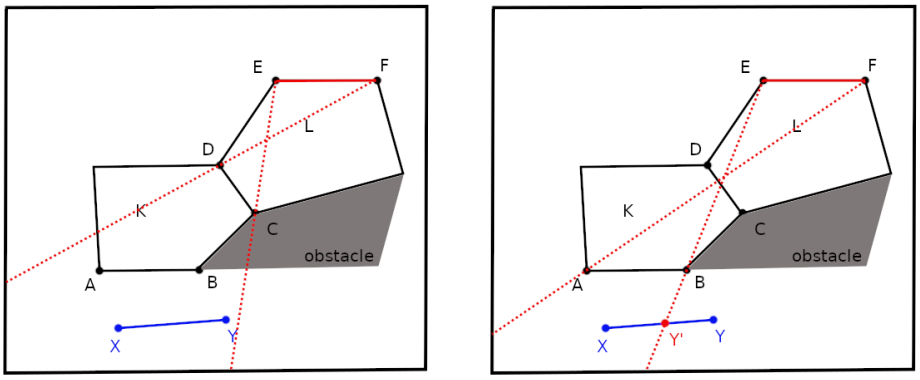
where $V^t(m, s)$ is visibility region from m through s. The algorithm computes visibility from mother segment m over segments e_i with roots r_i. This limits roots on m as any other points are undesirable.

$$r_i \subseteq r_{i-1} \subseteq \cdots \subseteq r_0 \subseteq m \tag{4.13}$$

From Eq. 4.12 and Eq. 4.13, root can be computed as the intersection of the previous root and backward visibility region:

$$r_e = r \cap V^t(e, s). \tag{4.14}$$

Fig. 4.9a shows that looking back over the last expanded segment is insufficient because it can include obstacles to the backward visibility region.



(a) : Looking back over the immediate predecessor.

(b) : Looking back over the further predecessor.

Figure 4.9: Looking back over the last segment is not enough since it may cause looking at obstacles (first image). Backward visibility can be limited by any segment from the previous expansion (second image).

Because new segments from forward expansion have to be visible through all previously expanded segments, the roots of new segments have to be also visible through all previously expanded segments (the segment and its root are mutually visible). This changes Eq. 4.12 to

$$r_i \subset V^t(e_i, e_{i-1}) \wedge r_i \subset V^t(e_i, e_{i-2}) \wedge \cdots \wedge r_i \subset V^t(e_i, e_0) \tag{4.15}$$

which with 4.14 gives equation

$$r_i = r_{i-1} \cap V^t(e_i, e_{i-1}) \cap V^t(e_i, e_{i-2}) \cap \cdots \cap V^t(e_i, e_0). \tag{4.16}$$

To compute a new root as in Eq. 4.16, the algorithm looks back through all previously expanded segments and remembers the most limiting boundaries as in Fig. 4.9b.

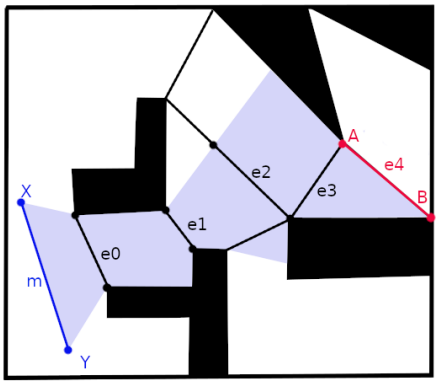
4.6.2 Implementation

Once a new expandable segment e_i from forward expansion in Alg. 7 is provided, a new root must be computed by looking back. New segments should carry a reference to their predecessor for easier access. The algorithm computes the area of backward visibility over all predecessors and remembers the most limiting boundaries (a different predecessor can define each boundary). A part of the previous root, visible in the most strict boundaries, is saved as the new root. Fig. 4.10 shows an example with four predecessors.

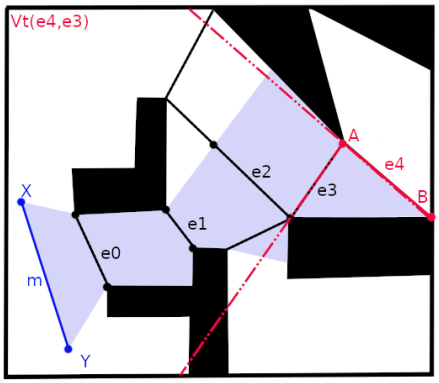
New expandable segment e is passed to Alg. 8. At first, the new root is set equal to the last one because $r_i \subseteq r_{i-1}$ in 4.13 (line 6, in Fig. 4.10a $r_e = \overline{XY}$). The algorithm iterates through all previous segments e_i always by getting a reference to the predecessor of the currently evaluated one (lines 7-9). If boundaries of back visibility $V^t(e, e_i)$ (lines 10-13) intersect current r_e, it is limited by these boundaries (lines 14-18).
Four cycles of the loop on lines 8 to 18 are depicted in images $(b) - (e)$ in Fig. 4.10. The new root is limited in Fig. 4.10d, but boundaries in Fig. 4.10e are even more strict. The example results in a new root $\overline{X''Y''}$. Looking from the root over its segment $\overline{AB}$ generates new segment $\overline{BC}$ as in Fig. 4.10f.

Algorithm 8 PEA-E Root evaluation

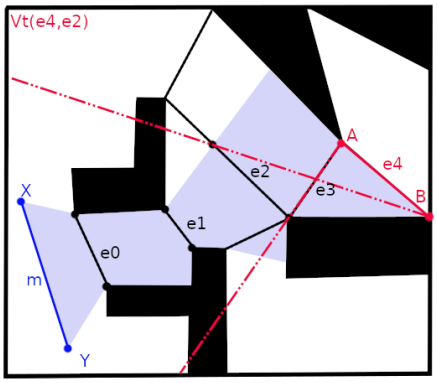
1: **Inputs:**
2: r: Root of previous segment $s(L_r, R_r)$.
3: e: New segment $e(L_e, R_e)$.
4: **Output:**
5: r_e: New root for e ($r_e \subseteq r$).
6: $r_e \leftarrow r$
7: $current \leftarrow e$
8: **while** $\exists\, current.predecessor$ **do**
9: $current \leftarrow current.predecessor$
10: $[R, L] \leftarrow current$
11: $[X, Y] \leftarrow r_e$
12: $rightBoundary \leftarrow \overline{L_e R}$
13: $leftBoundary \leftarrow \overline{R_e L}$
14: **if** $leftBounary \cap r_e \neq \varnothing$ **then**
15: $X \leftarrow leftBoundary \cap r_e$
16: **if** $rightBounary \cap r_e \neq \varnothing$ **then**
17: $Y \leftarrow rightBoundary \cap r_e$
18: $r_e \leftarrow [X, Y]$
return r_e



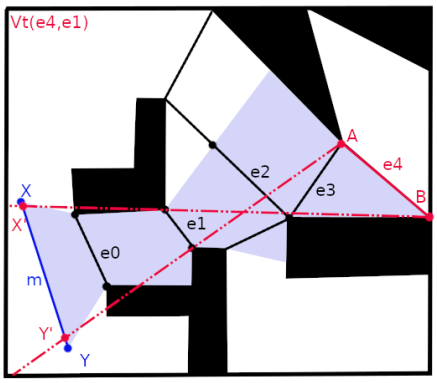
(a) : New segment $\overline{AB}$ from forward expansion.



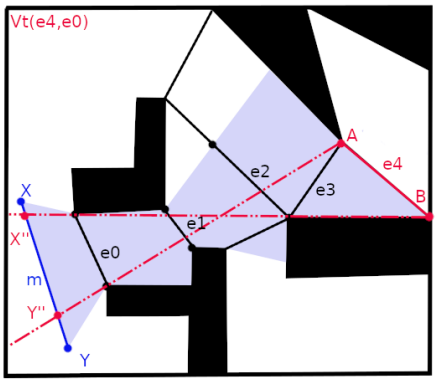
(b) : Looking back over segment e_3 does not create any limits for root: $r_4 = m$.



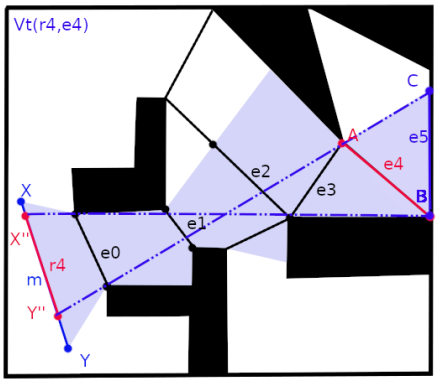
(c) : Looking back over segment e_2 does not create any limits for root: $r_4 = m$.



(d) : Looking back over segment e_1 limits root from both sides: $r_4 = \overline{X'Y'}$.



(e) : Looking back over segment e_0 limits root, even more, resulting in final root: $r_4 = \overline{X''Y''}$.



(f) : With new root r_4 algorithm can expand segment $\overline{AB}$.

Figure 4.10: Example of how algorithm computes root for new expandable segment $\overline{AB}$ and expands it for new segment $\overline{BC}$.

4.7 Numerical stability

During expansion, new subsegments can be defined with an intersection between an edge in the mesh and a boundary line. Evaluation of intersection belongs to the basics of analytic geometry. Given two lines in R^2 defined as $p : a_p x + b_p y + c_p = 0$ and $r : a_r x + b_r y + c_r = 0$ point of their intersection I can be computed as

$$\begin{aligned} I &= [x_I, y_I], \\ a_p x_I + b_p y_I + c_p &= 0, \\ a_r x_I + b_r y_I + c_r &= 0, \\ &\vdots \\ x_I &= \frac{b_p c_r - b_r c_p}{a_p b_r - a_r b_p}, \qquad (4.17) \\ y_I &= \frac{c_p a_r - c_r a_p}{a_p b_r - a_r b_p}. \qquad (4.18) \end{aligned}$$

Roundoff errors of computers cause the computed intersection of two lines to be only a point near the real intersection. No robust and fast implementation for the evaluation of intersections has yet been produced. Usage of intersections burdened by rounding error results in incorrectly computed visibility regions.

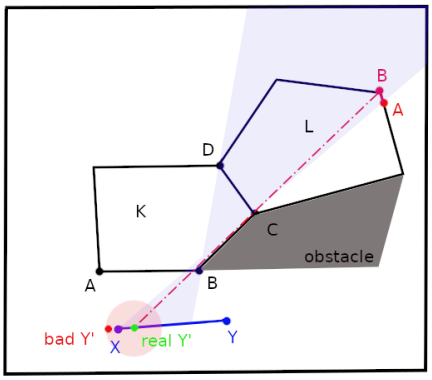


Figure 4.11: Numerical errors in intersection evaluation can cause very strange results as looking through obstacles or computing visibility from different segments than the mother segment.

