Utrecht University
Faculty of Science
Graduate School of Natural Sciences
Supervisor: Frank Staals

Master Thesis

Visibility graphs of portalgons

—

Rens de Heer
4214080

July 2021

1 Introduction

We study a generalization of polygons, called *portalgons*. This structure was introduced by Löffler et al[8]. A portalgon is a set of simple polygons where pairs of same-length edges can be glued together by *portals*. This results in a 2-dimensional orientable manifold induced with the Euclidean metric.

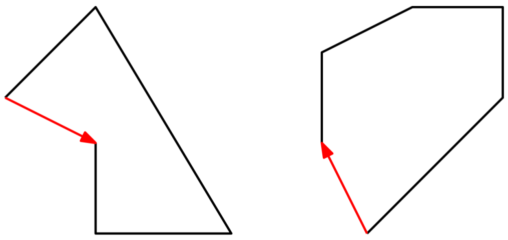
In this thesis we analyse the structure of portalgons using *visibility graphs*. These are graphs on the vertices of a polygon, where two vertices are connected by an edge if there exists a straight line segment between them that lies completely within the polygon. These graphs can be used to compute more complicated queries like shortest path maps[6] and minimum-link paths[9].

There exists multiple algorithm for computing the visibility graph of a polygon. In the literature study multiple of these algorithms are inspected for their adaptability in portalgons. In this thesis we adapt and implement the algorithm by Lee[7]. First we describe the complete algorithm including degenerate cases. After this we describe the implementation, the acquisition of a set of portalgons and finally we analyse the visibility graphs of these portalgons computed by the algorithm.

1.1 Portalgons

Definition. A *portalgon* is a collection of simple polygons, called *fragments*. A pair of edges that have the same length can form a *portal*. These edges are topologically glued together in such a way that the resulting surface remains orientable. Edges of portal pair are called *portaledges* and do not have to be part of the same fragment.

An example of a portalgon can be seen in figure 1a. A portal is sometimes drawn with an arrowhead to indicate how the edges must be glued together. All fragments in a portalgon are simple, as each hole can be removed by splitting the polygon in two pieces by two cuts from the outer boundary to the hole and adding a portal at all the dividing segments, see figure 1b.


(a) A portalgon consisting of 2 fragments

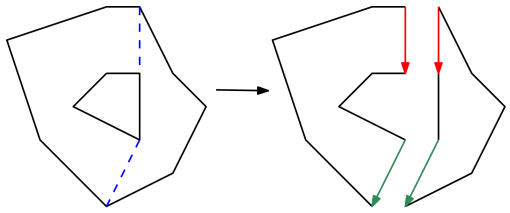

(b) A non-simple fragment can always be transformed into multiple simple fragments by adding extra portals

Figure 1

1.2 Visibility graph

Definition. A *visibility graph* of a polygonal surface is a graph containing all the vertices of the surfaces boundary as nodes with edges between nodes if there exists a straight line segment on the surface between the two vertices.

Definition. A pair of vertices in a portalgon are joined by an edge in the visibility graph if there exist a sequence of line segments in the portalgon that, when all portals are glued together, form a single straight line segment over the resulting surface.

The visibility graph of a portalgon is quite different from a visibility graph of polygonal domain. If a polygonal domain consists of multiple polygons, the visibility graph consists of multiple disjoint graphs and can be computed separately. But a portalgon consisting of multiple fragments could have visibility edges from one fragment to another.
In a normal polygon there exists only 1 straight line segment between two vertices, but in a portalgon two vertices could see each other in multiple ways and the visibility graph could therefore best be viewed as a multigraph. Another interesting property is the fact that a single vertex could see itself through a sequence of portals, which introduces loops in the visibility graph.

1.3 Thesis structure

In section 2 we describe the work done on visibility algorithms, and explain why we choose the algorithm by Lee for adaption on portalgons. In the next section 3, we define some useful structures that are used in the algorithm. After this, in section 4, we describe the adapted algorithm and prove its correctness. A short description of the implementation and the handling of degenerate case can be found in section 5. After this we perform some experiments using the implementation. The experimental setup, like the acquisition of portalgons, and the results of these experiments are described in section 6. Then we discuss some intresting phenomena, like self-seeing vertices, that happen in portalgons in section 7. We conclude in section 8.

2 Related work

There exists a number of algorithms that compute the visibility graph of a normal polygon. These algorithm differ in computational complexity from $O(n^3)$ to $O(E + n \log n)$ time (where E is the number of visibility egdes). The naive algorithm takes all pairs of vertices and checks if there exists an edge between them. This only works because every visibility edge is a straight line segment, which is not the case for portalgons. This algorithm takes $O(n^3)$ time. The algorithm by Lee[7] rotates a sweep line around each vertex and reports each visible vertex. This algorithm was the most suitable and its adaption will be explained in this thesis. This algorithm (on normal polygons) takes $O(n^2 \log n)$ time. The algorithm by Welz[13] rotates a sweep line for all vertices concurrently, and updates for each vertex what edge is visible in that direction. In a portalgon this visible edge could be a portal, and an algorithm would need to extend the search for a visible vertex after this edge. This algorithm takes $O(n^2)$ time. The algorithm by Welz and Overmars[10] is slight adaption of the previous algorithm, which makes the runtime output-sensitive: $O(E \log n)$. Finally, the algorithm by Ghosh and Mount[2] is a sweep line algorithm that iteratively adds vertices and builds the visibility graph. It is unclear how an iterative algorithm could be adapted to portalgons. This algorithm takes $O(E + n \log n)$ time.

Short summary of Lee's algorithm

The algorithm by Lee takes every vertex v and rotates a sweep line around v, while maintaining a search structure containing the edges intersected by the sweep line, sorted from the closest to v to the farthest. Every other vertex than v is an event that changes the search structure. When a vertex is closer than the closest edge, a visibility edge is reported.

In this thesis we have chosen the easiest algorithm for adaption, but this does not mean it is impossible to adapt the other algorithms.

3 Preliminaries

The portalgon and visibility graph definition can be found in the introduction. The algorithm by Lee rotates a sweep line around each vertex. The adaption for portalgons does something similar, but rotates a ray around v. This ray could hit a portal and the straight line the ray follows is extended through the portal. To describe this ray we use the following definition:

Definition. An *extended ray* is an ordered list of k (directed) line segments $s_0 ... s_{k-1}$, which form a straight line when all portals of the portalgon are glued together. See figure 2b for an example.

During the algorithm there will be one extended ray which will be denoted by $\mathcal{R}$. Instead of maintaining each ray segment explicitly, we maintain a combinatorial representation called *Extensions*.

Notation. The counterpart of a portaledge e is denoted by $\mathit{portal}(e)$.

Definition. An *Extension* $\mathcal{E}$ representing segment s_i maintains the following data describing the ray segment:

- $\mathit{start}(\mathcal{E})$: The edge on which s_i starts.
- $\mathit{end}(\mathcal{E})$: The edge on which s_i ends.
- $\mathit{source}(\mathcal{E})$: The point in $\mathbb{R}^2$ over which the segment rotates when the extended ray rotates.
- $\mathit{depth}(\mathcal{E}) = i$.
- $\mathcal{E}_T$: The transformation matrix that transforms $\mathit{portal}(\mathit{start}(\mathcal{E}))$ into $\mathit{start}(\mathcal{E})$.

Definition. The *Base* $\mathcal{B}$ is the extension describing the first segment. The Base only stores $\mathit{end}(\mathcal{B})$ and $\mathit{source}(\mathcal{B})$ and has no need for the other extension components.

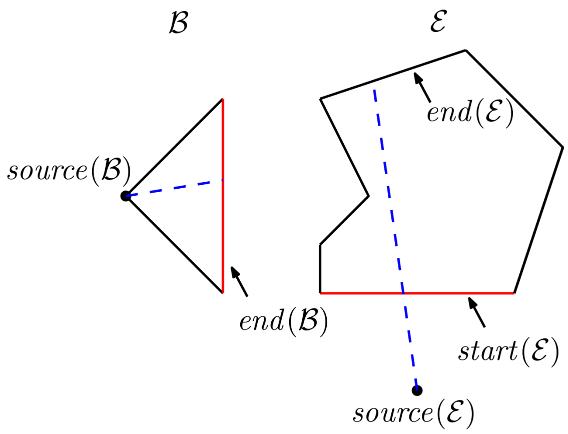
See figure 2a for an illustration. In this figure the portal in $\mathcal{B}$ must be rotated by 90° and translated over $\begin{pmatrix} 6 \\ -4 \end{pmatrix}$ to coincide with the portal in $\mathcal{E}$. And because $\mathcal{E}$ is the first extension, its *depth* is 1:

$$\mathcal{E}_T = \begin{pmatrix} 0 & -1 & 6 \\ 1 & 0 & -4 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \mathit{depth}(\mathcal{E}) = 1$$

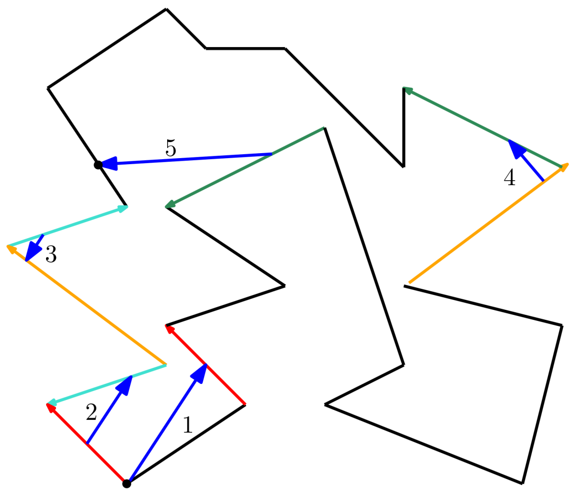
Finally some other notations for convenience:

Notation. The $\mathit{leftedge}(v)$ of a vertex v is the edge that lies on the left of v when looking towards the interior of the fragment. And $\mathit{leftvertex}(v)$ is the other endpoint of $\mathit{leftedge}(v)$.
Similar for $\mathit{rightedge}(v)$ and $\mathit{rightvertex}(v)$.

Notation. For two points u, v, by $\overrightarrow{uv}$ we denote the direction of the vector from u to v.



(a) An extended ray $\mathcal{R}$ (blue) represented by the base and an extension



(b) An extended ray in a portalgon

Figure 2

4 Algorithm

As mentioned, the algorithm will be an adapted version of the algorithm by Lee[7]. This is a sweep line algorithm where the sweep line is a ray that is rotated along a vertex v and the events are vertices which the ray intersects. As in the original algorithm, not every event corresponds to a visible vertex. In the original algorithm these invisible vertex events where used to update a binary search tree such that the next edge after a corner vertex can easily be found, see figure 3.

This method could be used in a portalgon, but there is a simpler method with an equal running time. Instead of maintaining a binary search tree, we can instead shoot a ray each time we get a corner vertex. The explanation of why this results in an equal (theoretical) running time is given in section 4.8.

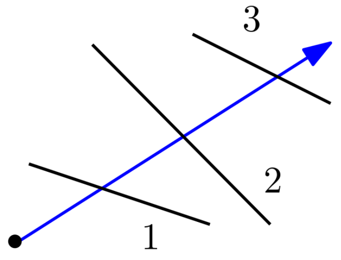


Figure 3: First step in the algorithm by Lee

The sweep line for the portalgon algorithm is not a simple ray, but an extended ray. To maintain this sweep line, we need to handle how the ray is extended through a portal. To move the sweep line we need to know what the minimum angle is the ray has to rotate in order hit a vertex somewhere along the sweep line. Because the sweep line consists of multiple ray segments which all rotate around their own point on the plane, the minimum angle for the complete ray is the minimum over all segments, see figure 4. In Lee's algorithm, the next vertex around v is easily found by just looking in the sorted list of vertices around v. But this is not sufficient for portalgons as the segments of the ray rotate around points in $\mathbb{R}^2$ that are not known beforehand.

So in order to adjust the original algorithm to work for portalgons, we need a correct way to extend the ray, and a way to determine the next vertex around an arbitrary point on the plane.

In the next sections I will explain how the complete algorithm works, starting at the top and working my down to the details.

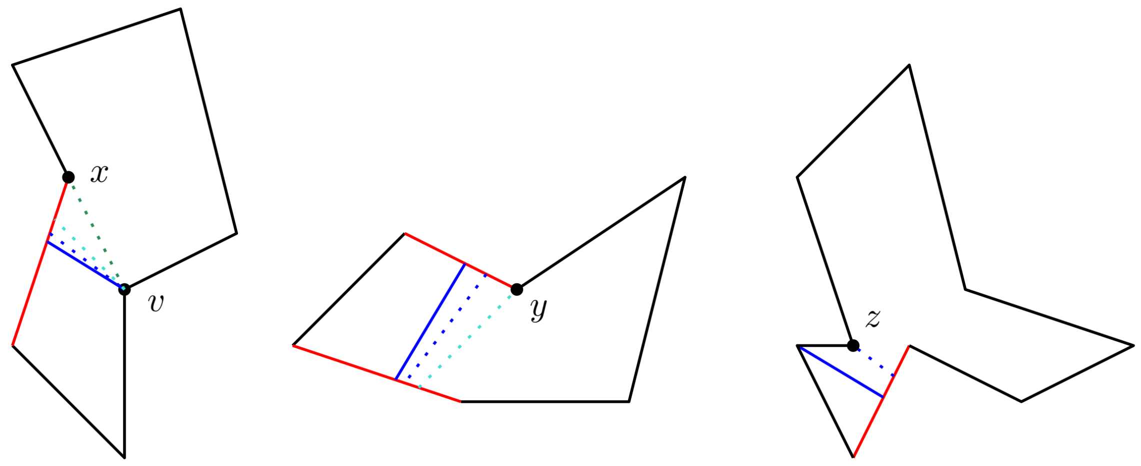


Figure 4: The next vertex around v is z

4.1 Preprocessing

Before the algorithm loops over all vertices to rotate a ray for each one, it first does a few preprocessing steps.
As in the original algorithm, it sorts for each vertex v, the vertices in the same fragment around v. In addition it computes for each fragment an arrangement of lines where the lines are the dual of the vertices in the fragment.