Apart from errors in the resulting visibility region, some incorrectly evaluated intersections can cause impossible situations as an exchange of root points depicted in Fig. 4.11. In the example, the expansion to polygon L resulted in a new short segment $\overline{AB}$. The right point of new root Y' is then poorly evaluated due to numerical error and placed left to the right root point X which breaks the Eq. 4.13 (new root has to be a subset of the old root).

Our first implemented version of PEA-E failed for 30% edges on meshes from the IronHarvest dataset [HHJ22]. Numerical errors created too unrealistic situations.

To avoid possibly incorrect evaluation of intersections, a new abstract representation of intersection points is introduced. Intersections occur only when the visibility region is limited by the environment in form of an obstacle vertex. One root point and the obstacle vertex create a boundary line which together with some mesh edge define an intersection.

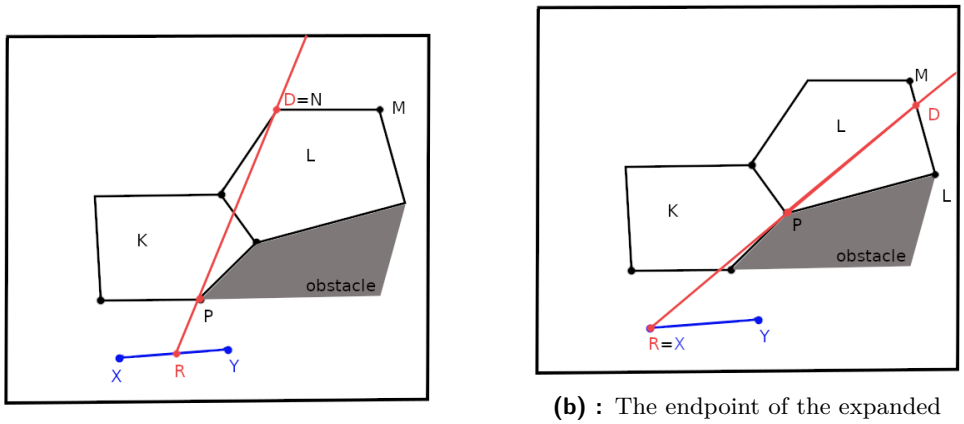
The new representation stores four mesh vertices defining the mentioned line and the intersected edge. Because vertices are a part of the mesh no numerical error is introduced by them. The algorithm presented so far builds on computed intersections and thus must be adapted for the new representation of intersections.

4.7.1 Stable forward expansion

In forward expansion, boundary lines are defined by starting point on the root ($R \in r$) and the second (directional) point on the expanded edge ($D \in e_i$). R and D can be either vertex or intersection. Because intersections are not evaluated, both boundaries must be defined by some vertices. Boundaries are computed separately with the same rules, so changes can be shown on one general boundary $\overrightarrow{RD}$.

If both R and D are defined by a vertex, a boundary can be specified directly as $b = \overrightarrow{RD}$. If one of the points is an intersection, the boundary line is defined with the line defining the intersection. Fig. 4.12a shows where root point R was computed as $\overrightarrow{NP} \cap \overline{XY}$ after creating a new segment with vertex N. In the next expansion $D = N$ (meaning D is a vertex), boundary $\overrightarrow{RD}$ is defined as $\overrightarrow{PD}$ respectively $\overrightarrow{PN}$ where both P and N are mesh vertices. The same goes for Fig. 4.12b, where D is an intersection, and R is a vertex.

If both R and D are intersections, either one had to be defined as an intersection first. In case R was first defined as an intersection (Fig. 4.13), an expansion that generated D was bounded by a line defining R, and thus D is also defined by the same line. When D was defined as an intersection first, it was defined by the old root vertex and some vertex from the environment. The boundary lines are a subset of the expansion visibility region, which is visible only from the endpoints of the root (Section 4.5.1).



(a) : The root is an intersection.

(b) : The endpoint of the expanded segment is an intersection.

Figure 4.12: When only one point that should define visibility boundary is an intersection, the boundary can be defined by the line defining the intersection.

Because D is directly on the boundary line, it is visible only from R, so it cannot produce any new root defined as an intersection and R must be a vertex.
Both lines defining intersections R and D have to be equal and can be used as new boundaries.

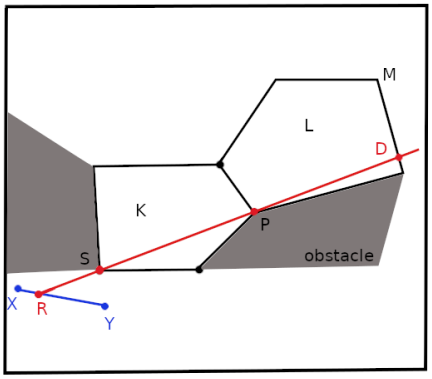


Figure 4.13: If both points defining the boundary are intersections, the boundary can be defined with a line defining the point lying on the mother segment.

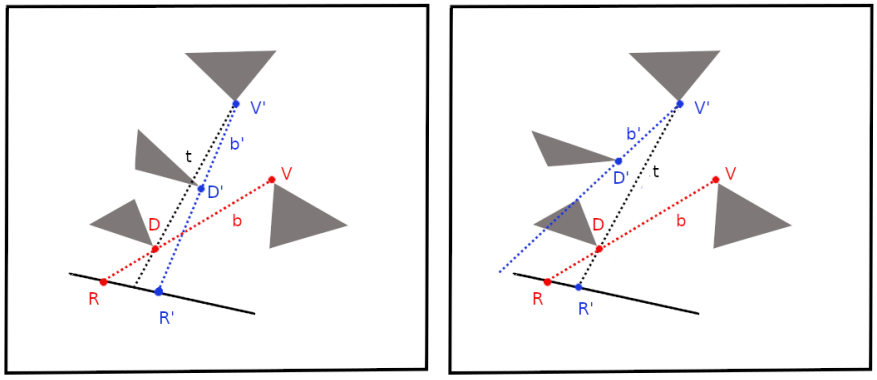
4.7.2 Stable root computation

During root computation, boundaries of back visibility can be defined with a point on the new segment D and a point on one of the previous expanded segments E_i. In addition, there is root point R, which can either stay the same or be shifted by backward boundaries. All three points can be either vertices or intersections.

In the unstable implementation, the position of R relative to backward boundaries was computed by passing all three points to orientation evaluation, which stated if R is limited by the boundary or not. With the alternative representation of intersections, it is not possible.

All cases where E_i is an intersection can be ignored because if E_i is an intersection, it was limited by some previous expansion E_{i-n}, which was a vertex (vertices introduce limitations, intersections only pass them further). This means only cases where E_i is a vertex are relevant. Further, if D is an intersection, it was created by the boundary line defined by the last root. The end of the last Section 4.7.1 shows that if D is an intersection, it cannot change the root point. This paragraph results in backward boundaries always defined by vertices (otherwise, they are irrelevant). R defined as vertex can be passed to winding which decides about new R'.

If R is an intersection, its defining line is most likely not equal to the current backward boundary $\overline{DE_i}$, and they don't even share a defining vertex. In Fig. 4.14 is old intersection R defined with line $b = \overline{VD}$ and the new R' with line $b' = \overline{V'D'}$. Whether backward boundary $\overrightarrow{V'D'}$ limits the root can be decided by the position of D' to new helping line $\overrightarrow{V'D}$. From the end of the previous Section 4.7.1 follows that D had to be obtained during looking back, which means it must be used for defining boundary when computing the root of V'. This reduces the problem to a question of what boundary is more strict. When the left root is resolved as in Fig. 4.14, counter-clockwise winding of points V', D, D' means $\overline{V'D'}$ is more strict than $\overline{VD}$ and new root R' will be created by line $\overline{V'D'}$ (Fig 4.14a). If the winding is clockwise or the points are in line, new root R' is also created, but with the line, $\overline{V'D}$, because D is a vertex of a more limiting obstacle (Fig 4.14b).



(a) : Evaluated obstacle D limits back visibility and creates new R'.

(b) : Evaluated obstacle D does not limit back visibility and R' will be created by further obstacle.

Figure 4.14: The only differently evaluated case in root computation can occur if the previous root was an intersection.

4.8 Summary

In this chapter, a visibility region from a segment was defined as a set of all points that are visible from at least one point in the segment. A more practical definition of reduced visibility from a segment was introduced to cut out undesirable bending behind corners. The polygonal expansion algorithm for edges was introduced as a way to compute reduced visibility region from a segment on a polygonal mesh. As numerical errors of computers can harshly affect new algorithms, a more robust implementation was proposed.

Chapter 5

EdgeVis

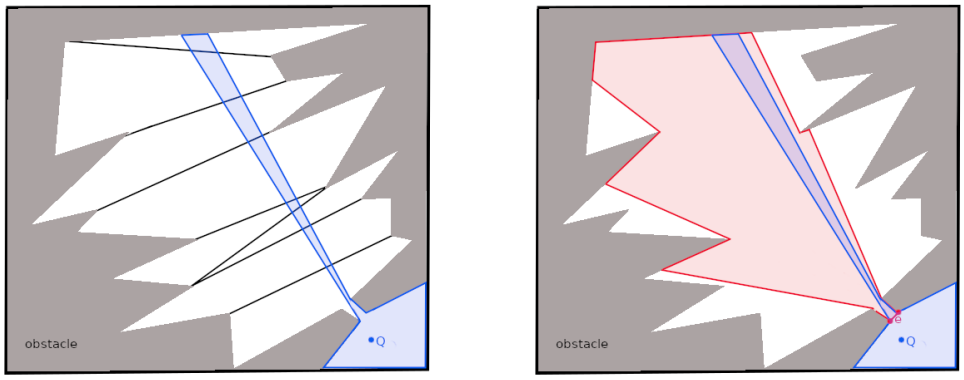
5.1 Motivation

The performance of TEA and PEA depends on how many triangles or polygons they have passed during the computation. Looking through narrow gaps creates small but long visibility regions. If the environment is complex enough, the algorithms must expand through many polygons to reach an obstacle, as in Fig. 5.1a. The idea behind the new algorithm *EdgeVis* is to precompute visibility regions of all edges in the mesh and use them as a structure for the computation of visibility regions from query points. When a query point is given, the algorithm locates its initial polygon as TEA or PEA. Instead of expanding the visibility over initial edges, the precomputed visibility from the edge is used as a reduction for the environment polygon W. Three methods for finding the visibility region of Q over the initial edge are discussed in this chapter. An example reduction of environment complexity is in Fig. 5.1b.

This chapter first shows that visibility from a point over a segment $\mathbf{V}^t(Q, s)$ is bounded by the visibility region of the segment $\mathbf{V}(s)$ (only one side is sufficient):

$$\mathbf{V}^t(Q, s) \subset \mathbf{V}(s). \tag{5.1}$$

All discussed methods work better on a triangular mesh. The basic method is to compute visibility with a query point for each vertex defining $\mathbf{V}(s)$. It is a very naive approach but performs similarly to PEA on polygons.



(a) : Small visibility region from Q can cross many polygons in complex environment.

(b) : Precomputed visibility from edge e decrease complexity of the environment.

Figure 5.1: Comparison of complexities when visibility is computed in polygonal mesh or on visibility region of visible edge.

The first improved method is to precompute all points that are visible from any point in the entrance triangle. Visibility for edges in the mesh is solved separately for both sides, and the entrance triangle is a triangle on the other side of the edge. Only $\mathbf{V}(s)$, which entrance triangle is the same as the query initial triangle, are evaluated. Because being always visible is quite a hard condition, only a few vertices are marked as always visible and do not have to be evaluated. This optimization brings around 5% speed up.
The last discussed method is cutting off surely not visible parts of $\mathbf{V}(s)$ online. For all vertices of $\mathbf{V}(s)$, the algorithm precomputes the root as in PEA-E. When a query is given, the naive method is followed until a vertex is not visible. Then, a boundary line is created from the last visible vertex and query Q, and its intersection R with s is evaluated. The not-visible vertex has its whole root next to R (can be both sides). Until a vertex with a root containing R or even with a root on the other side of R is reached, all vertices are marked not visible and can be skipped. The method, as described, adds around 5% speed up. The low improvement is caused by the high cost of pruning triggers and the possibility of starting pruning even for one not visible vertex.

5.2 Bounding by edge visibility

Let us have some observable space $\mathbf{W}$, a segment s and point Q where $s \in \mathbf{W}$ and $Q \in \mathbf{W}$. Visibility of Q over s can be defined as

$$\mathbf{V}^t(Q, s) = \{\forall X \in \mathbf{W} | \overline{QX} \subset \mathbf{W} \wedge \overline{QX} \cap s \neq \varnothing\}, \tag{5.2}$$

and the visibility region of s can be defined as

$$\mathbf{V}(s) = \{\forall Y \in \mathbf{W} | \exists Z \in s : \overline{ZY} \subset \mathbf{W}\}. \tag{5.3}$$

Eq. 5.2 tells that all points in $\mathbf{V}^t(Q, s)$ can be connected with Q by a line that intersects s. Marking the intersection point as P changes Eq. 5.2 to

$$\mathbf{V}^t(Q, s) = \{\forall X \in \mathbf{W} | \exists P \in s : \overline{QX} \subset \mathbf{W} \wedge \overline{QX} \cap s = P\} = \{\forall X \in \mathbf{W} | \exists P \in s : \overline{QX} \subset \mathbf{W} \wedge P \in \overline{QX}\}. \tag{5.4}$$

Putting Eq. 5.3 and Eq. 5.4 together as is Eq. 5.1 gives

$$\{\forall X \in \mathbf{W} | \exists P \in s : \overline{QX} \subset \mathbf{W} \wedge P \in \overline{QX}\} \subset \{\forall Y \in \mathbf{W} | \exists Z \in s : \overline{ZY} \subset \mathbf{W}\}. \tag{5.5}$$

For a proof by a dispute, we can swap sides

$$\{\forall Y \in \mathbf{W} | \exists Z \in s : \overline{ZY} \subseteq \mathbf{W}\} \subset \{\forall X \in \mathbf{W} | \exists P \in s : \overline{QX} \subset \mathbf{W} \wedge P \in \overline{QX}\}. \tag{5.6}$$

Because $\forall Y \subset \forall X$, X has to fulfill conditions of Y and it can be substituted as $X = Y$:

$$\{\forall Y \in \mathbf{W} | \exists Z \in s : \overline{ZY} \subseteq \mathbf{W}\} \subset \{\forall Y \in \mathbf{W} | \exists P \in s : \overline{QY} \subset \mathbf{W} \wedge P \in \overline{QY}\}. \tag{5.7}$$

Because Q can be an arbitrary point, Eq. 5.7 can be disproven very easily with Fig. 5.2.

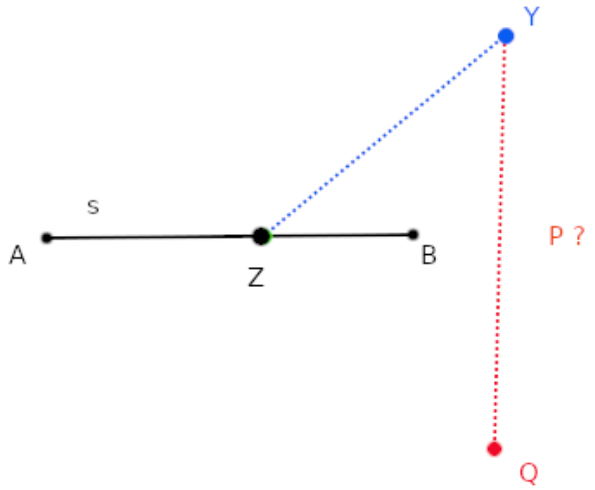


Figure 5.2: Example that point Y visible from segment s does not have to be visible from point Q over s. (By Eq. 2.4 $\overline{QY}$ must intersect s, which is not the case.)

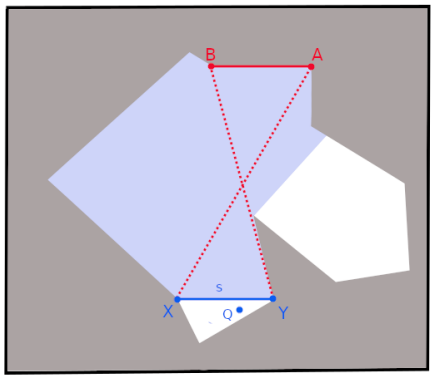
The reasoning above shows that the visibility region of the segment bounds all visibility regions from any point over the segment. This is the base stone of all versions of the proposed *EdgeVis* algorithm.

5.3 Basic structure for EdgeVis

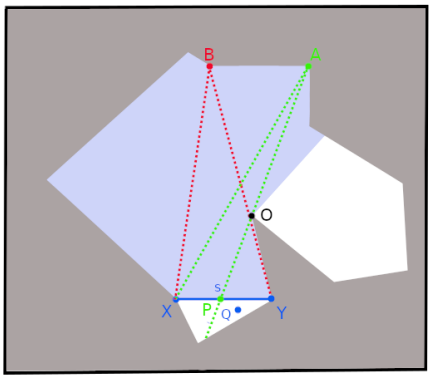
Let us assume visibility regions for edges were computed by numerically stable PEA-E. The output is a list of non-traversable segments whose endpoints are defined either as a vertex in mesh or as an abstract intersection. This representation is not very useful, so the algorithm splits the segments into endpoints and computes roots for them (as in PEA-E root is a subsegment on s fully visible from point $X \in \mathbf{V}(s)$). In Fig. 5.3a is one output segment of PEA-E for edge $\overline{XY}$. Preprocessing is common for all versions of EdgeVis, splits the segment to endpoints and computes their roots as shown on segment $\overline{AB}$ in Fig. 5.3. Endpoint X can be visible only if query Q is in its visibility region over its root (in Fig. 5.3b A is not visible from Q):

$$X \in \mathbf{V}(Q) : Q \in \mathbf{V}^t(X, r_x). \tag{5.8}$$

The set of all converted vertices with their roots is called $\mathbf{S}^V(T, e)$, where T is an entrance triangle and e is an edge. The algorithm loops through the vertices counter-clockwise.



(a) : PEA-E output end segment with root.



(b) : EdgeVis input single points with roots.

Figure 5.3: Conversion of the output of PEA-E to structure suitable for EdgeVis.

5.4 EdgeVis 1: Naive

The naive approach is to go through all vertices of $\mathbf{S}^V(T, e)$ and evaluates their visibility from Q. The computation can result in *visible*, *left*, and *right* states. The *left/right* state means that vertex X_i is *left/right* to $\mathbf{V}^t(Q, e)$. Because $\mathbf{V}^t(Q, e)$ is being computed, the state is computed from position to $\mathbf{V}^t(X_i, r_{X_i})$, which is the opposite. In Fig. 5.3b Q is left (CCW) to $\mathbf{V}^t(A, r_A)$, and so its state will be right when looking from Q.

The visibility of the vertices alone is not enough to define visibility region $\mathbf{V}^t(Q, e)$. When two adjacent vertices are in different states, the segment between them must be partially visible. Partial visibility triggers intersections of the segment and some limiting line defined with Q and past or future vertex. Multiple examples of such intersections are in Fig. 5.4. For example, point C' is defined by previous point B and D' is defined with future point E. The intersection is triggered only when leaving the *right* state or entering the *left* state. The last visible point P_{last} and the last visible segment s_{last} (at least partially) must be saved for evaluation of the intersections. The algorithm starts with the right vertex of e, which is always *visible* and then follows the diagram in Fig. 5.5.

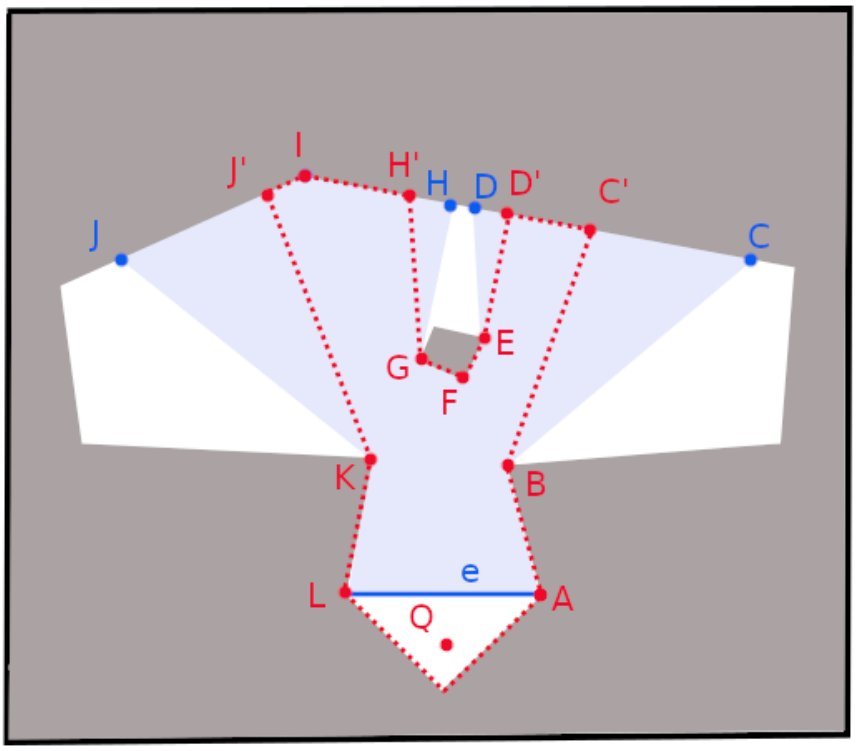


Figure 5.4: Example of $\mathbf{V}(Q)$ in $\mathbf{V}^t(Q, e)$.