4.2 Rotating around vertex

The algorithm performs the same steps for each vertex and reports each vertex that is visible from that vertex.

Let v be the vertex that the algorithm rotates around.
First determine which edge is visible from v when looking in the direction along *leftedge*(v). This is done by shooting a ray R from *leftvertex*(v) with direction $\overrightarrow{leftedge(v)}$ away from v, see figure 5.

R becomes the first segment of $\mathcal{R}$. Then we rotate $\mathcal{R}$ clockwise by a minimum angle to a next vertex w. If w is visible we report (v, w). If $\mathcal{R}$ ends on a portal, we extend it through the portal. Repeat this process until $\mathcal{R}$ is rotated to the direction of *rightedge*(v).

During the rotation, $\mathcal{R}$ consists of multiple segments each described by an extension. When $\mathcal{R}$ rotates around v, the first segment s_0 simply rotates around v, but the other segments s_i each rotate around their own specific point on the plane.

Extending through a portal and finding the minimum angle are the main problems that we will explain in the next sections.

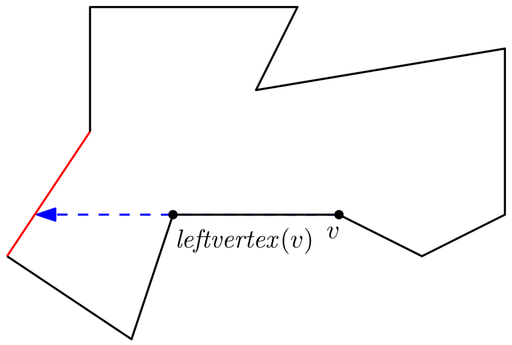


Figure 5: Finding the first visible edge

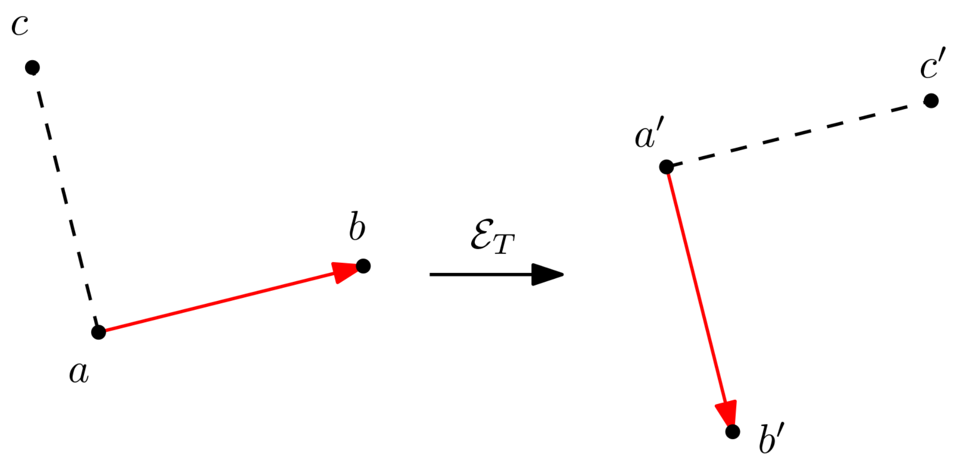


Figure 6: The portal ab is transformed to portal $a'b'$ by $\mathcal{E}_T$

4.3 Extending through a portal

When the extended ray $\mathcal{R}$ ends on a portal e, $\mathcal{R}$ is extended through the portal and we build an extension $\mathcal{E}_i$. To compute $source(\mathcal{E}_i)$ we need to find a point such that $source(\mathcal{E}_i)$ is relative to $portal(e)$ as $source(\mathcal{E}_{i-1})$ is relative to e. To find this point, we construct a transformation matrix that transforms e to $portal(e)$ in a rigid transformation, which we call $\mathcal{E}_T$. We use $\mathcal{E}_T$ to compute $source(\mathcal{E}_i)$. This matrix can be found using a system of linear equations. The two vertices a, b of e must be transformed to the vertices a', b' of $portal(e)$. To make the transformation rigid, we need a two points c, c' that are not collinear to a, b and a', b' such that $\angle bac = \angle b'a'c'$, see figure 6. These points can be constructed by taking $c = \begin{pmatrix} a_x + (a_y - b_y) \\ a_y + (b_x - a_x) \end{pmatrix}$ and $c' = \begin{pmatrix} a'_x + (a'_y - b'_y) \\ a'_y + (b'_x - a'_x) \end{pmatrix}$. These points c, c' satisfy the condition because $\overrightarrow{ac} \cdot \overrightarrow{ab} = 0 = \overrightarrow{a'c'} \cdot \overrightarrow{a'b'}$ and $\overrightarrow{ac} \times \overrightarrow{ab} = \overrightarrow{a'c'} \times \overrightarrow{a'b'}$. This gives the following system of linear equations:

$$\begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{pmatrix} \mathcal{E}_T = \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ 1 & 1 & 1 \end{pmatrix}$$

Therefore:

$$\mathcal{E}_T = \begin{pmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{pmatrix}^{-1} \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ 1 & 1 & 1 \end{pmatrix}$$

To find $end(\mathcal{E}_i)$ we compute the point p where s_i exits the portal by transforming the point where s_{i-1} enters the portal by $\mathcal{E}_T$. From p we shoot a ray in the same direction as $\overrightarrow{source(\mathcal{E}_i), p}$. We set $end(\mathcal{E})$ to the edge that the ray hits. The depth is increased by 1: $depth(\mathcal{E}_i) = i$ and we set $start(\mathcal{E})$ to $portal(e)$. When s_i ends on another portal, $\mathcal{R}$ is extended again.

4.4 Finding the next minimum angle

To compute the minimum angle $\mathcal{R}$ has to rotate we take the minimum over all extensions and the base. The minimum angle in the base is simply computed by searching in the sorted list of vertices around v. Because the rotation point in an extension is not a vertex or a point we know beforehand, we can not use a sorted list around this point as in the base. Instead we perform a search query involving the dual to find the next vertex, which is explained in the next section.

After computing the minimum angle over which $\mathcal{R}$ has to rotate to hit the next vertex w, the subroutine returns w, the direction to w, and the extension w is found in.

4.4.1 Minimum angle of an extension

To find the next vertex that a segment s_i hits when rotating the ray around v, we rotate a line around $s := \mathit{source}(\mathcal{E}_i)$ and stop a the first vertex. To solve this problem we use duality.

We use the standard duality transform Δ where vertices become lines and lines become points:

$$\Delta((a,b)) \mapsto y = ax - b$$

$$\Delta(y = ax + b) \mapsto (a, -b)$$

During the preprocessing phase of the algorithm, all vertices are transformed to lines and added to the dual into an arrangement of lines.

Let s' be the line dual of $\mathit{source}(\mathcal{E})$ and l' be the dual of the supporting line of s_i.

Lemma 1. The dual of all lines through $\mathit{source}(\mathcal{E})$ form a line in the dual coinciding with s'.

Proof. Denote the coordinates of $\mathit{source}(\mathcal{E})$ by (x_s, y_s) and denote a line k through $\mathit{source}(\mathcal{E})$ by $k : y = ax+b$. Then $\Delta(\mathit{source}(\mathcal{E})) = s' \mapsto y = x_s x - y_s$, and $\Delta(k) \mapsto (a, -b)$. $\Delta(k)$ lies on s' iff $-b = x_s a - y_s$. Because k goes through $\mathit{source}(\mathcal{E})$, we know that $y_s = x_s x + b$ which completes the proof. □

Rotating a line around $\mathit{source}(\mathcal{E})$ is therefore equivalent to moving a point along s'.

Lemma 2. Let w' be the dual of vertex w, then the primal of the intersection between s' and w' is the line going through $\mathit{source}(\mathcal{E})$ and w.

Proof. Using lemma 1, we know that points on the line w' maps to lines going through w and points on s' maps to lines going through $\mathit{source}(\mathcal{E})$. So a point on both lines must map to a line going through both $\mathit{source}(\mathcal{E})$ and w. □

Theorem 1. The first line in the dual intersected by s' starting from l' is the dual of the first vertex hit by rotating a line around $\mathit{source}(\mathcal{E})$ starting from the supporting line of s_i. Walking towards $x \to -\infty$ is rotating clockwise.

Proof. Because of lemma 2, an intersection of the line going though $\mathit{source}(\mathcal{E})$ with a vertex w dualizes to an intersection between s' and $\Delta(w)$. Rotating a line clockwise lowers its slope, which translates to lowering the x-coordinate of its primal point. □

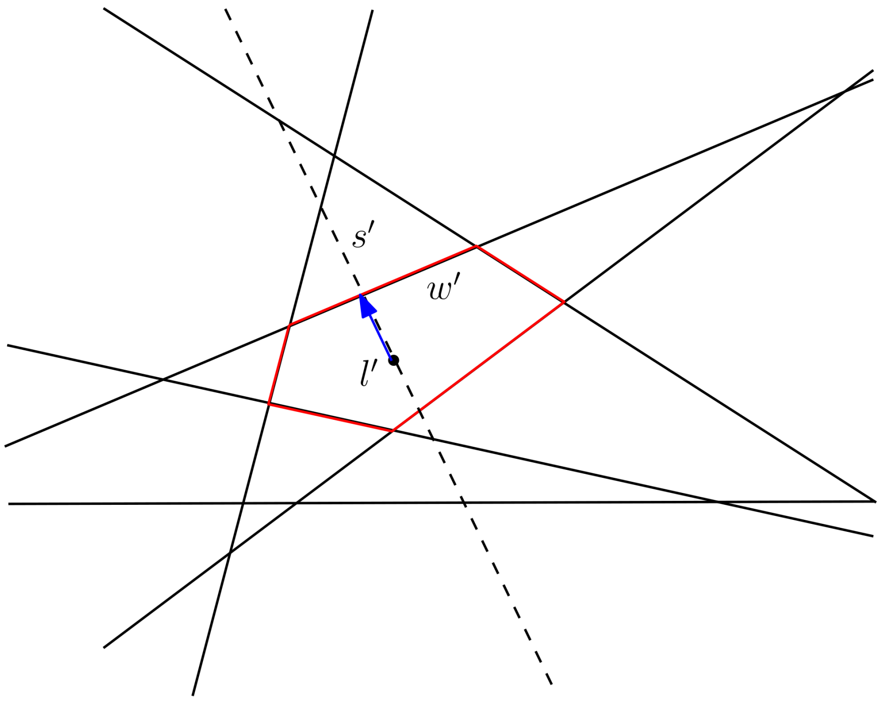


Figure 7: Traversing the red boundary will return w'

Using theorem 1 we can construct an algorithm to quickly find the next vertex s_i will hit. During the preprocessing phase after adding the dual of all the vertices, a trapezoidal decomposition is constructed to allow fast point location queries.

The main idea of the algorithm is finding the face F where l' is located, and then traversing the boundary of F and finding the line w' that s' intersects, see figure 7 and algorithm 1.

Algorithm 1 DUAL_VERTEX_HIT(s_i,$\mathcal{D}$): The next vertex the supporting line of s_i will hit.

```
$s' \leftarrow \Delta(source(s_i))$
$l' \leftarrow \Delta(support(s_i))$
$r \leftarrow RAY(source = l', support = s', direction = x \rightarrow -\infty)$
$F \leftarrow Locate(l', \mathcal{D})$
$E \leftarrow INTERSECTION(Boundary(F), r)$
if $E$ is not NULL then
  return $\Delta^{-1}(E)$
else
  return $NULL$
end if
```

To find the intersection of F and r we use a binary search performed on the boundary. This is possible because the boundary of F is convex.

Lemma 3. A face in an arrangement of lines is convex.