In Fig. 5.4, $\mathbf{V}(e)$ has only eleven vertices, and each of them is used for computing at least one vertex from $\mathbf{V}^t(Q, e)$. In realistic scenarios from Iron Harvest [HHJ22], the average $\mathbf{V}(e)$ has between one and five hundred vertices and on average half of them have no relation to $\mathbf{V}^t(Q, e)$.

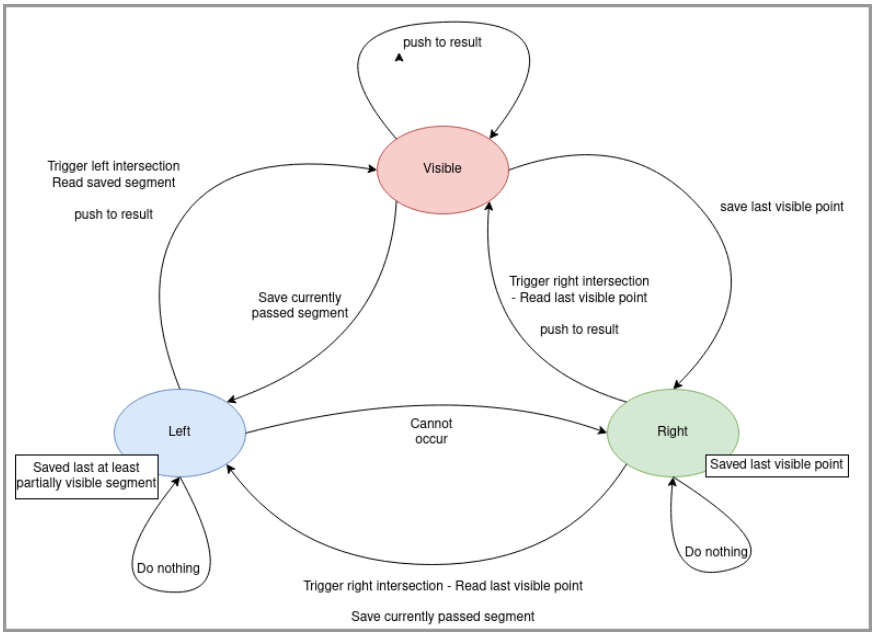


Figure 5.5: State machine that represents the naive version of EdgeVis.

Applying the diagram in Fig. 5.5 on the example in Fig. 5.4 will produce following actions on vertices:

A is *visible*: $\mathbf{V}^t(Q, e) = \{A\}$,

B is *visible*: $\mathbf{V}^t(Q, e) = \{A, B\}$,

C is *right*: $P_{last} = B$,

D is *left*: $\overrightarrow{QP_{last}} \cap \overline{CD} \Longrightarrow C'$; $\mathbf{V}^t(Q, e) = \{A, B, C'\}$; $s_{last} = \overline{CD}$,

E is *visible*: $\overrightarrow{QE} \cap \overline{s_{last}} \Longrightarrow D'$; $\mathbf{V}^t(Q, e) = \{A, B, C', D', E\}$,

F is *visible*: $\mathbf{V}^t(Q, e) = \{A, B, C', D', E, F\}$,

G is *visible*: $\mathbf{V}^t(Q, e) = \{A, B, C', D', E, F, G\}$,

H is *right*: $P_{last} = H$,

I is *visible*: $\overrightarrow{QP_{last}} \cap \overline{HI} \Longrightarrow H'$; $\mathbf{V}^t(Q, e) = \{A, B, C', D', E, H', I\}$,

J is *left*: $s_{last} = \overline{IJ}$,

K is *visible*: $\overrightarrow{QK} \cap \overline{s_{last}} \Longrightarrow J'$; $\mathbf{V}^t(Q, e) = \{A, B, C', D', E, H', I, J', K\}$,

L is *visible*: $\mathbf{V}^t(Q, e) = \{A, B, C', D', E, H', I, J', K, L\}$,

5.5 EdgeVis 2: Always Visible

The most expensive part of the vertex evaluation is the computation of two orientation predicates to obtain the visibility state of a vertex. The idea of this improvement is to find all vertices of $V(e)$ that are visible from any point in the entrance triangle $\mathbf{T}^e$. The goal is to find all vertices that follow

$$\forall X \in \mathbf{V}(e) : \forall Q \in \mathbf{T}^e, X \in \mathbf{V}^t(Q, e). \tag{5.9}$$

The algorithm checks if the whole entrance triangle is in the visibility region from X over its root:

$$\mathbf{T}^e \cap \mathbf{V}^t(X, r_x) = \mathbf{T}^e. \tag{5.10}$$

This is fulfilled only if $r_x = e$ and the vertex of $\mathbf{T}^e$ opposite to e (Y_e) is visible from X ($Y_e \in \mathbf{V}^t(X, r_x)$). Not many vertices fulfil this condition, so the computation of orientation predicates may be skipped only for 5-8% resulting in a 4-7% speedup compared to the naive approach. Fig. 5.6 shows that vertices B, F, G, K from the example in Fig. 5.4 are visible from any Q in the entrance triangle.

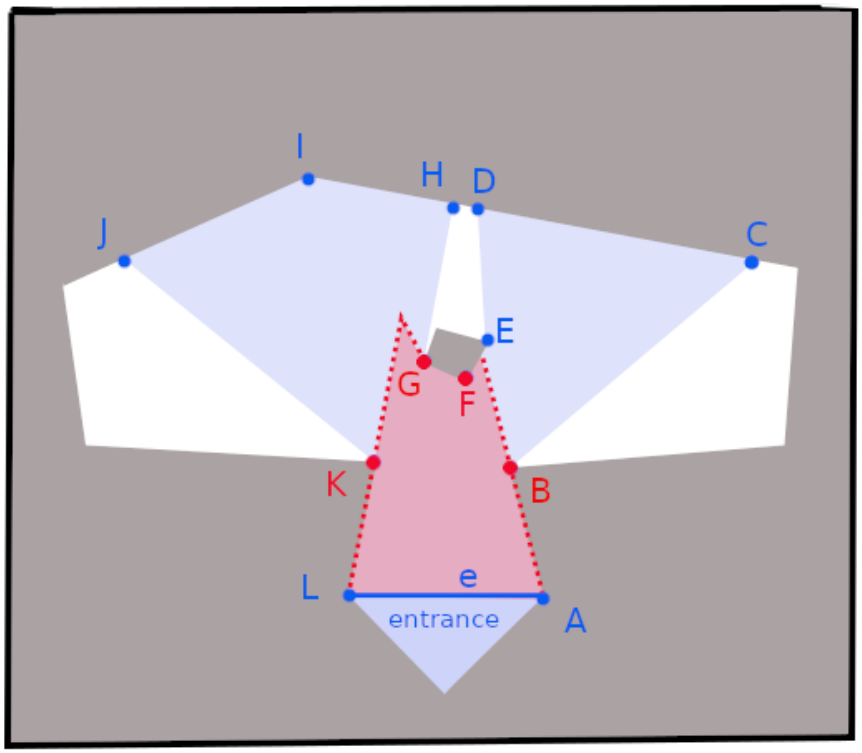


Figure 5.6: Example of the always visible area when looking from the entrance triangle over edge e.

5.6 EdgeVis 3: Online Pruning

The last discussed method uses the fact, that if the algorithm leaves *visible* state, the last visible vertex of $\mathbf{V}(e)$ defined a boundary of $\mathbf{V}^t(Q, e)$ (true for entering the *right* state, but a little more complicated for the *left* state). In Section 5.3 is shown that vertex $X \in \mathbf{V}(e)$ is visible from Q, only if Q is visible from X over its root. Also, the fundamental definition of visibility region for a point in Eq. 1.1 is connecting two visible points with a line. This applies that X is visible from Q only if

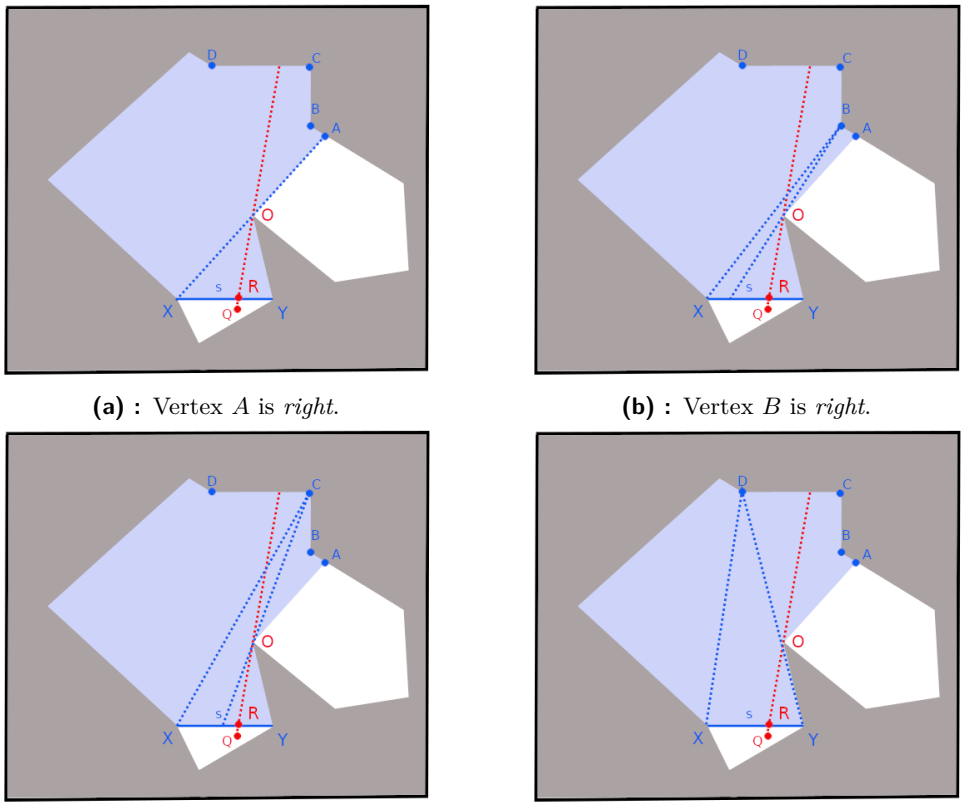
$$\overline{QX} \cap r_X \neq \varnothing. \tag{5.11}$$

When the *visible* state is left, the boundary created with Q and the last visible vertex intersects edge e creating a limiting root point R. Until a vertex whose root contains R is reached, all vertices can be skipped. (If the root of a vertex *'jumps'* to the other side of R, the state was changed from *right* to *left*, and intersections need to be triggered.)