Proof. Assume the opposite, then there exists a face in the arrangement where there exist two points such that the line segment between them is intersected by an edge. This edge must be a line and therefore separates the two points from each other and can not be in the same face, contradiction. □

To illustrate the workings of *INTERSECTION*, we look to figure 8. We take the first stored vertex u_1 on the boundary of F. Then we perform a binary search on the rest of the vertices u_i by comparing if r lies clockwise in between $l'u_1$ and $l'u_i$. This results in two vertices u_i and u_{i+1} such that r lies clockwise in

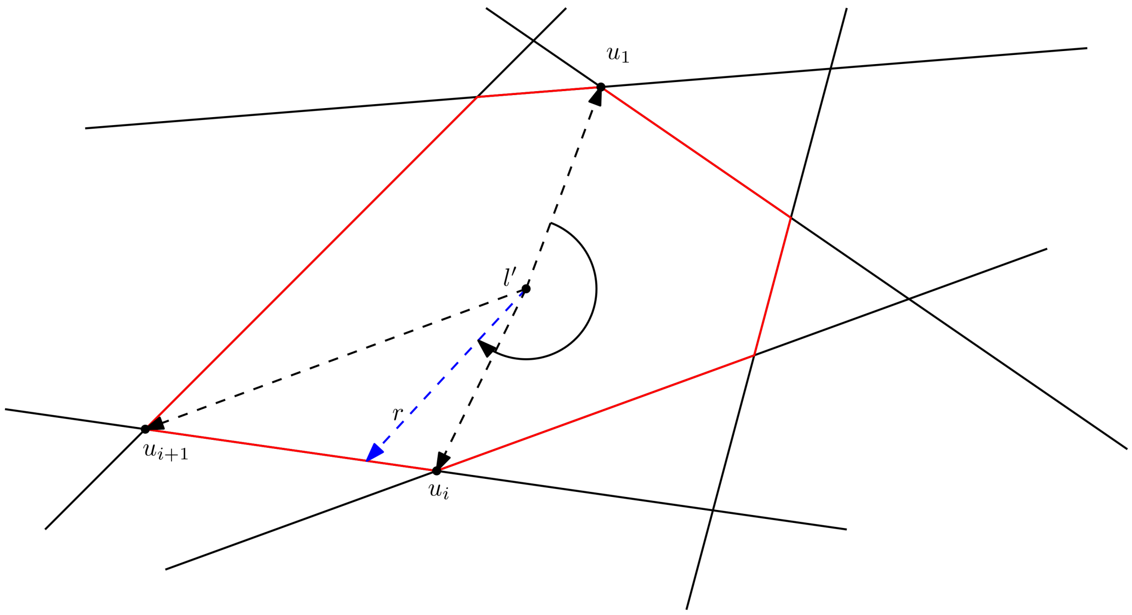


Figure 8: We compare u_i to r by which one comes first after u_1

between $l'u_i$ and $l'u_{i+1}$ and r intersects the edge u_iu_{i+1}.

It could happen that F is unbounded and that r will not intersect any edge of F. In this case, the binary search will not return an edge of F. To deal with this issue we look at what is primal equivalent of this situation. When a point along r can move indefinitely towards $x \to -\infty$, the primal line will have an increasingly downwards slope all the way to infinity.

Rotating over the vertical line $x = c$ is problematic as our dual does not have a representation for this line. But if we rotate the polygon 90° and construct a second dual, we transformed our problem to rotating over the line $y = c$ which has a simple representation in the dual.

It could happen that the second dual also does not return an edge. In that case, we can prove the following:

Lemma 4. If both queries in the dual do not return an edge, the primal line has to rotate at least 90° to encounter a vertex.

Proof. Move the ray to the origin. If the ray points towards quadrant I (top right) or quadrant III (bottom left), then the ray can rotate at least 90° before rotating over $x = 0$. If the ray points toward quadrant II (top left) or quadrant IV (bottom right), the ray points towards quadrant I or III after rotating it 90° for the rotated dual, and can therefore rotate 90° before rotating over $y = 0$. So when both queries fail to return an edge, the ray can rotate 90° without encountering a vertex. □

The complete query is described in pseudocode in algorithm 2

Lemma 5. Algorithm 2 returns an edge.

Proof. If the first two steps don't return an edge, the ray rotates 90°. If the next two steps don't return an edge, the rotates could rotate another 90°. But then the ray has rotated 180° without encountering a vertex. But the ray goes through a portal which bounds the angle the ray can rotate over by 180°. So the fourth step has to return an edge. □

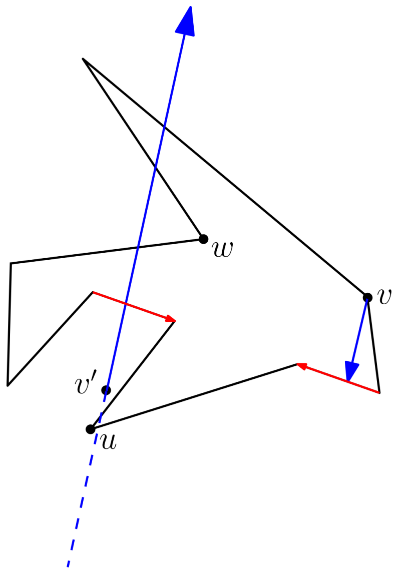
After finding w, we can easily compute the angle the ray has to rotate to hit w. But there is one small caveat in this calculation. Because we rotate a line around $source(\mathcal{E}_i)$ and not a ray, we could encounter a vertex that lies in the opposite direction of the ray, see figure 9a. This vertex will certainly not be visible but we can not immediately discard it and just return a vertex of another extension. This problem also arises when a vertex lies behind an edge.

Algorithm 2 EXTENSION_NEXT_VERTEX($\mathcal{E}_i$)

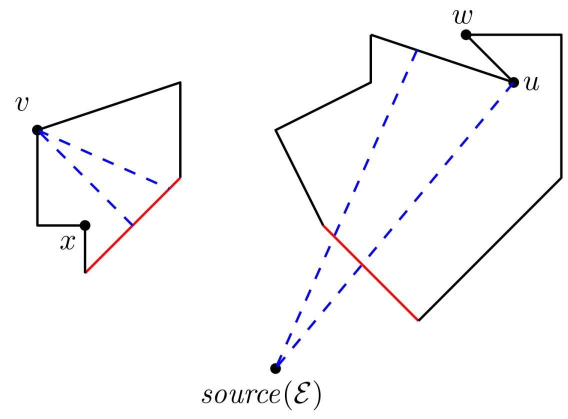
```
$\mathcal{D}$ := Normal Dual
$\tilde{\mathcal{D}}$ := Rotated Dual
$w \leftarrow$ DUAL_VERTEX_HIT($s_i$,$\mathcal{D}$)
if $w$ != $NULL$ then
    return $w$
end if
$\tilde{s_i} \leftarrow$ Rotate $s_i$ 90° around $source(\mathcal{E}_i)$
$w \leftarrow$ DUAL_VERTEX_HIT($s_i$,$\tilde{\mathcal{D}}$)
if $w$ != $NULL$ then
    return $w$
end if
$w \leftarrow$ DUAL_VERTEX_HIT($\tilde{s_i}$,$\mathcal{D}$)
if $w$ != $NULL$ then
    return $w$
end if
$w \leftarrow$ DUAL_VERTEX_HIT($\tilde{s_i}$,$\tilde{\mathcal{D}}$)
return $w$
```

In figure 9b, we see a situation where this results in a problem. In this figure we see that vertex w is the next vertex found, but is not visible. If we discard this vertex and just rotate the ray to the next vertex x in another extension, we rotate the ray further than v and we will never report it. To solve this problem, we need to rotate the line to vertex w, and than recompute the next vertex.

Checking that a vertex lies on the opposite site of the ray can be done by checking if the rotation angle towards this vertex is more than 180°. When this is the case, we subtract 180° from this angle. In this way we rotate the ray until the supporting line hits this vertex.



(a) Rotating the line around v' will hit u before w



(b) Ignoring vertex w would result in missing the visibility edge (v, u)

Figure 9

Theorem 2. Algorithm $EXTENSION\_NEXT\_VERTEX(\mathcal{E}_i)$ returns the first vertex that the supporting line of segment s_i will hit when rotating around $source(\mathcal{E}_i)$.

Proof. Combining lemma 4 and lemma 5 we know that algorithm 2 returns an edge that is dual to the closest vertex around v that s_i will hit, according to lemma 1. □

4.5 Handling the vertex

The subroutine returns the next vertex w around v, the direction δ towards w, and the extension $\mathcal{E}_i$ in which w is found. To handle this event, we first check if w is visible from $start(\mathcal{E})$. w is visible from $start(\mathcal{E})$ if there is no edge between $start(\mathcal{E})$ and w. Because we know that there is no edge between $start(\mathcal{E})$ and $end(\mathcal{E})$

When w is found in the base, the check only consist of checking whether w lies in front (or on) $end(\mathcal{E})$. When w is found in one of the extensions, the check consists of the following:

- δ is within 180° clockwise of the current direction of R.
- the interior of $(source)$ intersects $start(\mathcal{E})$.
- The $end(\mathcal{E})$ is is either attached to or lies further away than w.

The first check is to ignore all vertices that are hit by the back end of the rotating line (see 4.4). The second and third check is to ensure that the found vertex lies between the start portal and the end of the ray.

If all these criteria are met w is visible and (v, w) is reported, possibly with extra data describing the visibility path. Because $\mathcal{R}$ now ends on w, all extensions $\mathcal{E}_k$ with $k > i$ are no longer relevant and are deleted.

To prepare for the next iteration, we need to update $end(\mathcal{E}_i)$. Either the left edge is visible from v, or we shoot a ray from p in the same direction as $\overrightarrow{(source(\mathcal{E}), p)}$, see figure 10. If w is not visible, nothing happens. The direction of $\mathcal{R}$ is set to δ and the algorithm looks for the next vertex around v.

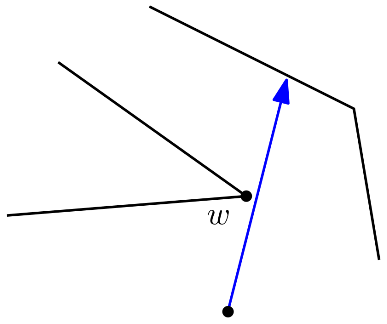


Figure 10: After rotating to w, the ray is extended further around the corner

4.6 Ray shooting

Because fragments in a portalgon are always simple polygons, we can use a technique by Hershberger[5] to shoot a ray in $O(\log n)$ time. This technique computes a Steiner triangulation of the polygon and then just iterates over the triangles until an edge is reached. Any triangulation would work, but the one presented in the paper guarantees a $O(\log n)$ query time. See figure 11 for an example.

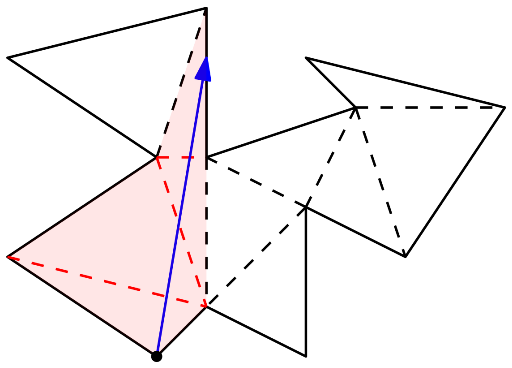


Figure 11: Ray shooting in a simple polygon

4.7 Proof of correctness

A problem

Before we prove that the algorithm is correct, we first need to address a problem. When a ray is extended the extension could hit another portal and the ray must be extended again. In some situations this could happen indefinitely and the algorithm would not terminate. The implementation has a workaround by just limiting the depth of the extensions. To completely solve this problem, we would need to examine when and how a ray can extend indefinitely. More on this in section 7.1.

Proof of correctness

Because the algorithm is bounded by a certain depth, the visibility graph it returns is a subgraph of the complete graph:

Theorem 3. Given a portalgon $\mathcal{P}$, and an integer c. The algorithm computes all visibility edges (v, u) which all consist of at most c segments each.

To prove this we first observe the following fact about visibility edges:

Observation. A visibility edge (v, u) corresponds to a ray shot from v in some direction δ that ends on u.

The algorithm rotates a ray around each vertex v and reports whenever it ends on a vertex u. As the algorithm is the same for each vertex, we only need to prove that for a vertex v, the algorithm computes all visibility edges (v, u).

Notation. $\mathcal{R}_\delta$ denotes the ray shot from v in direction δ.

When the algorithm rotates to a visible vertex, it changes the representation of the ray to represent the ray right after the vertex. For this reason we introduce the following notation:

Notation. $\mathcal{R}_\delta^+$ denotes the ray $\mathcal{R}_\delta$ rotated infinitesimally further clockwise.

Lemma 6. When the ray consists of k segments $s_0..s_{k-1}$, the algorithm returns the next vertex around v that is hit by the supporting line of one of the segments.

Proof. For a single segment s_i, the algorithm returns the next vertex the supporting line of s_i will hit according to lemma 2. Then the algorithm takes the minimum angle over all segments and return this vertex. □

We assume that the vertices are in general position, degenerate cases will be dealt with in section 5.2.

Theorem 4. For each vertex v, the algorithm rotates a ray $\mathcal{R}$ starting from v towards each direction such that $\mathcal{R}$ ends in a vertex w and consists of at most c segments.

Proof. Let $\gamma_0, ..., \gamma_k$ be the ordered set of directions around v such that a ray shot from v in direction γ_i ends in a vertex and consists of at most c segments. The ordering $\prec$ follows the clockwise ordering around v on the interval $\left[\overrightarrow{\mathit{leftedge}(v)}, \overrightarrow{\mathit{rightedge}(v)}\right]$
Let $\delta_0, .., \delta_m$ be the ordered set of directions around v the algorithm rotates to during execution. δ_0 is the direction the algorithm start with, which is $\overrightarrow{\mathit{leftedge}(v)}$. δ_m is the last direction $\overrightarrow{\mathit{rightedge}(v)}$. We need to prove that every γ_j is equal to some δ_l.

We use induction. For each i, we prove for each δ_i, the algorithm has the correct representation of $\mathcal{R}^+_{\delta_i}$, and that every for all $\gamma_j \preceq \delta_i$, there exists a $l \leq i$ such that $\gamma_j = \delta_l$.

Base case: At the start of the algorithm, the ray is shot in the direction of $\overrightarrow{\textit{leftedge}(v)}$ and is extended when it hits a portal until is has reached the maximum depth of c. This ray is a correct representation of $\mathcal{R}^+_{\delta_0}$, and there is no $\gamma_j \preceq \delta_0$.

Induction step: The algorithm has rotated the ray to δ_i and has reported every $\gamma_j \preceq \delta_i$. The algorithm now rotates to δ_{i+1}. Now suppose that $\gamma_{j+1} \prec \delta_{i+1}$, and therefore $\gamma_{j+1} \in (\delta_i, \delta_{i+1})$. Then let x be the vertex on the end of $\mathcal{R}_{\gamma_{j+1}}$.
Because the representation of $\mathcal{R}^+_{\delta_i}$ is correct and according to lemma 6, the algorithm stop at any vertex that changes the representation. But the algorithm returned δ_{i+1}, and therefore there is no vertex that changes the representation between δ_i and δ_{i+1}. So x is not between δ_i and δ_{i+1} So $\gamma_{j+1} \not\prec \delta_{i+1}$.
Therefore γ_{j+1} is either equal to δ_{i+1} or $\delta_{i+1} \prec \gamma_{j+1}$.
If $\gamma_{j+1} = \delta_{i+1}$, then the ray is cut off after x and is possibly extended after rotating to $\mathcal{R}^+_{\delta_{i+1}}$. If $\delta_{i+1} \prec \gamma_{j+1}$, then the representation of the ray does not change because only a visible vertex changes the representation. In either case, the representation stays correct and for all $\gamma_{j'} \preceq \delta_{i+1}$, there exists a $l \leq i+1$ such that $\gamma_{j'} = \delta_l$. This proves the induction step.
Because $\gamma_k \prec \delta_m$, there must exist a δ_l for all γ_j such that $\gamma_j = \delta_l$. This concludes the proof. $\square$

4.8 Computational complexity

We analyze the runtime of the algorithm that computes the visibility graph where all edges consist of at most c segments. A portalgon can consist of multiple fragments, each with their own preprocessed dual and sorted lists of vertices. To simplify the runtime analysis, we first consider portalgons consisting of a single fragment.

Preprocessing

Let's call the number of vertices in the fragment n. Sorting a list of n vertices takes $O(n \log n)$ time. This is done for each vertex, so this takes in total $O(n^2 \log n)$ time. The construction of the dual consists of adding n lines to an arrangement, which takes $O(n^2)$ time in total. After the construction phase, a trapezoidal decomposition of the arrangement is made to facilitate the point location queries. Making a trapezoidal decomposition of an arrangement with m complexity takes $O(m \log m)$ time. So it takes $O(n^2 \log n)$ time to construct a trapezoidal decomposition of the dual.

Rotating the ray

The compute the visibility of a single vertex, we first shoot a ray and then rotate it multiple times to the next vertex. Shooting a ray in a simple polygon takes $O(\log n)$ time using Hershbergers technique[5]. Before each rotation we perform a query on each ray segment to find the next vertex the ray segment will hit. This query consists of a point location, followed by a binary search on the boundary of a face in the dual. A point location query using a trapezoidal decomposition takes $O(\log n)$ time, and a binary search on binary search also takes $O(\log n)$ time. So each query takes $O(\log n)$ time per extension. In each step we do not need to recompute the next vertex for each extension, only for the one that returned the closest vertex in

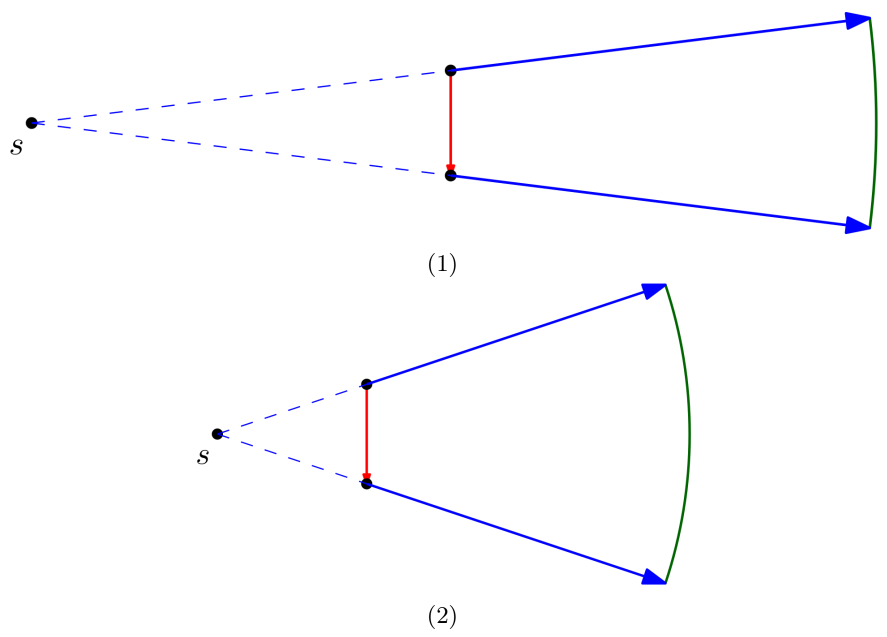


Figure 12: The further s lies from the portal, the smaller the visible area becomes

the previous step. This means that each step takes $O(\log n)$ time.

To calculate the number of steps the algorithm performs for each vertex, we need to calculate how many vertices the ray could rotate to. We can calculate an upper bound for this, but this bound is probably not very tight. In the worst case, the portalgon consists of only portals and the ray always extends to c segments. Take the first segment which rotates around the starting vertex and sees $O(n)$ portaledges. Then all the $O(n)$ second segments could also see $O(n)$ portaledges, and so on. This would result in $O(n^c)$ events while rotating the ray around a single vertex and therefore in a total runtime of $O(n^{c+1} \log n)$.

This upper bound assumes that the second, third... segments all see $O(n)$ portaledges. But as the ray extends further, the $\mathit{source}(\mathcal{E}_i)$ for each segment s_i moves further away from $\mathit{start}(\mathcal{E}_i)$ and the angle over which s_i rotates decreases and therefore decreases the chance the ray will hit a vertex, see figure 12.

Also the sum of these angles for each s_i with the same depth is upper bounded by $180°$. These observations lead to the suspicion that the number of events is more tightly bound than $O(n^c)$.

The implementation described in the following sections could provide a useful insight in this bound.

4.9 Multiple fragments

Suppose the portalgon consists of k fragments $\mathcal{F}_i$, $0 < i \leq k$. Denote n_i as the number of vertices in fragment $\mathcal{F}_i$. The preprocessing time for these fragments is then $O(\sum_{i=1}^{k} n_i^2 \log n_i)$. Because the ray can extend into every fragment, the number of events and the query time is bounded by the fragment with the most vertices. Let F_m be the fragment with the most vertices. Each query can take $O(\log(n_m))$ time. And the number of events is bounded by $O(n_m^c)$. This brings the total runtime to $O(n(n_m^c \log(n_m))$.

5 Implementation

The algorithm from section 4 is implemented for exploratory analysis. The implementation is written in c++17 and uses the CGAL[11] library (version 5.3) and the Eigen[3] library (version 3). To prevent floating point errors, CGAL provides multiple exact kernels. The implementation uses the Exact Rational kernel (see [4]). This kernel represents the results of division (and other operations) as a fraction represented by an integer numerator and denominator. But this kernel does not provide an exact representation for results of the sine and cosine functions, which complicates the calculation for extending the ray through a portal. To deal with this problem, the implementation uses a rigid matrix transformation which translates and rotates a portaledge e_1 onto $portal(e_1)$. Calculating this matrix can be done by solving a set of linear equations, defined by the translation from the start and end of e_1 to the start respectively end of $portal(e_1)$ and a third point to retain a rigid transformation. To solve these equations, we use the Eigen library.

Furthermore, the implementation uses multiple other CGAL modules and functions to represent the polygon, the dual and the triangulation:

- The arrangement module [12] for representing the polygon and the dual. Each with extra annotations to vertices, edges or faces. This module also allows point location queries using a trapezoidal decomposition.
- The triangulation module [14], which computes the constrained Delaunay triangulation of a fragment to represent the ray shooting data structure.
- Small functions to handle transformations, compute intersections, etc.

5.1 Achieved complexity

The theoretical runtime for the ray shooting query is $O(\log n)$ time, but this requires a triangulation as described in 'Shoot a ray, take a walk'[5]. During the implementation we choose to just use a simple triangulation. In the worst case, a ray shooting query could now take $O(n)$ time. To improve the implementation the triangulation method needs to be replaced by Hershbergers triangulation, and the ray shooting method needs to be adjusted to deal with the steiner points.

The implementation is also not using the caching technique described in section 4.8 that would reduce the query time over all extensions. The achieved runtime for each step is therefore $O(c \log n)$ time instead of $O(\log n)$ time.

5.2 Collinearity and degeneracies

As with all geometric algorithms, collinearity and degeneracies is a problem that the algorithm needs to handle. We will discuss the problematic cases that could happen in the dual query and the cases outside of the dual separately.

5.2.1 Degeneracies in the primal

A degenerate case in the primal arises when the extended ray goes through two vertices at the same time. This happens when there are three vertices of the polygon that lie on the same line, but could also happen when the *source* of an extension lies on the same line as two vertices of the polygon. Before we handle this case, we need to choose how we define visibility. In figure 13 we see two vertices that lie on the same line as the ray. Vertex w lies behind vertex v, and to make the algorithm easier, we choose that only the first vertex v is visible. When 3 vertices lie on one line we need to ensure that we only handle the first vertex and discard the rest. To achieve this, the sorted list around a vertex puts the closest vertex before the others, so that only this vertex would be returned in the search query for the base extension, see section 4.4. The correctly deal with this problem in an extension, the dual query is adjusted, see section 5.2.3.

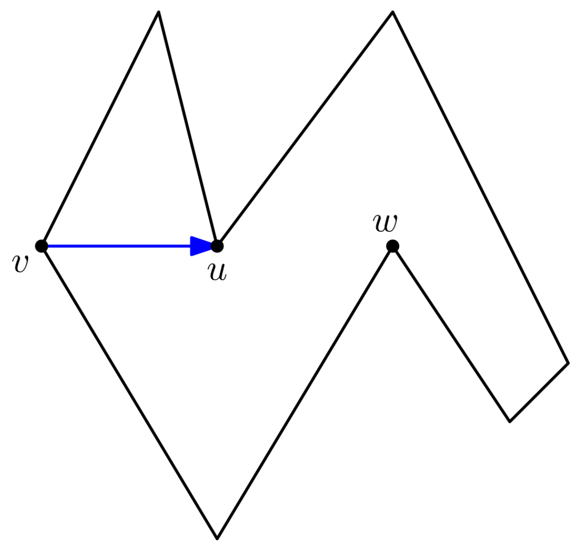


Figure 13: v, u and w are collinear, but v does not see w

When the algorithm performs a ray shooting query and the result is a vertex v (instead of an edge), we check if *leftedge*(v) lies on the right of the ray. In this case, we just return *leftedge*(v), otherwise we continue the ray shooting query from v.
When the ray extends through a portal from a vertex of the portaledge, we need to check if the ray could extend through the portal, and is not blocked by the adjacent edge, see figure 14. In this case, no ray shooting query is performed and *leftedge*(v) is set as *end*$(\mathcal{E})$, where v is the left vertex of the portal. When *leftedge*(v) is parallel to the ray, the ray extend alongside *leftedge*(v) and a ray shooting query is performed from the left vertex of *leftedge*(v).

5.2.2 Non-zero measure events in the dual

Even when the vertices are in general position, there are still some cases for which one would expect to happen only with measure 0, but the algorithm inherently constructs these cases.

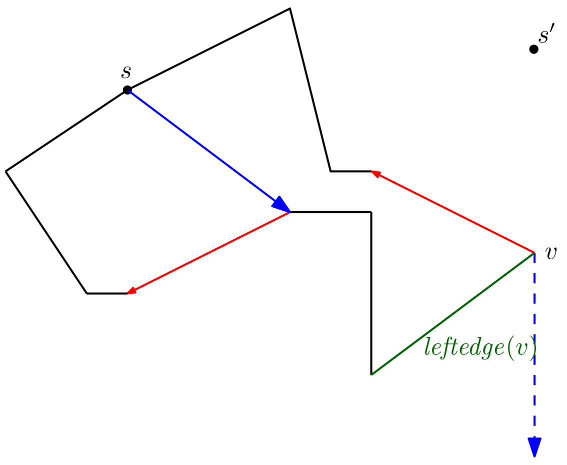


Figure 14: The ray extension directly hits *leftedge*(v) and no ray shooting is needed

As mentioned, a query in the dual start by locating a point in the arrangement of lines. The algorithm describes the case in which the point is located on a face. But it could happen that it is located on an edge or a vertex. First let's look at what that means in the primal.

The point in the dual is equivalent to the initial ray in the primal. An edge in the dual is a vertex of the polygon in the primal. So when the point is located on an edge in the dual, the initial ray (or the opposite) goes through the vertex in the primal. This seems like an event that happens with measure 0, but during the algorithm the ray is rotated to the next vertex around v and the initial ray in the next step goes through this vertex. So this event happens even when the vertices are in general position, so we need to be able to handle these events even in the most basic implementation.