Fig. 5.7 shows an example of pruning. In Fig. 5.7a A is evaluated as not visible, and bounding root point R is computed from the last visible vertex O. In Figs. 5.7b and 5.7c point R is not on roots r_B and r_C, so vertices B and C can be marked as not visible without computing orientation predicates. In last Fig. 5.7d $R \in r_D$ and vertex D is *visible*. The intersection between C and D is computed as in the naive method.

During preprocessing, all roots r_X are transformed to normalized interval $(0, 1)$, which describes on which part of the edge e the root is. For example, roots in Fig. 5.7 would be represented approximately as $r_A = (0, 0)$, $r_B = (0, 25)$, $r_C = (0, 5)$, and $r_D = (0, 1)$. When R is computed it is converted to a scalar position on the edge ($R = 0.66$ in Fig. 5.7). Instead of computing orientation predicates, vertices are checked by simple " $<$ ", " $>$ ".

The computation of R is quite expensive, and if it does not cut a lot of vertices, it is not very worth it. Future research can explore possible decisions for triggering the pruning procedure.



(a) : Vertex A is *right*.

(b) : Vertex B is *right*.

(c) : Vertex C is *right*.

(d) : Vertex D is *visible*.

Figure 5.7: About visibility of vertices can be decided by the location of boundary root point R.

5.7 Summary

This section introduced a new algorithm $EdgeVis$, for computing the visibility region of a point on a preprocessed structure of visibility regions from edges. Three algorithm variants were discussed, but all were slightly better than PEA on a polygonal mesh. This algorithm gives many options for future research due to complex relations in visibility regions for edges.

Chapter 6

Experiments

This chapter presents the results of the presented algorithms. We did not optimise every subroutine, but the optimisation level should be the same for all implementations (TEA and PEA should be affected by additional optimisation more than EdgeVis variants because EdgeVis has most of its subroutines in preprocessing). All tests were done on the same machine (Table 6.1) and in the same conditions (temperature and power supply).

OS	Ubuntu 22.04.1 LTS x86_64
Kernel	5.15.0-56-generic
CPU	11th Gen Intel i5-11400H (12) @ 4.500GHz
GPU	Intel TigerLake-H GT1 [UHD Graphics]
GPU	NVIDIA GeForce RTX 3050 Mobile
RAM	1x {DDR4, 3200 MT/s, 8 GB}

Table 6.1: The specification of testing laptop MSI GF63 Thin 11UC REV:1.0
.

Six experiments on the Iron Harvest dataset [HHJ22] are presented in this chapter. The dataset is introduced in Section 6.1 together with the first experiment of CDT and M-CDT mesh preparation. In Section 6.2 a comparison between TEA and PEA is presented. Section 6.3 shows the basic performance properties of the computation of edge visibility regions. The performance improvement of *EdgeVis* is shown in Section 6.4. The last sections 6.5 and 6.6 are dedicated to all algorithms' preprocessing times and the difference between robust and naive evaluation of orientation predicates.

6.1 Iron Harvest maps

The dataset is a set of complementary representations for 35 different level maps from the game *Iron Harvest*. J. Mikula, for his work on TriVis [MK22], created a new representation where every map is defined by its border and set of obstacles (n polygons of m vertices). The representation of J. Mikula is used together with CDT and merged CDT from *polyanya* paper [CHG17] to generate triangular or polygonal meshes. All maps in the dataset are complex enough to provide a sufficiently general environment. (example map *sp_pol_04* is shown in Fig. 6.1).

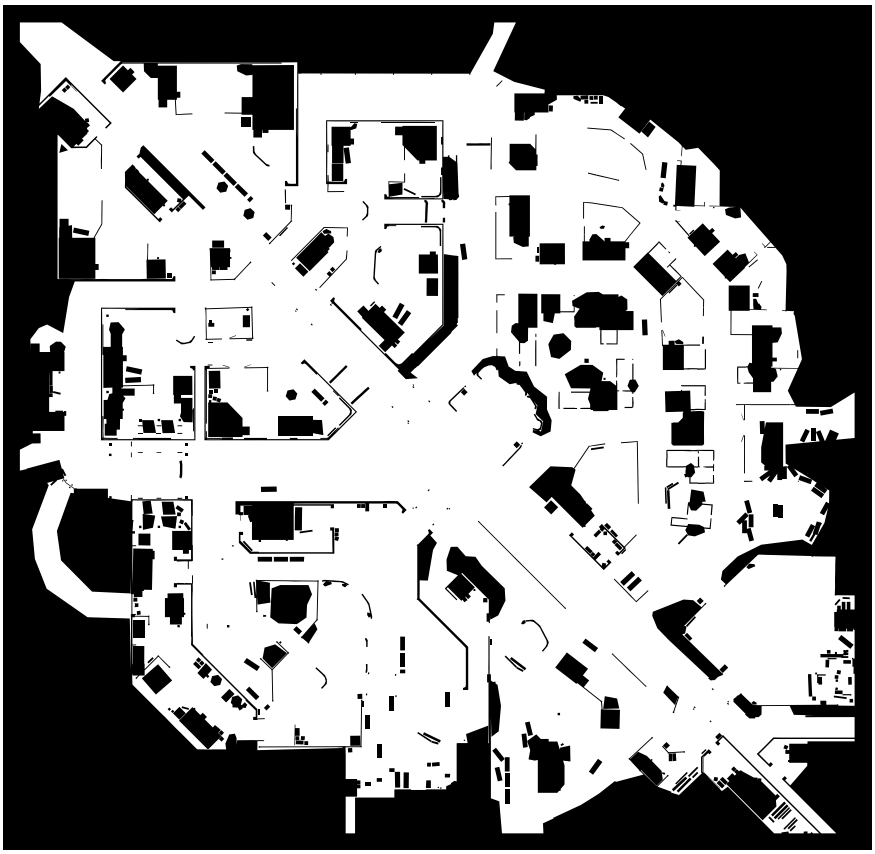


Figure 6.1: Iron harvest sp_pol_04 map.

The basic properties of all maps and meshes are listed in Table 6.2. The complexity of the maps is expressed by the number of obstacles and vertices needed to describe the environment. Polygonal meshes have less than half the elements as triangular ones, indicating possible performance improvement.

map	obstacles	vertices	triangles	polygons
mp_2p_01	263	3184	3673	1716
mp_2p_02	165	2301	2612	1143
mp_2p_03	199	3608	3985	1789
mp_2p_04	63	1932	2051	959
mp_4p_01	407	4110	4859	2136
mp_4p_02	376	6159	6888	2907
mp_4p_03	632	7396	8559	3952
mp_6p_01	376	4999	5680	2519
mp_6p_02	297	5359	5944	2422
mp_6p_03	287	4692	5254	2226
sp_cha_01	174	2413	2747	1184
sp_cha_02	254	3428	3926	1616
sp_cha_03	454	5320	6176	2775
sp_cha_04	623	6570	7768	3374
sp_endmaps	675	7987	9254	4070
sp_pol_01	63	1052	1173	552
sp_pol_02	336	5283	5917	2624
sp_pol_03	558	6775	7836	3308
sp_pol_04	479	6165	7054	3217
sp_pol_05	339	3934	4578	2031
sp_pol_06	679	8236	9508	4293
sp_rus_01	276	3098	3621	1644
sp_rus_02	167	2695	2998	1399
sp_rus_03	372	4855	5574	2499
sp_rus_04	360	5595	6280	2665
sp_rus_05	367	5966	6646	2974
sp_rus_06	598	7875	8986	4028
sp_rus_07	206	3568	3947	1774
sp_sax_01	163	2396	2703	1186
sp_sax_02	368	7002	7700	3478
sp_sax_03	368	6671	7368	3265
sp_sax_04	431	6862	7659	3470
sp_sax_05	65	1948	2070	907
sp_sax_06	269	4358	4849	2169
sp_sax_07	348	4132	4807	2088

Table 6.2: Properties of maps in Iron Harvest dataset.

6.2 TEA vs. PEA

To compare the performance of TEA and PEA, visibility regions were computed for 10^6 queries for both algorithms. Table B.1 shows that PEA is approximately 21.32% faster on average with the lowest improvement of 17.0% on map *sp_rus_03* and the most significant improvement of 30.6% on map *sp_sax_05*. Fig. 6.2 shows that *sp_sax_05* is more open and has significantly fewer obstacles. Large open spaces can be represented with only a few polygons, even if they are defined with many vertices. This causes PEA to be more efficient in such areas.

(a) : sp_rus_03 **(b) :** sp_sax_05

Figure 6.2: Maps with worst 6.2a and best 6.2bimprovement of PEA to TEA.

Table B.1 also shows the difference between average expansions and average maximal recursion depth for both algorithms. PEA reduces the number of expansions by more than 50%, but a more complex evaluation of expansion for PEA results in the final improvement of 21.32%. The reduction of expansions also provides an upper bound for PEA improvement. If we could optimise the PEA expansion procedure to the same speed as TEA expansion, the difference in performance would be the same as the difference in the number of expansions. However, the creation of polygonal mesh can be optimised as J. Mikula did for triangulation for TEA [MK22].

map	runtime[s]			avg. exp.[-]		avg. depth[-]	
	TEA	PEA	↑ [%]	TEA	PEA	TEA	PEA
mp_2p_01	15.95	12.49	21.7	228	100	47	18
mp_2p_02	14.68	11.15	24.1	213	90	52	20
mp_2p_03	20.80	15.86	23.7	294	125	61	23
mp_2p_04	6.05	4.35	28.0	83	37	32	13
mp_4p_01	16.68	13.80	17.3	232	102	51	21
mp_4p_02	23.13	19.15	17.2	315	132	67	27
mp_4p_03	14.21	11.27	20.6	177	79	46	18
mp_6p_01	18.31	14.64	20.0	251	108	60	25
mp_6p_02	18.61	14.41	22.6	247	101	63	24
mp_6p_03	13.44	10.44	22.3	184	77	53	20
sp_cha_01	23.53	17.63	25.1	346	142	63	25
sp_cha_02	28.23	21.16	25.1	431	187	68	27
sp_cha_03	14.06	11.23	20.1	186	81	52	20
sp_cha_04	14.27	11.62	18.5	188	83	41	17
sp_endmaps	15.51	12.34	20.4	201	93	46	18
sp_pol_01	6.61	4.86	26.5	96	43	27	11
sp_pol_02	9.59	7.64	20.3	126	53	40	15
sp_pol_03	18.83	15.13	19.6	248	103	57	21
sp_pol_04	21.52	17.15	20.3	291	125	61	23
sp_pol_05	11.77	9.68	17.8	164	72	43	17
sp_pol_06	14.95	11.86	20.6	174	77	50	20
sp_rus_01	10.20	8.13	20.3	144	64	40	15
sp_rus_02	6.22	4.86	21.9	87	42	26	11
sp_rus_03	6.51	5.40	17.0	85	38	28	12
sp_rus_04	13.30	10.76	19.1	180	76	46	18
sp_rus_05	14.89	11.71	21.3	194	83	48	19
sp_rus_06	12.95	10.57	18.4	160	71	46	19
sp_rus_07	12.09	9.51	21.3	167	73	45	18
sp_sax_01	10.00	8.01	19.9	144	64	38	16
sp_sax_02	19.24	15.54	19.2	248	107	56	22
sp_sax_03	10.93	8.88	18.7	142	62	36	15
sp_sax_04	10.24	7.88	23.1	129	55	36	14
sp_sax_05	5.75	3.99	30.6	79	32	29	10
sp_sax_06	14.27	11.44	19.9	194	85	53	21
sp_sax_07	10.72	8.20	23.5	147	63	40	15
Average	14.23	11.22	21.32	194	84	47	18

Table 6.3: Comparison of TEA and PEA performance.