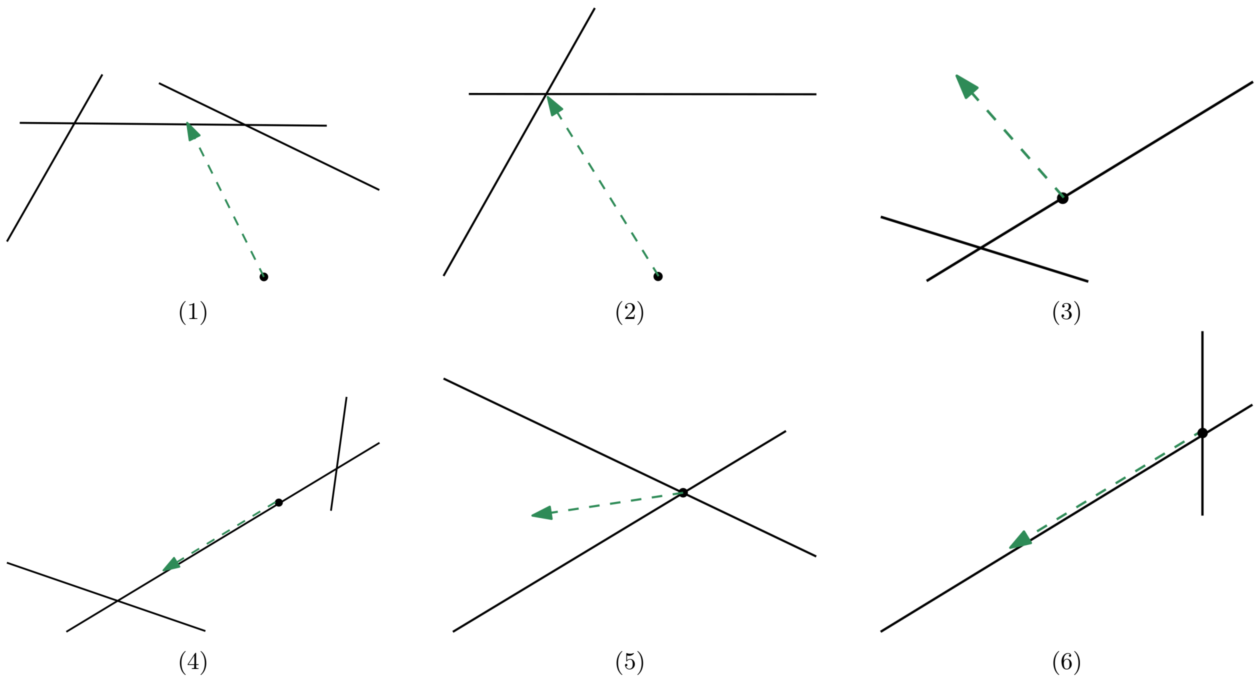


Figure 15: Possible cases in a dual query

5.2.3 Degeneracies in the dual

When the vertices are not in general position, there are multiple scenarios for the query in the dual.
Let r be the ray the dual, with s being the source of r. The query can lie on a face, edge or vertex and have a certain direction. We distinguish 6 cases:

1. s lies in the interior of a face, and r hits an edge.
2. s lies in the interior of a face, and r hits a vertex.
3. s lies on an edge, and r points towards an incident face.
4. s lies on an edge, and r points in the same direction as the edge.
5. s lies on a vertex, and r points towards an incident face.
6. s lies in the interior of a face, and r points in the same direction as an incident edge.

Case 1: The query lies on the interior of a face and hits an edge. This is the base case and is explained in section 4.

Case 2: The query lies on the interior of a face and hits a vertex. Because the vertex is an intersection point of 2 or more edges this is equivalent to hitting 2 (or more) vertices with the rotating line in the primal at the same time. Because we want to handle the closest visible vertex, we need to determine which of these vertices are visible and which is the closest. Determining if a vertex w is visible can be done as described in section 4.5. Finding the closest visible vertex can be done by simply looping over all these vertices. The theoretical runtime mentioned in section 4.8 is dependant on the number of vertices that cause an event. Because only vertex is the closest, only one event is triggered and a loop over these vertices does not increase the theoretical runtime.

Case 3: The query starts on an edge, and is directed towards the interior of an incident face. This happens when the primal ray starts in a direction in which it already hits a vertex before moving. This case can easily be dealt with by just performing the normal binary search on the face that the query walks into. This could then result in Case 1 or 2. As mentioned in 5.2.2, this case also happens when the vertices are in general position.

Case 4: The query lies entirely on an edge in the dual. This happens when the primal ray rotates around a point that coincides with a vertex of the polygon. Because we will ignore this vertex (it is not visible), we want to find the first edge in the dual that the ray properly intersects. A possible solution would be to find the combined boundary of the two incident faces and perform binary search on that. Another solution is just to find the vertex at the end of the edge in the dual, and perform the same trick as in case 2.

Case 5: The query start on a vertex and moves inwards to the interior of a face. This is a variant of case 3, and the solution is the same. We just find the face the query walks into, and perform binary search on this face.

Case 6: The query starts on a vertex and lies entirely on an edge incident to this vertex. This is a variant of case 4, and the solution is the same, just find the next vertex incident to the edge and perform the same trick as case 2.

6 Explorative data analysis

Using the implementation of the visibility graph algorithm, we can explore the algebraic surface described by the portalgon. Because the concept of portalgons is very new there are not many portalgons to explore. First we need to generate portalgons and then we can explore the surfaces.

6.1 Experimental setup

We use two ways of obtaining portalgons. First we use already generated polygons from the Salzburg Database of Geometric Inputs[1] and transform those into portalgons.

Second is a basic generation technique using a Delaunay triangulation of random generated points. This method also generates polygons that are then transformed into portalgons. The problem of transforming polygons to portalgons lies in the fact that portals are pairs of edges of the same length. Not every polygon can be transformed into an interesting portalgon, because portals are pairs of edges of the same length. Some polygons have all different length edges, and therefore no edges can become a portal.

After generating the portalgons, we run the algorithm and report multiple metrics. Interesting metrics include:

- The number of visibility edges.
- The depth of visibility edges (the number of ray segments).
- The length of visibility edges.
- The total angle β_i over which the ray has depth i.

6.2 Generating portalgons

6.2.1 Salzburg database

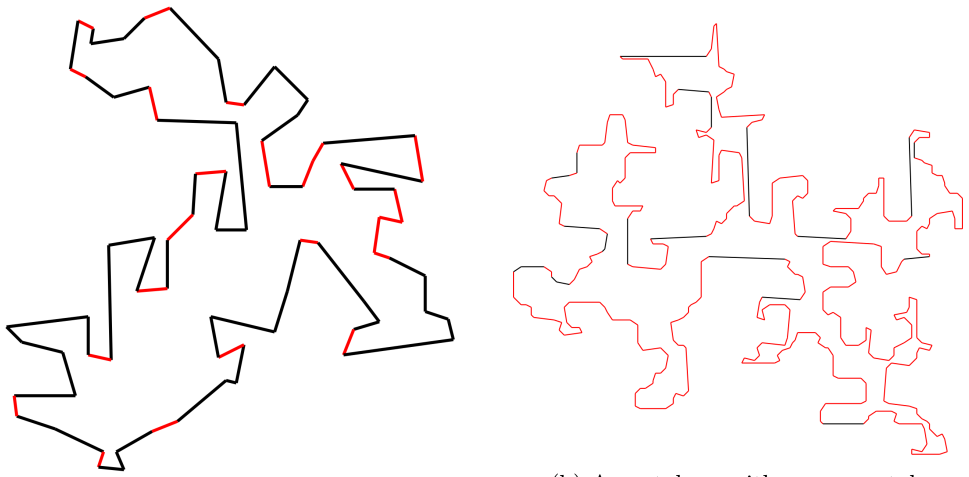
As mentioned, the first method uses pre-generated polygons from a database. The problem with these polygons is that the vertices have floating point coordinates. This results in almost all unique edge lengths, and are therefore not suitable for forming a portal pair. A possible solution to this would be to split an edge into two two collinear edges such that one part is the same length as another edge. But this would results in an inexact representation of the new vertex. And an inexact representation could in turn result in unexpected errors in the implementation.

Another solution is to round all vertices to the nearest integer coordinate. All edge length would then be the square root of an integer and the possibility of two same-length edges increases. It could happen that snapping to grid points results in a non-simple polygon. This could happen by snapping two vertices to the same point, or by snapping in such a way that the polygon overlaps itself. Scaling the vertices before snapping reduces this problem, but it also decreases the chance of same length edges.

Using these criteria, there was only one type of polygons in the Salzburg Database[1] that resulted in portalgons with a large number of same length edges, namely polygons created by the Super Random Polygon Generator.

After snapping the vertices, we can match each two edges with the same length into a portal pair. This matching resulted in portalgons where up to 90% of the edges where portals. Scaling the vertices before snapping lowered the number of portals, to almost none.

For the experiments we used the polygons with less than 400 vertices. Larger polygons resulted in an unfeasable runtime. Using different scales, we obtained many portalgons with a wide range in the number of portals. Portalgons where more than 90% of the boundary is a portal were more sporadic, but using a low scale we obtained at least 10 portalgons with this property.


(a) A portalgon with a few portals

(b) A portalgon with many portals

Figure 16: Two portalgons from the Salzburg database

6.2.2 Delaunay Triangulation method

Because of the lack of random polygons suitable for the generation of portalgons we decided to design our own algorithm to create polygons with integer coordinates.

Start with a set of n randomly generated points with integer coordinates and compute the Delaunay triangulation of these points. The convex hull of these points is the start of the polygon.

Then we repeatedly take an edge of the polygon and include/exclude one of the two triangles incident to this edge. We can either include the triangle that is not inside of the polygon, or exclude the triangle that is inside. In some cases such a step can decrease the number of vertices of the polygon (figure 17a), which is sometimes not desirable. As the polygon starts out as only the convex hull of all the vertices we need to add more vertices than remove them.

To tweak the algorithm we need 3 binomial parameters:

- p_{in}: The chance we exclude the inner triangle.
- p_-: The chance we decrease the number of vertices of the polygon (when we encounter such a case).
- p_+: The chance we increase the number of vertices of the polygon (when we encounter such a case).

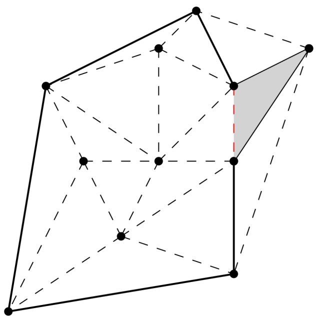
In algorithm 3 we can read the pseudocode for this algorithm: Using this algorithm we can generate random polygons, but there are some restrictions on these polygons. First, edges are not very long (in comparison to the whole polygon) because the edges of the polygon are also edges on the constrained Delaunay triangulation.

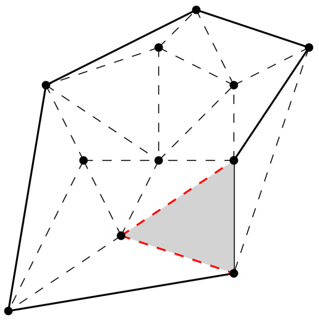
Tweaking the parameters to create an interesting set of polygon in a task on its own. A particular set of parameters that created polygon containing a sufficient number of vertices and an interesting boundary can be found in table 1.

This set of parameters generates portalgons like figure 18. These portalgons have around 300 vertices and a boundary where up to 40% is a portal. Further tweaking of the parameters could generate different sized portalgons, but tweaking is a hard task on its own.

n	500
gridsize	500x500
#steps	2000
p_{in}	0.8
p_-	0.2
p_+	1

Table 1: Parameters used for figure 18


(a) Removing the triangle would reduce the number of vertices


(b) Removing the triangle would increase the number of vertices

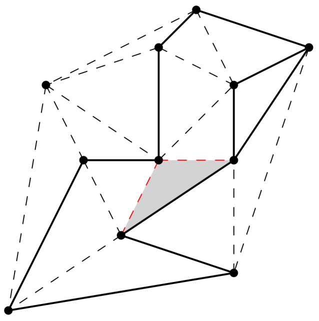

(c) Removing the triangle would make the polygon non-simple

Figure 17

Algorithm 3 Generate random polygon using a Delaunay triangulation

```
Generate $n$ random points
Compute Delaunay Triangulation
$P \leftarrow CH(points)$
for $i = 0$ to #steps do
  $e \leftarrow RandomEdge()$
  if $rand(0,1) < p_{in}$ then
    $t \leftarrow InnerVertex(e)$
  else
    $t \leftarrow OuterVertex(e)$
  end if
  if $t \in P$ and $t$ is not incident to $e$ then
    continue
  end if
  if $t \in P$ then
    if $rand(0,1) < p_-$ then
      $e_1 \leftarrow$ Vertex of $e$ adjacent to $t$
      $e_2 \leftarrow$ Vertex of $e$ not adjacent to $t$
      Replace $\{e$ and $Edge(e_1, t)\}$ by $Edge(e_2, t)$ in $P$
    end if
  else
    if $rand(0,1) < p_+$ then
      $e_1, e_2 \leftarrow$ Vertices of $e$
      Replace $e$ by $\{Edge(e_1, t)$ and $Edge(e_2, t)\}$ in $P$
    end if
  end if
end for
```

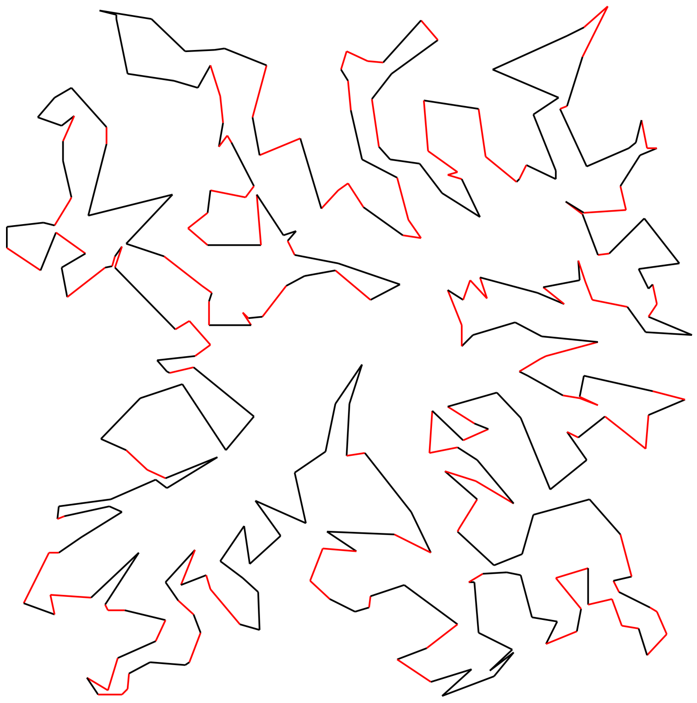


Figure 18: Portalgon generated using algorithm 3

6.3 Observations

The constructed polygon differ mostly in the portion of portaledges, portalgons generated by the Salzburg database can have a percentage of portals ranging from 0% to 95%, where as the Delaunay generated portalgons have a percentage of portals of 0% up to 45%. Especially the portalgons created by the Salzburg database are interesting for analysing the upper bound of events as mentioned in 4.8 due to their high percentage of portals. In the following sections we plot multiple metrics to the *depth* of a visibility edge. To ease the analysis of the portalgons we introduce three new definitions for metrics:

The first is the **depth** of a visibility edge, which is defined as the number of segments minus 1. For example, a visibility edge that does not cross a portal has a depth of 0. This definition corresponds to the *depth* of an extension.

The second is called the **portalfraction** of a portalgon which is the portion of the boundary length that is a portal.

The third is based on the interior angle. Define $\Gamma_i(v)$ as set of rays shot from v that can extend at least i times. And define r_0 as the first ray segment of r. Then we take the set $\{\overrightarrow{r_0} | r \in \Gamma_i(v)\}$, which is a set of intervals of directions. Then we define $\beta_i(v)$ as the sum over these intervals using the measure operator μ: $\beta_i(v) = \mu(\{\overrightarrow{r_0} | r \in \Gamma_i(v)\})$.

Then define β_i as the sum of $\beta_i(v)$ over all vertices v. We call β_i the **Visible angle of depth** i.

The total interior angle is the sum of the interior angle for all vertices. In a single polygon of n vertices the interior is thus $(n-2) * 180°$. So $\beta_0 = (n-2) * 180°$ in a polygon of n vertices.

6.3.1 Comparison between Salzburg portalgons and Delauny Portalgons

In figure 19 we plot the distribution of visibility edges compared to the depth of the edge for 50 portalgons created by the Delaunay triangulation method. Because the total number of visibility edges are very different across multiple portalgons, we have decided to normalize the number of edges. We take the maximum number of edges that have the same depth, and then divide all numbers of edges by this maximum. This results in graphs that have a maximum of 1. In this way we can plot multiple portalgons in the same figure, and compare the distributions.

The color of a line indicates the portalfraction for each portalgon. A red line indicates a very low ($< 10\%$) portalfraction, and a blue line a very high ($> 90\%$) portalfraction. Because the Delaunay method only generates portalgons with a portalfraction up to 40%, there are no blue lines.
In figure 20 we see a similar graph or portalgons created by the Salzburg database. The main difference to figure 19 is the number of blue lines that are present. These lines seem to follow different distributions. In figure 21 we see another graph of the Salzburg database, but this time the portalgons all have a portalfraction less than 45% so we can compare the two methods.

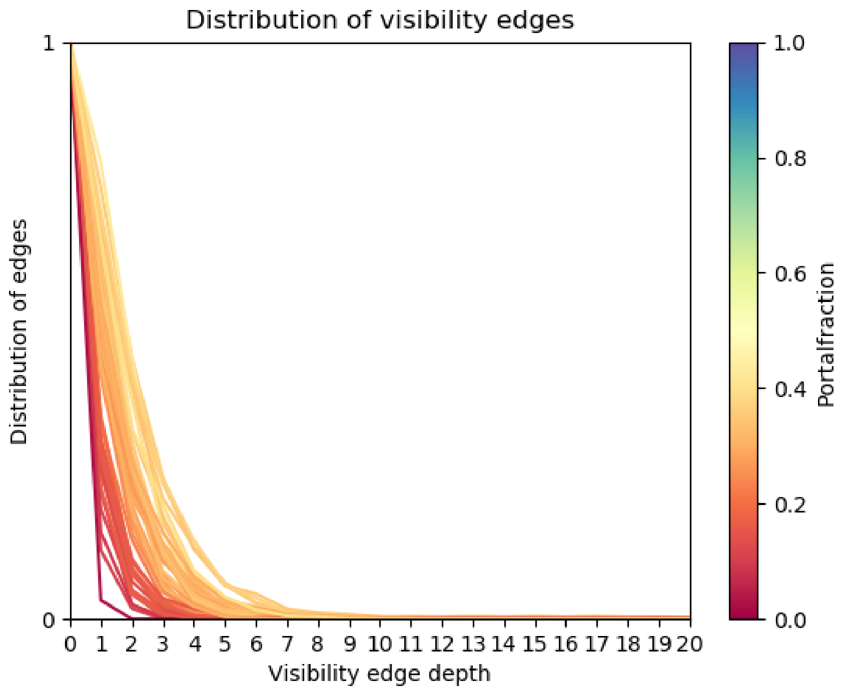


Figure 19: 50 portalgons generated using the Delaunay method

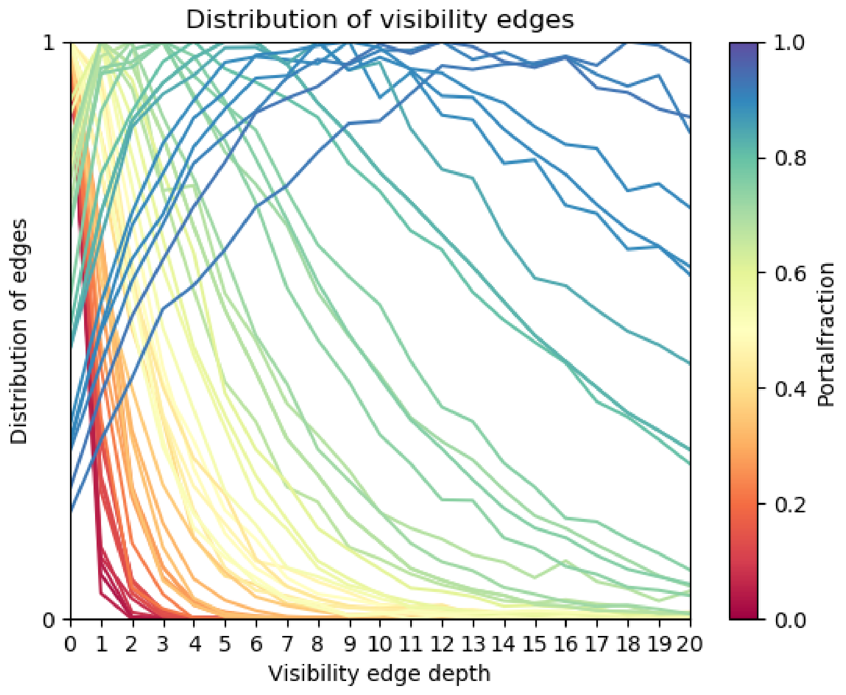


Figure 20: 50 portalgons from the Salzburg database

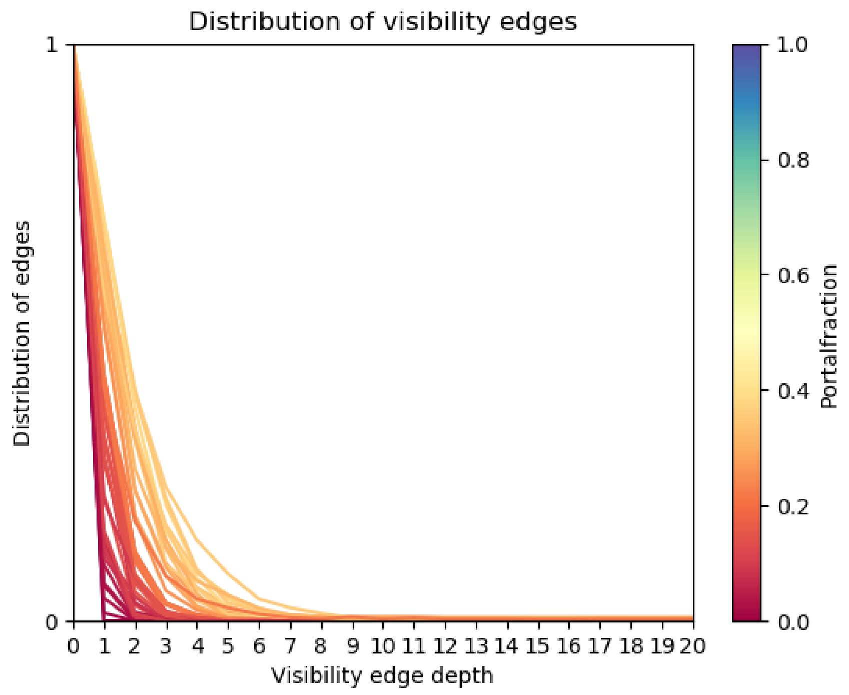


Figure 21: 50 portalgons from the Salzburg database with a maximum portalfraction of 45%

The two graphs rapidly decrease, and show almost no visibility edges with a depth of 6 or more. The distributions that are skewed to the right are the same color in both figures, meaning that they have a comparable portalfraction. So we can assume that there is no big difference in the visibility graphs of both types of portalgons. Because the Salzburg database also generates portalgons with a higher portalfraction, the rest of the analysis is done on those portalgons.

6.3.2 Number of visibility edges

To truly analyze the correlation between visibility edges and the depth of these edges, we need the Salzburg database, so that we can analyze what happens with a higher portalfraction. In the following sections all portalgons are created from the Salzburg database. From figure 20 we see that portalgons with a higher portalfraction have a very different distribution than the ones with a lower portalfraction.

In some cases, the distribution is so skewed that the top of the distribution does not lie at depth 0, some distributions even have their maximum at depth 12. This increase in visible vertices was somewhat expected in section 4.8, but a more precise explanation could be given combining two other metrics, the length of these edges and the extend depth.

6.3.3 Length of visibility edges

In figure 22 we plotted for 50 portalgons the average length for visibility edges of a certain depth. The color-coding (red-blue) is the same as in figure 20. Because different portalgons can have a different scale, we normalized the length to the length of the total boundary.

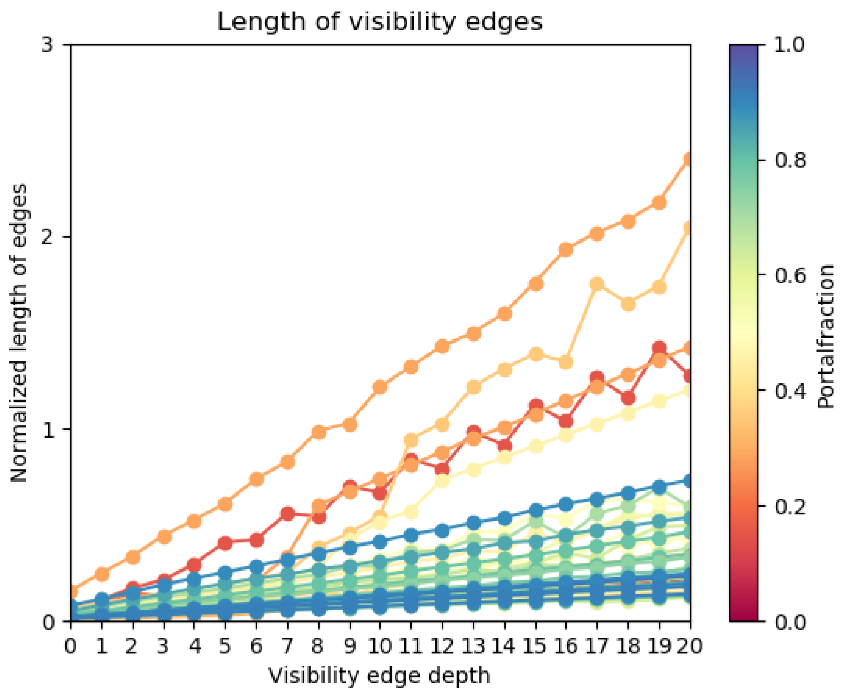


Figure 22: Length distribution of portalgons

Most lines seem to have a very linear correlation to the depth of the edge. This could be explained as follows: The visibility edge consists of multiple ray segments which are all segments from boundary to boundary. And so visibility edges of depth δ will have an average length of δ times the average distance from some point on the boundary to a point on the overlying boundary.

The graph also shows some lines which do not adhere to the linear pattern, but are instead alternating between a higher and a lower length around a certain slope. Further investigating these portalgons leads to an interesting phenomenon which will be discussed in section 7.1

6.3.4 Visible angle per depth

To plot the visible angles of a portalgon, we use a log-plot. In figure 23 we see the visible angle β_i for each depth i for 50 portalgons, again with the same color coding.

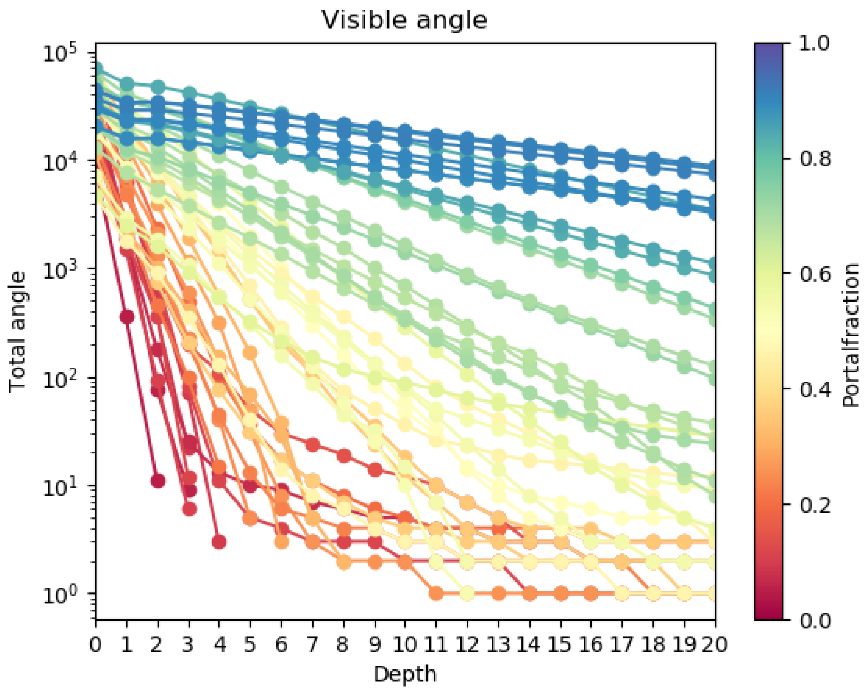


Figure 23: Level set size per depth level

The straight lines clearly indicate an exponential decrease in the visibility angle per depth. This correlation is most noticeable in portalgons with a high portalfraction. The portalgons with a higher portalfraction also have a lower decrease in angle measure than those with a lower portalfraction.

The visible angle for seem to decrease by some factor c per depth level, which could be explained as follows: A constant portion of the boundary is a portal, and the ray only extends when it hits a portal. Suppose that the ray has a uniform possibility for hitting each part of boundary. Then it has a chance equal to the portalfraction of extending. This results in an exponential decrease with a rate of chance equal to the portalfraction.

It is therefore reasonable to assume that the angle measure has a rate of change $c < 1$, which is closely related to the portalfraction.

6.3.5 Correlation between depth and visibility edges

Combining these two metrics could explain the correlation between depth and visibility edges. Take for example a collection of uniformly distributed points and a point p in the middle from which we measure the number of points x in a sector for which $a \cdot \epsilon < d(x, p) < (a + 1) \cdot \epsilon$ for some a and a small ϵ. This number is linear related to a and the angle of the sector.
This is similar to visibility edges from a single vertex, where a is the depth and ϵ is the average width of the portalgon. The angle over which the ray rotates where the ray is at least a deep, is the extend depth. This

would mean that the number of visible vertices is correlated to the length times the extend depth, which could be described using the following formula:

$$\#\,Vis(\delta) = O\left((a\delta + b)c^{\delta}\right)$$

Where δ is the depth, $a\delta + b$ is the trend line for the length of the edges and c is the portalfraction (see the above section). Note that the trend line for the length has an intercept term b, which probably is very small.

In figure 24, we can see the distributions formed by the above formula where $a = 0.015$, and $b = 0.02$, which are the parameters for the average trend line for the length of visibility edges. These plots are also normalized to their maximum.

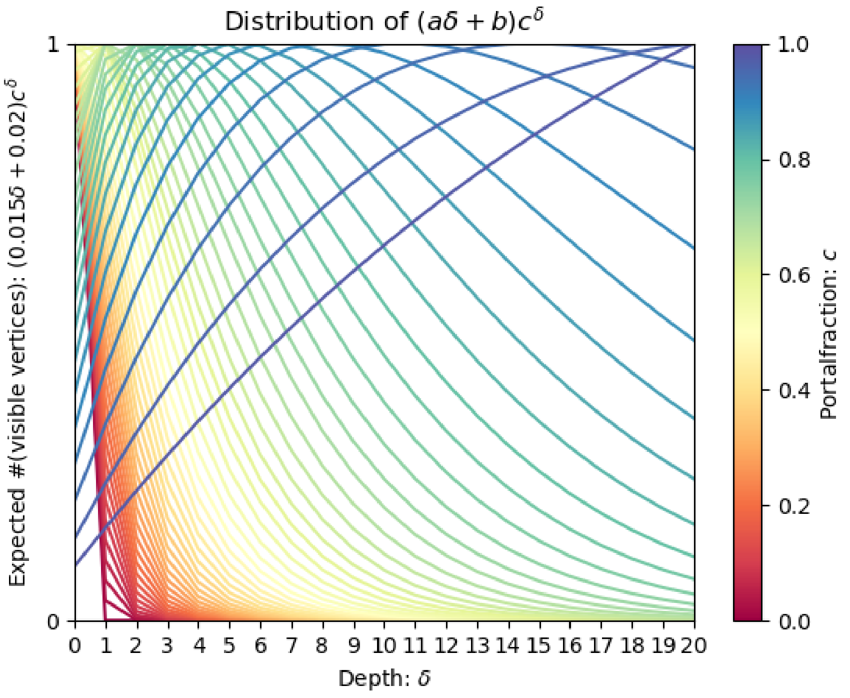


Figure 24: Distribution of $(0.015\delta + 0.02)c^{\delta}$

The distribution seem similar to figure 20, which gives reason to believe that the number of visible vertices is linearly dependant on the depth when the portalfraction is 1. But to truly analyze the runtime mentioned in 4.8 we need to count the number of events, as this metric determines the runtime.

6.3.6 Number of events

To analyze the runtime as mentioned in 4.8, we need to count the number of events for each depth and see if there exists an exponential correlation. In figure 25 we can see a similar graph to the number of visible vertices, with some differences. But these differences do not indicate an exponential correlation, as was to be expected according to section 4.8.

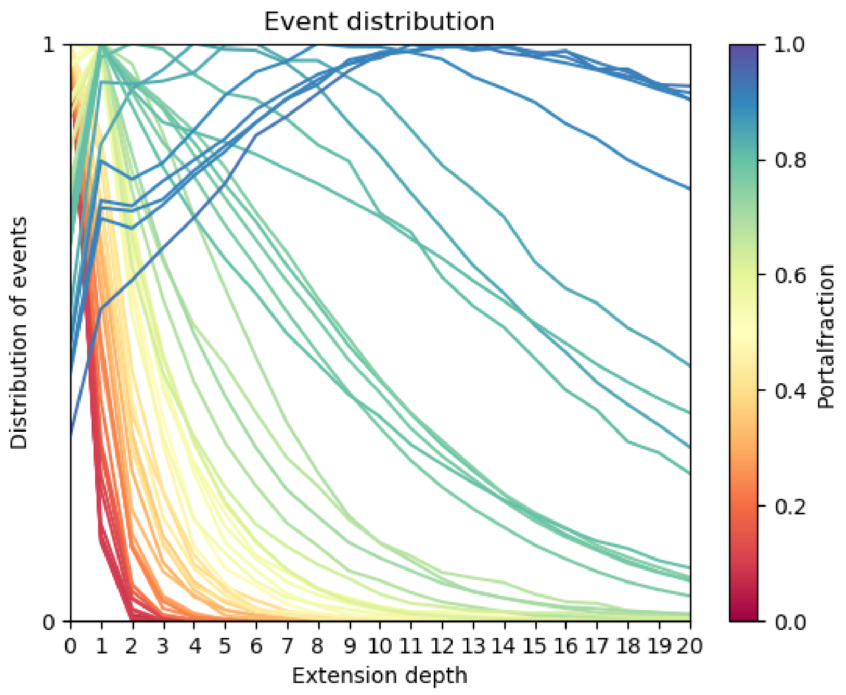


Figure 25: Distribution of events for 50 portalgons

In figure 26, we see the number of events plotted against the size of the portalgon. As expected, the number of events is higher for portalgons with a high portalfraction. Section 4.8 suggests that the number of events is proportional to $O(n^c)$ when the portalfraction is 1. The experiments are done with $c = 2$, which would results in $O(n^{20})$. This is clearly not the case. The low number of portalgons with a high portalfraction makes it difficult to see what the relation truly is.

The black line is the formula $7.5n^2$ (where n is the number of vertices), which fits the portalgons with a portalfraction of around 70%, this correlation is also visible in figure 27, which only shows the portalgons with a portalfraction lower than 40%. In this figure, the black line is the formula n^2. This line fits the portalgons with a portalfraction of around 20%. Both lines indicate that the number of events is quadratic in the number of vertices, and that the portalfraction plays a minor role.

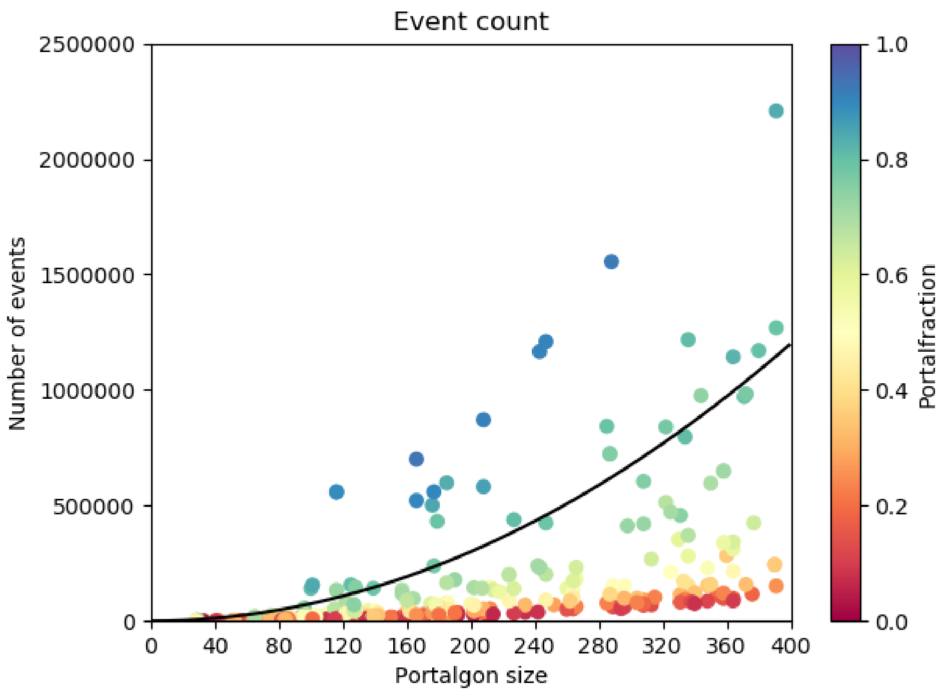


Figure 26: Number of events per portalgon

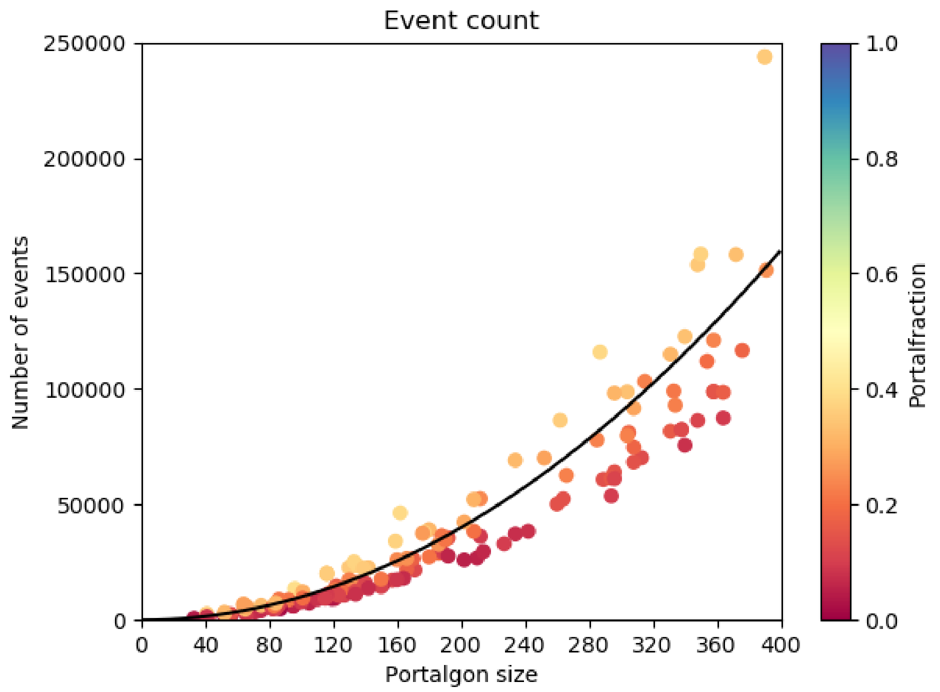


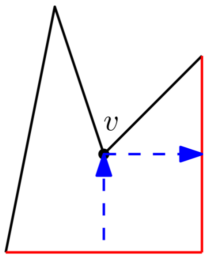
Figure 27: Number of events per portalgon with portalfraction < 40%

7 Interesting phenomena

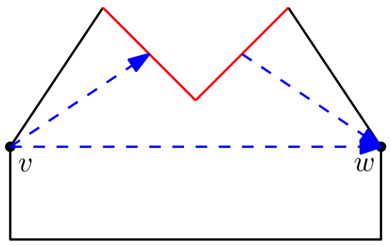
As mentioned in the introduction, portalgons have some interesting properties. The first being the fact that vertices sometimes can see itself, see figure 28a. Which means that the visibility graph may contain loops. Another interesting property is that pairs of vertices may see each other in multiple ways, see figure 28b, which makes the graph a multigraph. Where in normal polygons there exists only one straight line between two points u, v, there can be multiple in a portalgon, making use of portals. These straight lines do however have a limitation:

Lemma 7. All straight lines between two points in a portalgon have a unique sequence of portals that the line goes through.

Proof. Given a pair of vertices u, w and a sequence of portals such that u and w can see each other. If we glue this sequence of portals to each other, then there exist a single straight line between u and w which is the only visibility edge. □



(a) Vertex v can see itself through a portal



(b) Vertices v and w can see each other in multiple ways

Figure 28: Two interesting properties of portalgons

7.1 Infinite visibility edges

In some portalgons the distribution of visibility edges showed a constant number of visibility edges for depths greater than some δ. In most cases this constant number was 4, but sometimes 8 or 2. By examining the visibility graph of these portalgons, we discovered that this happens in portalgons where a pair of portals are parallel and can see each other, see figure 29.

When this happens, for any arbitrary depth a visibility edge exists between two opposing endpoints of the portals. There are two pairs of opposing endpoints, and each visibility edge (u, v) has an opposite edge (v, u), so for each pair of parallel portals results in 4 visibility edges per depth.
This could also be complicated by performing a cut between the two portals and replacing it with a portal, see figure 30. This process can be done multiple times, resulting in visibility edges with a depth of $n(k+1)$, $\forall n \in \mathbb{N}$, where k is the number of portals between the two vertices.

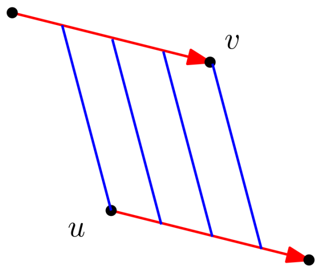


Figure 29: Parallel portals can create unbounded visibility edges

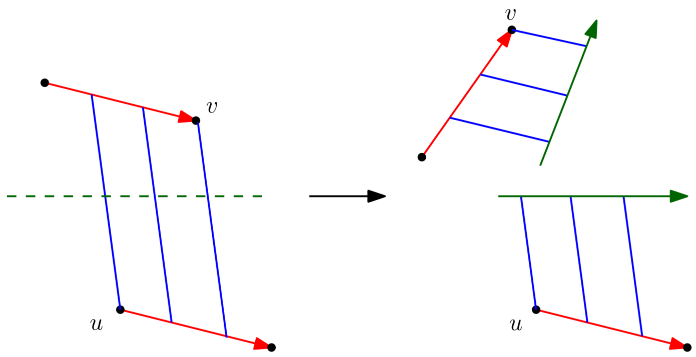


Figure 30: Constructing of a pair of non-parallel portals that create unbounded visibility edges

The zig-zag pattern in length mentioned in 22 is a result of this complication. The portalgon that inhibited this pattern has a pair of parallel portals with a substantial distance between them. It also has a pair of 'parallel' portals with a portal in the middle. The segment length of the visibility edges between these two portals was much smaller than the other pair. This resulted in 4 long visibility edges for every depth, and 4 short visibility edges for every even depth. The average length for the visibility edges is therefore shorter on every even depth.

When the parallel portals only see each other partially as in figure 31, we see something similar. The two non-opposing vertices (u, v) have an infinite number of visibility edges to the vertex in the middle w. This also results in 4 visibility edges per depth. When we add a portal in between as before, the distribution is slightly altered. Instead of just doubling the depth of every visibility edge, a visibility edge coming from u needs 1 more passing of portal to reach w, see figure 32. So visibility edges from u to w have an odd depth. This results in a distribution of 2 visibility edges per depth.

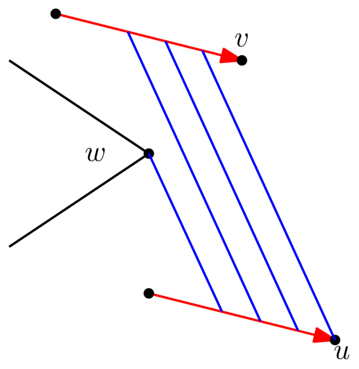


Figure 31: A vertex within the parallelogram spanned by two parallel portals may also leads to unbounded visibility edges

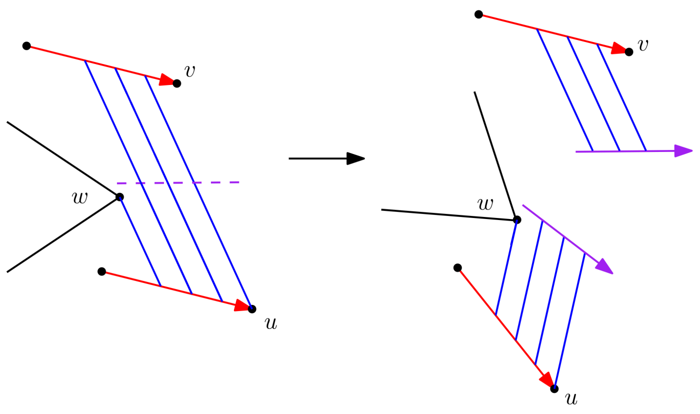


Figure 32: The depths of all visibility edges (u, w) are odd, and all visibility edges (v, w) are even

Proofs about infinite edges

The existence of rays that extend indefinitely complicates further development of algorithms. So it would be useful if we can prove certain properties about these rays. During the theoretical constructing of infinite rays we seem to notice a pattern concerning the starting point. This lead to the following conjecture:

Conjecture. A ray starting from v that can extend indefinitely intersects v infinitely many times.

This conjecture is disproved by the following counterexample:

Counterexample. Take a square with width x. Make the both pairs of opposing edges pairs of portals. Shoot a ray from the bottom left corner $(0, 0)$ in the direction towards (a, x). As the ray keeps extending the ray segments keep shifting a to the right until they reach the right side and continue from the left side. Each ray segment that start from the bottom side now starts at some value $c = k \cdot a \mod x$ for $k \in \mathbb{N}$. If we set a to be an irrational number, and $x \in \mathbb{N}$, c will never become 0 and the ray will therefore never intersect $(0, 0)$, see figure 33.

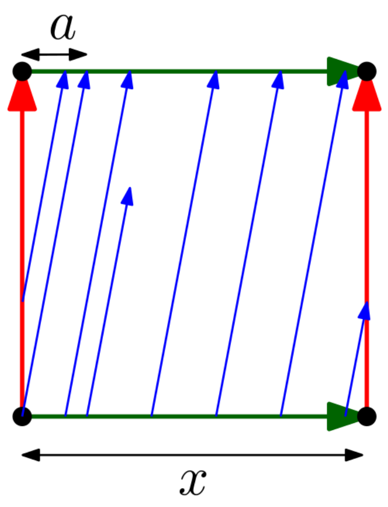


Figure 33: This ray shot from $(0, 0)$ will extend infinitely without hitting $(0, 0)$ ever again

This constructed counterexample does show two things: First the ray will come infinitely close to the starting point, and second the portalgon is fully made up of portals. Maybe these properties are necessary for infinite visibility edges, and we can prove the following conjecture:

Conjecture. A ray starting from v that can extend indefinitely either intersects v infinitely many times or the portalgon only consists of portals.

8 Conclusion

The first conclusion we take from this research on portalgons is a rather dull one. The area of portalgons is a very new one and therefore there are no datasets of portalgons to be used in experimental research. The Salzburg database of generated polygons was very helpful. A more wide variety of polygons would be helpful to examine how portalgons with a high portalfraction would behave in terms of visibility (and other fields), as the only comparison with this database were self-generated polygons with a maximum portalfraction on 45%.

In terms of runtime, the theoretical upper bound of $O(n^{c+1} \log n)$ for running the algorithm is possibly not a tight one. The number of events analysis suggests that the relation is quadratic in the number of vertices, with an extra factor determined by the portalfraction, see figure 26. The upper bound of $O(n^{c+1} \log n)$ assumes that for each level of depth, the number of events is multiplied by a factor n. This increment is not visible, not even in portalgons with a high portalfraction. At most, the number of events increases linear per level of depth, see figure 25.

The tests did lead to some interesting theoretical results about infinite visibility edges. First, parallel portals lead to unbounded visibility edges. The number of segments in these unbounded edges can take on multiple different values, as the complicated cases show, see section 7.1, making detecting these cases difficult.

8.1 Implementation improvements

The current implementation still has some inefficient procedures that increase the computational complexity.

The first being the triangulation structure mentioned in section 4.6. Implementing the triangulation presented in the paper by Hershberger[5] would guarantee a $O(\log n)$ time ray shooting query. furthermore, the implementation recomputes the minimum angle for all extensions each time. Because only the extension with the next vertex needs to be recomputed, this could be improved. This would improve the runtime per query from $O(c \log n)$ to $O(\log n)$.
In hindsight, computing the next vertex around v in the base can also be done using a dual query, instead of looking it up in the sorted list around v. Only the proof of lemma 5 would have to be adapted so it uses the fact that a ray can not rotate 180° within a polygon without hitting a vertex, instead of using the endpoints of the portal the ray enters from. This would not improve the computational complexity, but it would remove the need for sorting the vertices.
The handling of the degenerate case of hitting a vertex x in the dual (Case 2 in section 5.2.3) could be improved. In the implementation all vertices dual to the lines going through x are checked if they are visible (lie between $start(\mathcal{E})$ and $end(\mathcal{E})$, see section 4.5). But all these vertices lie on the same line $\Delta^{-1}(x)$ and therefore some sort of binary search could be applied to quickly find the closest visible vertex.

8.2 Future work

Further (experimental) research on portalgons can benefit from a diverse data set of portalgons, containing portalgons of different portalfractions, portalgons with multiple parallel portals and realistic portalgons. Portalgons currently only exists as theoretical structure with no realistic application, and an application would give the area of research a goal and direction. A diverse set of portalgons would allow to experiment with running algorithm on portalgons that are split into multiple smaller fragments. This could improve the runtime of algorithms, as this runtime usually is dependant on the number of vertices in a fragment, instead of the total number of vertices.

Aside from the practical improvements that can be made to the implementation of the algorithm, there is plenty of research that can be done on visibility algorithms on portalgons. For example, the algorithm by Welz[13] which uses duality to maintain a ray for each vertex, can maybe be adjusted for use in portalgons, by also maintaining extended rays. The problem of infinite and unbounded visibility edge mentioned in section 7.1 also remains unsolved, and could be tackled by some early detecting mechanism.

References

[1] Günther Eder, Martin Held, Steinþór Jasonarson, Philipp Mayer, and Peter Palfrader. Salzburg database of polygonal data: Polygons and their generators. *Data in Brief*, 31:105984, 2020.

[2] Subir Kumar Ghosh and David M Mount. An output-sensitive algorithm for computing visibility graphs. *SIAM Journal on Computing*, 20(5):888–910, 1991.

[3] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[4] Michael Hemmer, Susan Hert, Sylvain Pion, and Stefan Schirra. Number types. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.3 edition, 2021.

[5] John Hershberger and Subhash Suri. A pedestrian approach to ray shooting: Shoot a ray, take a walk. *Journal of Algorithms*, 18(3):403–431, 1995.

[6] Sanjiv Kapoor, SN Maheshwari, and Joseph SB Mitchell. An efficient algorithm for euclidean shortest paths among polygonal obstacles in the plane. *Discrete & Computational Geometry*, 18(4):377–383, 1997.

[7] Der-Tsai Lee. Proximity and reachability in the plane. Technical report, ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB, 1978.

[8] Frank Staals Maarten Löffler, Rodrigo I. Silveira. Shortest paths in portalgons. *37th European Workshop on Computational Geometry (EuroCG)*, 2021.

[9] Joseph SB Mitchell, Günter Rote, and Gerhard Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8(1):431–459, 1992.

[10] Mark H Overmars and Emo Welzl. New methods for computing visibility graphs. In *Proceedings of the fourth annual symposium on Computational geometry*, pages 164–171, 1988.

[11] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 5.3 edition, 2021.

[12] Ron Wein, Eric Berberich, Efi Fogel, Dan Halperin, Michael Hemmer, Oren Salzman, and Baruch Zukerman. 2D arrangements. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.3 edition, 2021.

[13] Emo Welzl. Constructing the visibility graph for n-line segments in o (n2) time. *Information Processing Letters*, 20(4):167–171, 1985.

[14] Mariette Yvinec. 2D triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 5.3 edition, 2021.