6.3 Edge visibility regions

The computation of the edge visibility region introduced in this paper is the first solution to the given problem (to the best of our knowledge). Fig. 6.3 shows the visibility region for a segment defined with two vertices in a polygonal mesh. Table 6.4 contains measurements of visibility region computation

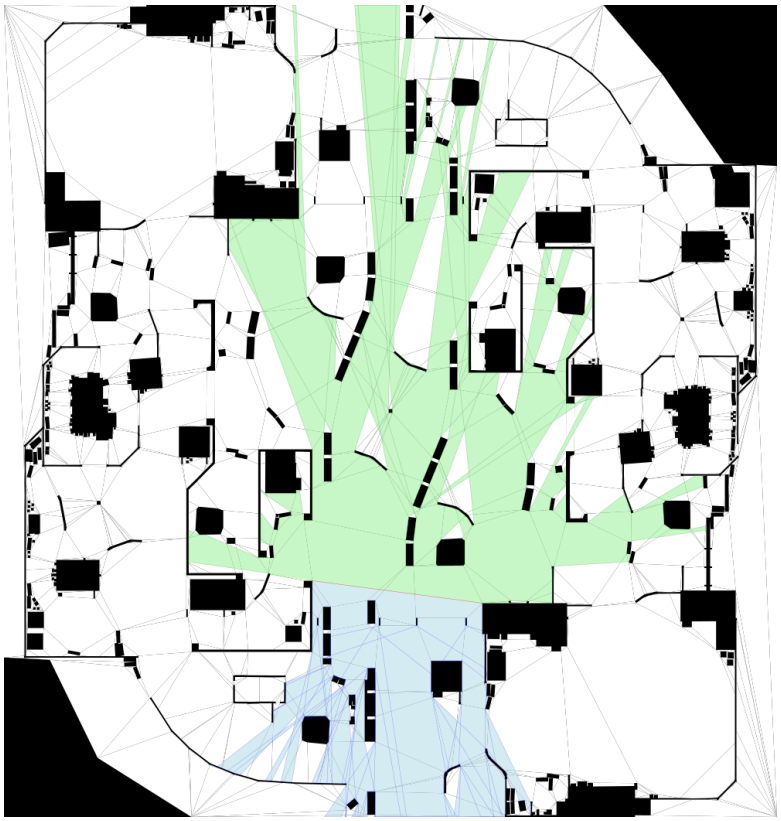


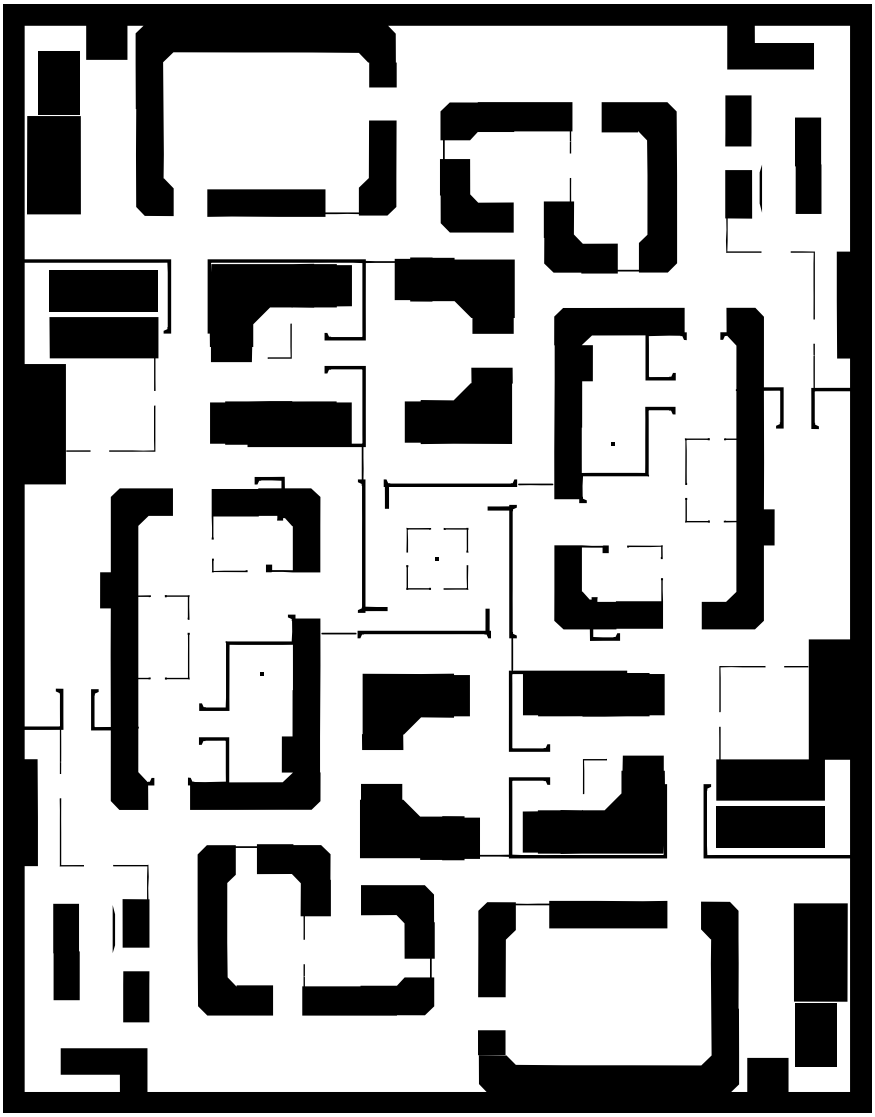
Figure 6.3: Edge visibility region in mp_2p_01 map.

for all traversable edges in triangular mesh for all maps. This setup is used as preprocessing for all EdgeVis variants, so it is most relevant for detailed evaluation. The average computation time for an edge is approximately one order of magnitude longer than for a point in the same environment (comparison of Table 6.4 and Table B.1). The table further shows the number of expansions and depth of recursion for **one side**. The higher the number of expansions, the more connected the map is. Fig. 6.4a shows the map *mp_2p_04* with low expansions. Due to its complexity, only a close neighbourhood is usually visible. The opposite case is map *sp_cha_02* in Fig. 6.4b, where lack of significant obstacles often causes edges to see large parts of the map. The recursion depth describes the presence of vision corridors. Maps in Fig. 6.4 have similar recursion depth, which means, that even though *mp_2p_04* is more complex, it has a lot of visible corridors allowing long vision in at least

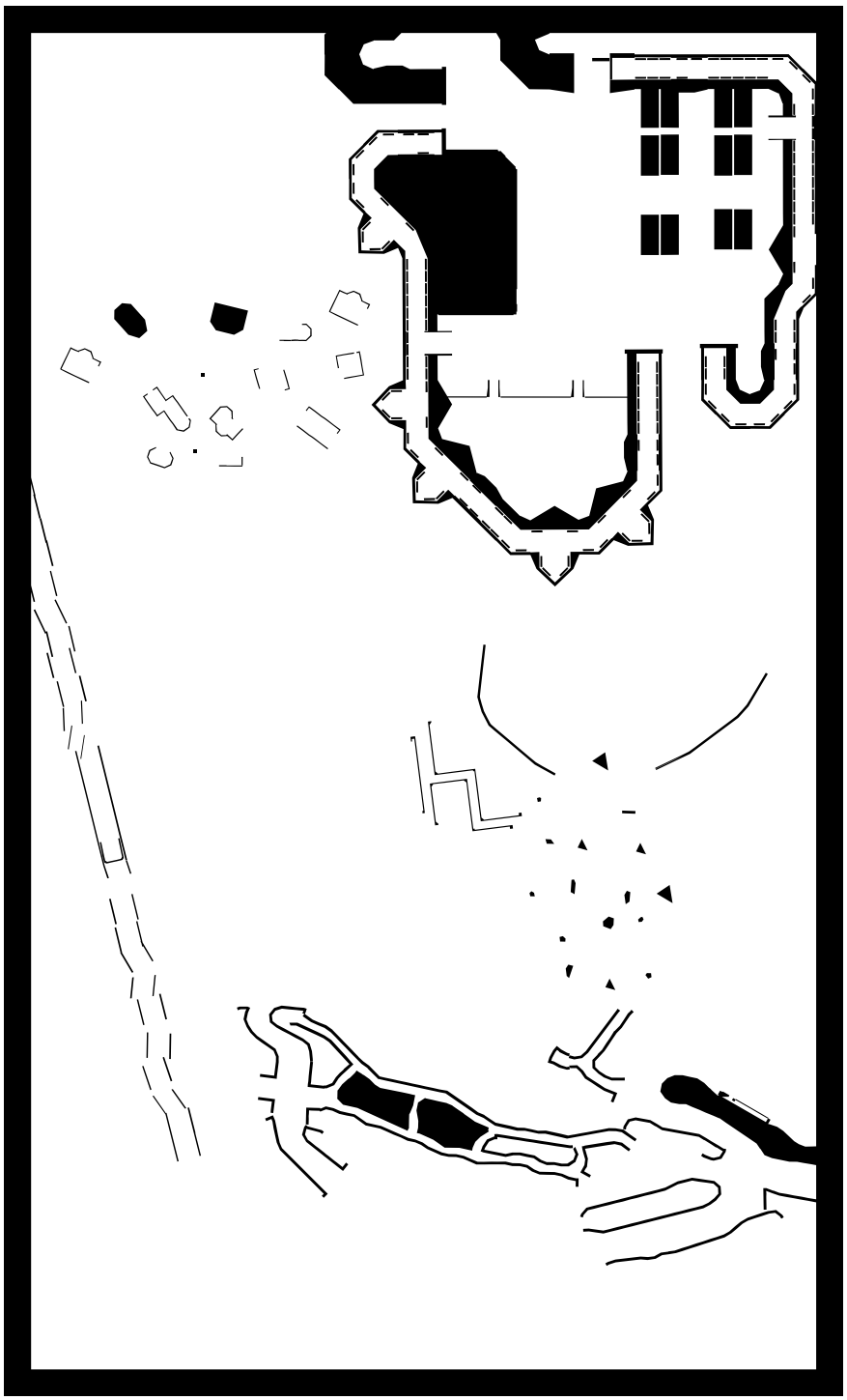
one direction. For example, map *sp_rus_03* is computed quite fast, even though it has many edges. Given its low recursion depth and low expansion count, it is probably a complex map, as shown in the last section in Fig. 6.2a.

map	count	runtime[s]	depth		expansions	
			mean	max	mean	max
mp_2p_01	3900	0.619	34	92	121	946
mp_2p_02	2759	0.490	39	105	135	843
mp_2p_03	4164	0.948	46	144	169	902
mp_2p_04	2108	0.174	27	111	59	271
mp_4p_01	5202	0.979	40	143	145	961
mp_4p_02	7242	1.969	55	155	201	1263
mp_4p_03	9091	1.111	30	125	92	1257
mp_6p_01	5986	1.095	42	147	138	1361
mp_6p_02	6233	1.313	49	154	156	920
mp_6p_03	5530	0.858	40	209	110	1001
sp_cha_01	2908	0.863	44	122	225	1570
sp_cha_02	4171	0.918	40	126	164	4590
sp_cha_03	6579	0.982	36	158	111	941
sp_cha_04	8344	1.166	32	107	108	1720
sp_endmaps	9847	1.313	35	155	97	1107
sp_pol_01	1232	0.094	21	54	59	274
sp_pol_02	6216	0.745	33	124	89	496
sp_pol_03	8340	1.754	44	173	160	1503
sp_pol_04	7465	1.516	37	179	151	2067
sp_pol_05	4884	0.712	35	129	113	1200
sp_pol_06	10102	1.727	43	193	120	885
sp_rus_01	3869	0.433	30	105	84	621
sp_rus_02	3135	0.184	17	55	42	368
sp_rus_03	5922	0.539	26	104	71	663
sp_rus_04	6606	0.912	35	127	103	805
sp_rus_05	6960	0.983	33	129	104	1250
sp_rus_06	9500	1.149	30	114	90	1007
sp_rus_07	4121	0.558	34	111	99	568
sp_sax_01	2848	0.411	33	105	111	1252
sp_sax_02	8031	1.452	39	135	131	1328
sp_sax_03	7698	0.968	31	131	94	714
sp_sax_04	8026	0.875	29	105	82	646
sp_sax_05	2128	0.156	25	93	54	273
sp_sax_06	5072	0.823	41	137	115	709
sp_sax_07	5135	0.620	31	112	90	651

Table 6.4: Computation of visibility regions of traversable edges in a triangular mesh.



(a) : mp_2p_04 : few expansions



(b) : sp_cha_02 : many expansions

Figure 6.4: Maps with few 6.4a and many 6.4b expansions in PEA-E.

6.4 EdgeVis performance

Table 6.5 presents improvement in the performance for all *EdgeVis* variants relative to PEA. The improvement is computed as

$$E_X \uparrow [\%] = 100 * (1 - \frac{t_{E_X}}{t_{PEA}}), \tag{6.1}$$

where E_X is one of the three variants of the *EdgeVis*, and $t_{algorithm}$ is total runtime of an algorithm for 10^6 queries.

The naive version is 33.1% faster than PEA, and the two improved methods reduce the query runtime by only a few percentage points, indicating that the resources put into the implementations are not worth it. However, the improvement by the second variant, which always precomputes visible vertices, is significant on some specific maps as *sp_sax_05*. The second variant is potent on maps with large areas with no obstacles because many vertices can be observed from large areas. The improvement of the third variant with online pruning is more consistent than the second one.

Columns under *evaluated v.* show the number of vertices from a precomputed edge visibility region for which *EdgeVis* must compute orientation. The decrease of evaluated vertices between the first and second versions is slight because the condition of sure visibility is demanding. On the other hand, the reduction presented by the third variant of *EdgeVis* is between $30 - 50\%$, which indicates effective pruning. The reason for the small improvement in query performance for *Edgevis 3* is the high cost of computing the boundary line for pruning. Therefore, the first improvement that comes to mind is to start pruning only if it probably saves more time than it spends on computing the boundary.

Our implementation of the *online pruning EdgeVis* contains a minor bug that we could not fix yet. It sometimes omits vertex in the final polygon if it is defined as an intersection and creates a border between *visible* and *right* state. Column **E3** ✓ in Tab. 6.5 shows the success rate for the third variant, where the query was computed successfully if an area is different by less than $10^{-6}\%$ from the other algorithms (the difference between areas from the other algorithms were less than $10^{-12}\%$ which can be attributed to rounding errors). The measurement is not affected by this bug.

map	↑ [%]			evaluated v.			E3 ✓[%]
	E1	E2	E3	E1	E2	E3	
mp_2p_01	33.3	35.7	37.1	291	272	176	100.00
mp_2p_02	33.1	36.2	37.7	274	255	161	100.00
mp_2p_03	38.2	40.0	41.5	356	334	211	100.00
mp_2p_04	21.1	27.9	21.2	106	92	75	99.72
mp_4p_01	38.0	40.3	41.4	298	277	176	99.65
mp_4p_02	44.2	45.9	46.9	383	361	219	100.00
mp_4p_03	34.6	37.5	36.3	219	202	141	99.90
mp_6p_01	40.0	42.1	42.7	303	284	175	100.00
mp_6p_02	42.5	44.5	44.4	281	262	169	99.96
mp_6p_03	35.4	38.5	37.2	220	202	139	99.97
sp_cha_01	38.4	39.8	43.3	423	403	235	100.00
sp_cha_02	30.6	31.4	42.5	707	680	347	99.99
sp_cha_03	36.9	39.8	39.1	225	208	141	100.00
sp_cha_04	32.3	35.1	34.4	235	219	146	100.00
sp_endmaps	28.0	30.5	34.3	336	317	186	100.00
sp_pol_01	23.0	29.5	23.9	117	103	82	100.00
sp_pol_02	29.6	34.3	30.3	151	134	101	100.00
sp_pol_03	37.4	39.4	40.8	313	294	179	100.00
sp_pol_04	38.9	40.7	42.5	367	347	208	99.94
sp_pol_05	33.2	36.6	35.8	203	186	125	99.65
sp_pol_06	38.3	41.4	40.0	217	200	136	100.00
sp_rus_01	28.5	32.3	29.7	174	158	118	100.00
sp_rus_02	21.3	29.5	23.1	111	95	84	99.97
sp_rus_03	22.7	28.7	23.3	106	93	73	100.00
sp_rus_04	33.9	36.7	35.6	221	203	137	99.98
sp_rus_05	36.4	39.1	38.4	234	215	147	100.00
sp_rus_06	36.7	39.9	37.7	191	174	126	100.00
sp_rus_07	33.8	37.1	34.7	199	181	131	100.00
sp_sax_01	31.4	35.5	34.5	187	171	118	100.00
sp_sax_02	41.9	44.1	43.5	298	276	181	100.00
sp_sax_03	33.4	37.4	33.9	171	154	114	100.00
sp_sax_04	28.5	33.2	29.9	158	142	104	100.00
sp_sax_05	18.3	26.0	18.2	93	79	68	100.00
sp_sax_06	36.6	39.3	37.9	231	211	150	100.00
sp_sax_07	28.6	32.6	30.7	181	163	118	100.00
Average	33.1	36.5	35.6	245	227	149	99.96

Table 6.5: Comparison of EdgeVis variants with PEA.

6.5 Preprocessing

Table 6.6 presents the preprocessing times for all discussed algorithms. The *CDT* and *M-CDT* columns contain the loading of a map and the CDT or M-CDT preprocessing. The M-CDT takes longer but compared to other necessary preprocessing, it is negligible. The improvement in query performance strongly outweighs the preprocessing cost, so PEA should always be prioritised over TEA.

The computation of visibility regions for all the edges in the mesh is costly, resulting in 30× longer preprocessing for *EdgeVis*. Column *E1* shows the additional time added to preprocessing of edge visibility regions for the naive version of *EdgeVis*. The additional variants of *EdgeVis* need further preprocessing built on the naive version, which is shown in columns *E2, E3*. Preprocessing of *EdgeVis* takes from a few hundred milliseconds up to a few seconds, which makes it suitable for applications that can afford longer preprocessing.

map	CDT[ms]	M-CDT[ms]	edges[ms]	E1[ms]	E2[ms]	E3[ms]
mp_2p_01	22	24	619	171	33	51
mp_2p_02	15	17	490	136	25	39
mp_2p_03	24	27	948	266	45	71
mp_2p_04	12	14	174	52	12	17
mp_4p_01	31	34	979	266	48	75
mp_4p_02	46	50	1969	547	88	139
mp_4p_03	64	70	1111	311	61	94
mp_6p_01	36	41	1095	303	54	83
mp_6p_02	38	42	1313	367	60	95
mp_6p_03	34	38	858	245	44	70
sp_cha_01	16	17	863	237	36	59
sp_cha_02	22	25	918	254	41	65
sp_cha_03	40	44	982	277	51	79
sp_cha_04	54	60	1166	321	63	98
sp_endmaps	69	75	1313	377	72	111
sp_pol_01	6	7	94	26	6	10
sp_pol_02	38	42	745	211	42	64
sp_pol_03	53	59	1754	482	84	132
sp_pol_04	50	55	1516	412	69	112
sp_pol_05	28	32	712	195	37	58
sp_pol_06	71	77	1727	502	86	133
sp_rus_01	22	24	433	122	25	39
sp_rus_02	18	20	184	56	14	21
sp_rus_03	36	40	539	149	32	50
sp_rus_04	41	45	912	256	49	76
sp_rus_05	44	49	983	286	53	81
sp_rus_06	65	72	1149	322	64	99
sp_rus_07	24	27	558	159	30	47
sp_sax_01	16	17	411	115	22	34
sp_sax_02	51	57	1452	415	72	119
sp_sax_03	50	55	968	275	54	84
sp_sax_04	52	59	875	247	51	78
sp_sax_05	12	13	156	46	11	16
sp_sax_06	31	34	823	237	43	66
sp_sax_07	30	33	620	176	35	54

Table 6.6: Table of preprocessing times for all algorithms.

6.6 Robust vs naive orientation

All the experiments above were performed with the use of Shewchuk's robust orientation test [She97]. Table 6.7 presents an increase in performance if the naive variant susceptible to rounding errors is used. All the algorithms have approximately 20% faster query response if the orientation is computed with limited precision. The resulting areas for robust and naive orientation tests are compared for 10^4 queries, and the difference was always less than 10^{-12}%. This indicates that using orientation tests with limited precision does not create errors in common scenarios. If an application can afford the risk of sporadic errors, orientation should be tested with limited precision to achieve faster query response.

map	TEA[%]	PEA[%]	E1[%]	E2[%]	E3[%]
mp_2p_01	23.45	24.69	21.73	18.79	18.82
mp_2p_02	23.93	25.84	23.45	20.04	22.00
mp_2p_03	22.50	24.09	20.19	17.81	18.52
mp_2p_04	23.48	24.82	24.83	19.71	23.91
mp_4p_01	24.02	23.64	20.74	18.47	18.76
mp_4p_02	23.20	22.79	19.98	17.72	17.94
mp_4p_03	21.95	22.44	20.82	17.96	18.41
mp_6p_01	22.72	22.54	19.87	17.58	18.88
mp_6p_02	22.20	23.06	20.40	17.90	18.40
mp_6p_03	21.64	22.97	21.57	18.15	19.83
sp_cha_01	23.68	26.67	20.35	17.96	18.42
sp_cha_02	22.74	26.61	19.87	17.53	19.25
sp_cha_03	22.67	22.37	20.47	17.06	17.86
sp_cha_04	21.67	23.07	20.68	17.90	18.65
sp_endmaps	22.78	22.38	22.50	19.60	20.98
sp_pol_01	23.09	25.79	27.89	22.95	25.99
sp_pol_02	23.45	22.66	22.63	18.58	20.40
sp_pol_03	22.19	21.99	20.52	17.93	17.72
sp_pol_04	22.33	22.75	19.30	17.28	17.05
sp_pol_05	24.06	24.15	21.39	18.05	19.10
sp_pol_06	23.39	19.37	19.86	16.39	17.41
sp_rus_01	22.77	24.63	22.78	19.33	20.49
sp_rus_02	23.74	24.36	26.49	20.23	22.74
sp_rus_03	22.95	22.43	23.20	18.72	21.41
sp_rus_04	21.79	22.98	20.80	18.18	19.27
sp_rus_05	21.84	20.65	21.05	16.49	18.68
sp_rus_06	20.81	21.10	20.59	17.35	18.68
sp_rus_07	23.80	22.22	21.09	17.80	19.14
sp_sax_01	22.03	25.77	23.04	19.11	21.88
sp_sax_02	22.42	21.52	20.03	17.27	18.03
sp_sax_03	21.99	21.80	21.01	16.95	19.41
sp_sax_04	21.68	22.27	22.12	17.65	19.58
sp_sax_05	23.07	22.90	25.79	20.56	23.85
sp_sax_06	21.29	21.98	20.75	17.85	18.21
sp_sax_07	23.19	23.22	21.61	17.90	19.25
Average	22.70	23.22	21.70	18.31	19.68

Table 6.7: Improvement in query the performance without usage of robust orientation predicates.

Chapter 7

Conclusion

This paper focuses on the computation of visibility regions for query points in 2D polygonal environments. We describe the state-of-the-art *triangular expansion algorithm* and use it as the benchmark for new algorithms.

First, we expanded TEA from triangular meshes to polygonal ones improving the query performance by 21%. The *polygonal expansion algorithm* presented in the *Polyanya* project [CHG17] expands visibility region to neighbouring convex polygons until it hits an obstacle. On meshes created with CDT and M-CDT used in the original project, PEA needs approximately 4× fewer expansion steps than TEA. Further optimization of the PEA expansion routine can increase the performance more than our 20% improvement. However, it is always bounded by the expansion reduction ratio (4 × fewer expansions $\sim$ 75% upper bound for optimization). The first improvement of PEA is to apply binary search in the expansion procedure as in the *Polyanya* project instead of the current linear search implemented by us. Another promising path is to find optimal polygonal mesh as J. Mikula did for TEA [MK22].

Next, we provided the first solution for the computation of visibility regions from segments. Until now, it had to be computed by sampling the segment with points which gave an approximate result. Our *polygonal expansion algorithm for edges* (PEA-E) gives a precise result, and its query performance is slower than TEA and PEA only by one order of magnitude. The algorithm works on a polygonal (or triangular) mesh and expands the visibility region to neighbouring polygons until an obstacle stops it. Unlike PEA, PEA-E must look back after every expansion to verify from which part of the original

segment the expanded visibility region can be observed. PEA-E can be improved with binary search in forward expansion similarly to PEA. Also, the current backward visibility goes through all previously passed polygons, which could be most likely also optimized.

At last, we presented a new algorithm *EdgeVis* for the computation of visibility regions for points which uses a precomputed structure of visibility regions from edges in a mesh. In the first naive proposed form, the algorithm looks from a point over the closest edges in the mesh and evaluates the visibility of every vertex in the edge visibility region. It is very similar to early first sweep algorithms, except the environment is reduced to what is visible from an edge. The first improved version checks whether the vertex of the edge visibility region is always visible from the entrance polygon. Vertices marked as such do not have to be evaluated because they must be visible. The second improvement proposes online pruning, which can skip sections of not-visible vertices. The naive version enhanced the query performance of PEA by 33%, and the improved versions of *EdgeVis* increased the speed of the naive version by 7-10%. The algorithm builds on new visibility from edges and opens the possibility of research for its optimization. The first clear option is to combine the two proposed improvements. Also, the variant with online pruning could be expanded with some decisions when the pruning is worth it. Since the computation of the online boundary is costly, it is not worth computing for cutting off just a few vertices.

Our best algorithm evaluates query twice as fast as the state-of-the-art algorithm TEA. We also opened a new section of research for computation of visibility regions for segments. The results of our work are not only valuable now, but they also create new challenges for the future research.

Appendix A

Bibliography

[Asa85] T. Asano, *An efficient algorithm for finding the visibility polygon for a polygonal region with holes*, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **68** (1985), 557–559.

[BHH+14] Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller, *Efficient computation of visibility polygons*, arXiv preprint arXiv:1403.3905 (2014).

[CHG17] Michael Cui, Daniel Damir Harabor, and Alban Grastien, *Compromise-free pathfinding on a navigation mesh*, Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, 2017, pp. 496–502.

[DB79] L. S. Davis and M. L. Benedikt, *Computational models of space: Isovists and isovist fields*, Computer Graphics and Image Processing 11(1), 49–72 (1979) (1979).

[EA81] H. ElGindy and D. Avis, *A linear algorithm for computing the visibility polygon from a point*, Journal of Algorithms (1981).

[Gho07] S. K. Ghosh, *Visibility algorithms in the plane*, Cambridge University Press (2007).

[HHJ22] D. D. Harabor, R. Hechenberger, and T. Jahn, *Benchmarks for pathfinding search: Iron harvest*, Proceedings of the Fifteenth International Symposium on Combinatorial Search (SoCS 2022) (2022).

[HM95] P. J. Heffernan and J. S. B. Mitchell, *An optimal algorithm for computing visibility in the plane*, SIAM Journal on Computing **24** (1995).

[IK09] Rajasekhar Inkulu and Sanjiv Kapoor, *Visibility queries in a polygonal region*, Computational Geometry **42** (2009), no. 9, 852–864.

[JS87] B. Joe and R. B. Simpson, *Corrections to lee's visibility polygon algorithm*, BIT 27(4) (1987).

[Lee83] D. T. Lee, *Visibility of a simple polygon*, Computer Vision, Graphics, and Image Processing (1983).

[Lee19] C. Lee, *Fade2d library for constrained delaunay triangulation*, `https://github.com/Lee0326/Fade2D`, March 2019.

[MK22] Jan Mikula and Miroslav Kulich, *Triangular Expansion Revisited: Which Triangulation Is The Best?*, Proceedings of the 19th International Conference on Informatics in Control, Automation and Robotics - ICINCO, INSTICC, SCITEPRESS - Science and Technology Publications, 2022, pp. 313–319.

[She97] Jonathan Richard Shewchuk, *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates*, Discrete & Computational Geometry **18** (1997), no. 3, 305–363.

Appendix B

Attached files

EdgeVis (attachment.zip)	
paper	compressed .zip file with latex source of the thesis
dependencies	maps from Iron Harvest dataset
source	all the code - structure described in README.md
scripts	python script for automatic experiments
README.md	file with a short description of the project
CMakeLists.txt	root file for CMake project

Table B.1: Table of attached folders.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Rosol Jakub** Personal ID number: **465842**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Cybernetics and Robotics**

Branch of study: **Cybernetics and Robotics**

II. Master’s thesis details

Master’s thesis title in English:

Fast Computation of Visibility Polygons

Master’s thesis title in Czech:

Rychlý výpočet polygonu viditelnosti

Guidelines:

1. Get acquainted with the Polyanya library (https://bitbucket.org/mlcui1/polyanya/src/master/) for an efficient path finding.
2. Modify the library for computation of a visibility polygon from a point on a polygonal mesh.
3. Design and realize an algorithm for computation of a visibility polygon from an edge.
4. Design and realize an algorithm which finds a visibility polygon from a point based on a visibility polygon from an edge.
5. Evaluate experimentally properties of the extended algorithm. Describe and discuss the obtained results.

Bibliography / sources:

[1] Shen, B.; Cheema, M. A.; Harabor, D.; and Stuckey, P. J. 2020a. Euclidean Pathfinding with Compressed Path Databases. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, 4229–4235
[2] Cui, M.; Harabor, D. D.; and Grastien, A. 2017. Compromise-free Pathfinding on a Navigation Mesh. In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, 496–502.
[3] Asano, T. (1985). An efficient algorithm for finding the visibility polygon for a polygonal region with holes. IEICE TRANSACTIONS (1976-1990), 68(9), 557-559.
[4] Bungiu, F., Hemmer, M., Hershberger, J., Huang, K., & Kröller, A. (2014). Efficient computation of visibility polygons. arXi preprint arXiv:1403.3905.

Name and workplace of master’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **13.09.2022** Deadline for master's thesis submission: **10.01.2023**

Assignment valid until: **19.02.2024**

RNDr. Miroslav Kulich, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature