Universiteit Utrecht

Local complexity measures for (simple) polygons

Willem Jansen - 6332153
06/2021

Preface

Before you lies my thesis “Local complexity measures for (simple) polygons”. this report is written as part of my graduation project for the master’s program Computing Science at Utrecht University. It is meant for anybody with an interest in geometric algorithms and with a background in computer science.

I started this project in September of 2020 and finished it in June of 2021. It was the final stage of my master education and my final assignment before graduating.

I would also like to use this part of the report to thank my supervisor Frank Staals for his continuous feedback and input on my work. I learned a lot from the discussions we had about the algorithm and our weekly meetings were very helpful.

I hope you enjoy your reading.

Willem Jansen

Bavel, June, 2021

Summary

Various application areas are considered important sources of the need to study visibility, the main ones are: robotics, geographic information systems (GIS) and computer graphics. Therefore, it is important to have efficient algorithms available to solve these problems. Many such algorithms are already presented in literature. Depending on the algorithm, the environment can be modeled in different ways, e.g. a terrain or a polygon. This project studies visibility inside of polygons, specifically simple polygons. A simple polygon is a polygon without holes and self-intersections.

It appears that the performance of these algorithms in practice is influenced by the polygons on which they are used. More specifically, their performance seems influenced by certain properties of a polygon. The project aims to gain insight in which properties can influence visibility computations for a simple polygon P. The first investigated polygon property is the size of the largest/average visibility polygon of P. The second property is the number of reflex vertices of P. The third property is the maximum/average number of vertices of P visible from a chord. An approximation of the chord property is also investigated. Here the maximum/average number of vertices visible from a diagonal is considered. The fourth and final property is a value that indicates how close P is to being a convex polygon.

We sketch an algorithm to compute the chord visibility property of a polygon P. Most but not all of the properties required by the algorithm are fully proved. The algorithm requires that we first pre-process P into a super polygon P_{super} containing the vertices of P and the vertices of each visibility polygon of every vertex of P. Pre-processing a polygon P is done in $O(n^2 \log n)$ time and $O(n^2)$ space. Given a pre-processed polygon we can compute all chords originating from some edge of P_{super} in $O(n^2 \log^2 n)$ time. The chord visibility value of P can be computed in $O(n^4 \log^2 n)$ time using this algorithm.

An implementation of several polygon properties is combined in a computer application to test if there exists some correlation between the parameters and the running times of visibility algorithms in practice. All algorithms are implemented in C++ using the Computational Geometry Algorithms Library (CGAL), an open source library of robust geometric algorithms and data structures. The application computes four different polygon properties. It computes the number of reflex vertices of the polygon. It computes how close the polygon is to being a convex polygon. It computes the size of the largest visibility polygon. It computes the diagonal visibility of the polygon.

The application is used to compute data on a range of polygons. The experiments showed that the time required to compute a visibility polygon is low when the convex hull property of a polygon is high. They also showed that the time required to compute a visibility polygon is low when the visibility polygon size property is high, but only when both values are normalized by the size of the visibility polygon. Finally, the experiments showed that the time required to compute a visibility polygon is low when the diagonal visibility property is high, but again only when both values are normalized by the visibility polygon size.

The convex hull vertices property seems the most significant. This the only property that showed the same relationship with computation time in both the regular and normalized results. The results also suggest that there is a correlation between the convex hull vertices property, the visibility polygon size property and the diagonal visibility property. Therefore the visibility polygon size and the diagonal visibility are also significant properties.

Contents

1 Introduction

There are three application areas that are considered important sources of the need to study visibility: robotics, geographic information systems (GIS) and computer graphics. Each field is discussed briefly below.

Robot motion planning and robotic surveillance raised many visibility challenges which could be transformed into many interesting theoretical problems. The best-known example of visibility in robotics is the use of the visibility graph in exact robot motion planning. The visibility graph is a graph of mutually visible (point) locations, typically for a set of points and obstacles in the Euclidean plane. Each vertex of the graph represents a location. Every edge represents a visible connection between two locations, i.e. the shortest straight uninterrupted path between mutually visible locations. In a simple version of the problem, an environment with obstacles and a source and target position for a (point) robot are given. An algorithm computes the visibility graph of the environment, then finds vertices that correspond to the source and target positions, and finally computes the shortest path between those vertices.

Another important topic in robotics is the exploration of (partially) unknown environments. Obviously, visibility plays an important role here, since we can only explore something when it is visible. The environment becomes known by examining the view of a robot at different positions. This technique yields a model of the environment and is also called model building.

In Geographic Information Science, visibility computations are referred to as viewshed analysis. The viewshed is the area of land that is visible from a given viewpoint. One of the most important fields of visibility analysis is the military. Maintaining visibility of the enemy, or the ability to stay out of view from them is essential for any military. Determining where to build a military base depends largely on the terrain, e.g. does the terrain provide the ability to detect hostiles approaching the base (by providing vantage points). Besides observation issues, (tele)communication which is also strongly related to visibility, is important in the military as well. Many algorithms exist for viewshed computation. Some of them consider the viewshed of a single viewpoint, others compute the overlay of the viewsheds for multiple viewpoints, and there are many more options.

Computer graphics is concerned with the mathematical and computational foundations of image generation and processing. Visibility computations occur mostly in the subfield of rendering. It is easilly recognized in techniques that attempt to speed up the rendering pipeline. Visibility is also strongly related to illumination. So, the techniques that are used in the computations of lighting and shadows, also involve visibility problems. The main objective of visibility computations in graphics is obtaining fast and efficient techniques that provide well-rendered images, i.e., images that look realistic.

Clearly computing visibility is a problem that occurs frequently in various different kinds of applications. Therefore, it is important to have efficient algorithms available to solve these problems. Many such algorithms are already presented in literature. Depending on the algorithm, the environment can be modeled in different ways, e.g. a terrain or a polygon. During this project we will study visibility inside of polygons, specifically simple polygons. A simple polygon is a polygon without holes and self-intersections.

It appears that the performance of these algorithms in practice is influenced by the polygons on which they are used. More specifically, their performance seems influenced by certain properties of a polygon. We intend to gain insight in which properties can influence visibility computations for a simple polygon P. The first polygon property we investigate is the size of the largest/average visibility polygon of P. The second property is the number of reflex vertices of P. The third property is the maximum/average number of vertices of P visible from a chord. We also investigate an approximation for the chord property where we look at the diagonals of P instead. The fourth and final property is a value that indicates how close P is to being convex.

The thesis is organised as follows. Section 2 provides a brief overview of scientific literature regarding the subject. In section 3 we define the geometric properties used in the document. Section 4 formally defines the properties we investigated during the project. In section 5 we discuss the research questions and hypotheses of the project. The visibility algorithms that we compare are described in section 6. We present the chord visibility algorithm in section 7. Details on the implementation are provided in section 8. The experimental setup and test results are discussed in section 9. Finally we give the conclusion in section 10.

2 Related Work

Computing $\mathcal{V}(q)$ for a single query point q, is a well known problem with a number of established algorithms. Asano [1] presented an algorithm that runs in $O(n \log n)$ time and $O(n)$ space. The algorithm follows the rotational plane sweep paradigm. Given some query point q inside the polygon it rotates a sweep-line around q and keeps track of what is and isn't visible in the process.

In [2] an algorithm is presented that computes the visibility polygon of a given point while using few variables. The researchers observed that reflex vertices have a larger influence on a visibility polygon than convex vertices. They developed an algorithm that expresses the running time not just in terms of n but also in terms of the number of reflex vertices r. Their final deterministic and randomized algorithms are parameterized in s, the number of working variables allowed. The deterministic algorithm runs in $O(\frac{nr}{2^s} + n \log^2 r)$ time using $O(s)$ variables. The randomized approach uses $O(\frac{nr}{2^s} + n \log r)$ expected time using $O(s)$ variables.

Guibas et al. [3] developed an algorithm to compute the visibility polygon of a triangulated polygon in linear-time using the shortest path tree. The shortest path tree is the union of all shortest paths from some source vertex s to all vertices v of the polygon. Guibas et al. first give an algorithm capable of computing the shortest path tree in $O(n)$ time for a triangulated polygon. Then they present an algorithm capable of computing the visibility polygon of a query point q in $O(n)$ time using the shortest path tree.

Another linear time algorithm for triangulated polygons is the triangular expansion algorithm [4]. It has a worst case complexity of $O(n)$ for simple polygons, but is $O(n^2)$ for polygons with holes. During a pre-processing phase the polygon is first triangulated. Then given a query point q, the algorithm locates the triangle in which it is contained by a performing a walk. The worst-case running time of the algorithm is actually $O(nh)$, where h is the number of holes of the polygon. Implying a linear running time for simple polygons and a quadratic upper bound for polygons with holes. The algorithm uses $O(n)$ space.

Joe and Simpson [5] also presented a linear time algorithm that runs in $O(n)$ time and space. However, their algorithm does not rely on a triangulation of the polygon like the previous two algorithms. Which in theory makes it faster as it is no longer required to compute a triangulation of the polygon. The algorithm works by performing a sequential scan of the polygon boundary. It uses a stack of boundary points which represents the visibility polygon when the algorithm is finished.

Bose et al. [6] consider an algorithm for answering a number of consecutive visibility queries efficiently. They first decompose the polygon into visibility regions and pre-process the regions for planar point location. A visibility region R is a subset of a polygon P with the property that any two points in R see the same subset of vertices of P. Next, they build the directed dual planar graph of the planar subdivision. The region of any point in P can be found using a standard point location query. When the region is identified, the vertices are enumerated by following a path from the regions node to any sink of the graph. The paper shows that the visibility polygon of the query point can be recovered in $O(\log n + k)$ time using the data structure, where k is the number of visible vertices of P. The algorithm requires $O(n^3 \log n)$ pre-processing time and $O(n^3)$ space.

The research in [7] demonstrates how to decompose a polygon into canonical pieces to keep their visibility information separately, allowing to reduce the storage and pre-processing time. The method makes use of persistent data structures [8] to reduce the space cost of $O(n^3)$ of the previous algorithm. The algorithm constructs a data structure of size $O(n^2)$ which can be computed in $O(n^2 \log n)$ time such that the visibility polygon $\mathcal{V}(q)$ for any point $q \in P$ can be reported in $O(\log^2 n + k)$ time. Where k again denotes the number of vertices of the visibility polygon. Recent research [9] has also shown that it is possible to report a vertex in $O(\log n + k)$ time when using persistent data structures, making the query times as fast as those of the previous algorithm while maintaining lower pre-processing time and space bounds.

The notion of chord visibility was first introduced in [10] to use in an FTP algorithm for the art gallery problem. The paper briefly demonstrates why chord visibility width is a good representation of local complexity in a polygon with respect to the notion of visibility. Although they introduce the parameter and use it to solve the art gallery problem, they do not give an algorithm to compute the parameter. The notion of local complexity and the relation between chord visibility and point visibility is explored in more detail in [11].

In [12] researchers introduce data structures that can be used to solve visibility and intersection problems

in simple polygons. They show that the visibility polygon of some edge of the polygon can be computed in $O(n \log n)$ time. They also show how to perform efficient ray shooting queries inside a polygon. The algorithms rely on a geometric data structure of the polygon in the dual plane. The data structure allows different visibility problems to be computed efficiently.

3 Preliminaries

Let $\mathbb{R}^2$ be the real coordinate space of dimension 2. A simple polygon P is a simply connected subset of the plane. The boundary ∂P of P consists of vertices and edges. We assume polygons are in general position, i.e., no three vertices of P are collinear. Since we deal only with simple polygons, we will refer to them as polygons in the remainder of the document. A polygon P has a set of vertices $v_1, v_2, ..., v_{n-1}, v_n$ such that each consecutive pair is joined by an edge, including the pair $\{v_n, v_1\}$. We assume that the points are in counterclockwise order such that the interior of the polygon lies to the left when the boundary of the polygon is traversed.

We say a vector $\overrightarrow{xy}$ is the vector originating at a point x and ending at y. Two points p and q can see each other if the line segment $\overline{pq}$ lies inside the polygon. When both p and q are boundary points, $\overline{pq}$ is called a *chord*. A chord is called a *diagonal* if both its endpoints are vertices. A chord s separates a simple polygon into two connected components. The *visibility polygon* $\mathcal{V}(q)$ of a point $q \in P$ is the set of points in P visible from q. A visibility polygon is a star-shaped polygon, i.e. a polygon for which there is a point z such that for each $p \in P$ the segment $\overline{zp}$ lies inside P. The boundary of a visibility polygon consists of parts of the polygon boundary and chords, called *windows*. Any point invisible to q is separated from $\mathcal{V}(q)$ by a window.

A visibility polygon can be represented geometrically by a circular list of vertices with their planar coordinates. With prior knowledge of P, we can also represent visibility polygons combinatorially using a circular list of vertices and edges of P in the order in which they appear on the boundary of $\mathcal{V}(q)$. This list is called the *combinatorial representation* of the visibility polygon $\mathcal{V}(q)$. The coordinates of each vertex of $\mathcal{V}(q)$ can be computed given the combinatorial representation.

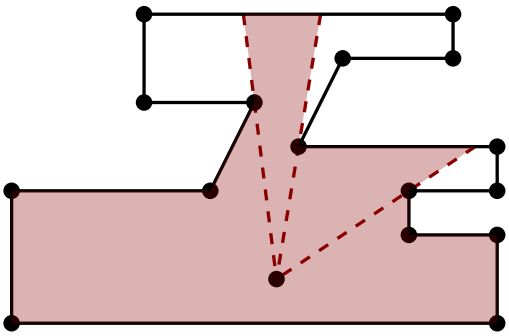
A property of simple polygons is that once the vertices and edges visible from a point are known, the order in which they appear on $\mathcal{V}(q)$ is uniquely determined, as it coincides with the order along the boundary of the original polygon. We say two visibility polygons are (combinatorially) *equivalent* if their combinatorial representations are identical (up to a circular shift), i.e. vertex and edge neighbours are identical but the order in which they appear may differ.

Given a line segment $S \subseteq P$ we define the *weak visibility polygon* $\mathcal{V}(S)$ of S to be the union of all visibility polygons induced by S, i.e. $\mathcal{V}(S) = \cup_{p \in S} \mathcal{V}(p)$. A point q can see a line segment S inside P if q sees at least one point on S, intuitively $\mathcal{V}(S)$ is the set of points in P which can see S. A weak visibility polygon can be represented geometrically by a circular list of vertices with their planar coordinates. Observe that each vertex of $\mathcal{V}(S)$ is either:

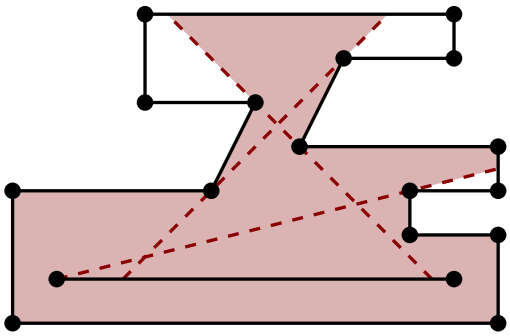
1. A vertex of P visible from S;
2. A point cast on the boundary of P by a ray that emanates from an endpoint e_1 or e_2 of S and passes through a vertex of P visible from that endpoint;
3. A point cast by a ray that emanates from some interior point on S and passes through two vertices v and w of P, such that the exterior of P lies on one side of the ray in the vicinity of v and on the other side of the ray in the vicinity of w.

With prior knowledge of P, we can also represent weak visibility polygons combinatorially. The combinatorial representation of $\mathcal{V}(S)$ consists of a circular list of pairs and edges. Each pair represents a vertex, and contains the vertex it represents as well as a (possibly empty) pointer to another vertex of the polygon. For vertices of type one, the pointer is empty. For vertices of type two, the pointer points to endpoint e_1 or e_2 of S. For vertices of type three, the pointer points to vertex v or w of P. The coordinates of each vertex of $\mathcal{V}(S)$ can be computed given the combinatorial representation. Two weak visibility polygons are considered

(combinatorially) *equivalent* if their combinatorial representations are identical (up to a circular shift), i.e. pair and edge neighbours are identical but the order in which they appear may differ.



(a) Visibility polygon of a point, $\mathcal{V}(q)$ is represented by the red area.



(b) Visibility polygon of a line segment, $\mathcal{V}(S)$ is represented by the red area.

Figure 1: Visibility inside a polygon, as seen from a point and a line segment

A vertex v_k is a reflex vertex when the angle at v_k induced by the edges (v_{k-1}, v_k) and (v_k, v_{k+1}) is greater than π on the inside of P. Given a vertex v_k and a point q, the line $l_k(q)$ passing through them splits P into disjoint components. A vertex v_k is reflex with respect to q if it is a reflex vertex, and the vertices v_{k-1} and v_{k+1} lie on the same side of $l_k(q)$. Observe that any vertex that is reflex with respect to q is a reflex vertex (in the usual sense), but the opposite is not true. Intuitively speaking, reflex vertices with respect to q are the vertices where important changes occur in the visibility polygon. That is, where the polygon boundary can change between visible and not visible.

A line l is tangent to a polygon P at vertex v if l passes through v, and l is inside P in an open neighborhood of v. Notice that a line can be tangent to P only at a reflex vertex. For a point $p \in P$ and a vertex v, consider the ray emanating from p, aimed at v, and crossing ∂P at a point w after v ($w \neq v$). If $\overline{pw}$ is inside P and the line containing $\overline{pw}$ is tangent to P at v, then the chord $\overline{vw}$ is called the constraint induced by p and v. Observe that for any point $p \in P$, the constraints induced by p are exactly the windows of $\mathcal{V}(p)$.

Every window of a point $q \in P$ is collinear with q. Given a window $\overline{vw}$ with v closest to q, we call v the *base* and w the *end* of the window. Note that the base of a window is always a vertex of P, but the end may not be. A window is denoted by $w(base, end)$. We refine the definition of windows slightly to eliminate ambiguities that exist due to collinearities. If there is a situation similar to that of (a) and (b) of Figure 2, where q is collinear with two reflex vertices of the polygon that lie to one side of q, we say that $w(b, e)$ is the window of point q. If a situation similar to that in (c) arises, we say that there are two windows, $w(b, e)$ and $w(b_1, e)$. Finally, if a situation similar to that in (d) arises, we say that there are two windows, one is $w(b, e)$ and the other is $w(b_1, e_1)$. A pocket of q in a polygon P is is associated with a window of q and defined to be a maximal connected subset of $P \setminus \mathcal{V}(q) \cup w(b, e)$. Each pocket is connected to the visibility polygon by a window.

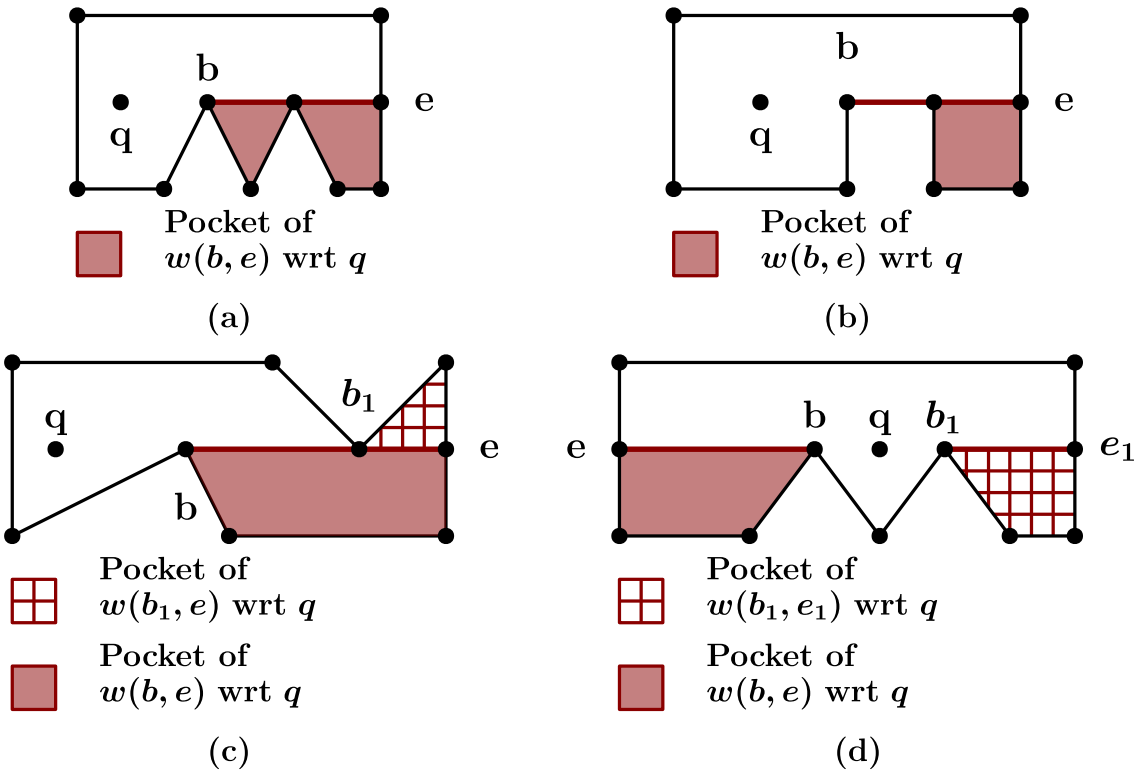


Figure 2: Redefined definition of windows.

4 Polygon Properties

The polygon properties introduced in the introduction are formally defined below. Each property, except polygon convexity, will be computed as a value and as a ratio with regard to the number of vertices of the polygon. The definition of these properties is given as a value rather then as a ratio. To compute the ratio of those properties with regard to a polygon, simply divide the values by the number of vertices n.

The polygon convexity property is only computed as a ratio and not as a value. Therefore, its definition is also given as a ratio while the other definitions are not.

4.1 Visibility Polygon Size

For a polygon P and a point $q \in P$, the size of visibility polygon $\mathcal{V}(q)$, denoted $|\mathcal{V}(q)|$, is equal to the number of vertices in the geometric representation of $\mathcal{V}(q)$. A visibility region R in a polygon P is a maximally connected subset of P with the property that any two points in R see the same subset of vertices and edges of P [6]; that is for two points p and q from R the visibility polygons $\mathcal{V}(p)$ and $\mathcal{V}(q)$ are (combinatorially) equal. A polygon P contains $O(n^3)$ visibility regions which is tight in the worst case [6]. Let S be a set of points inducing the combinatorially distinct visibility polygons of P, that is S contains a single point r from each visibility region $R \subseteq P$.The maximum visibility polygon size of P is the size of the largest distinct visibility polygon induced by S, i.e $|\mathcal{V}_{max}(P)| = \max_{r \in S} |\mathcal{V}(r)|$. The average visibility polygon size of P is the average size of the distinct visibility polygons induced by S, i.e. $|\mathcal{V}_{avg}(P)| = \frac{1}{|S|} \sum_{r \in S} |\mathcal{V}(r)|$.

4.2 Reflex Vertices

Let R be the set of reflex vertices of P. The edges incident to a vertex $r \in R$ must have an angle greater then π on the inside of P. The number of reflex vertices of P is equal to the size of R, denoted $|R|$.

4.3 Chord Visibility

If two points p and q on ∂P are visible to each other then $\overline{pq}$ is a chord of P. Let $S \subseteq \overline{pq}$ be the set of points inducing the (combinatorially) distinct visibility polygons seen from the chord. Each point $r \in S$ is the first point on the chord, when walking from p to q, at which a distinct visibility polygon is visible. Observe that the weak visibility polygon of $\overline{pq}$ is equal to the union of all polygons induced by S, that is $\mathcal{V}(\overline{pq}) = \cup_{r \in \overline{pq}} \mathcal{V}(r) = \cup_{r \in S} \mathcal{V}(r)$. Let $V_{poly}(\overline{pq})$ be the set of polygon vertices seen from $\overline{pq}$, with $V_{poly}(\overline{pq}) \subseteq \mathcal{V}(\overline{pq})$. Observe that $V_{poly}(\overline{pq}) = \cup_{r \in S} V_{poly}(r)$, where $V_{poly}(r)$ is the set of polygon vertices seen from r with $V_{poly}(r) \subseteq \mathcal{V}(r)$. Furthermore, notice that $V_{poly}(r)$ is the visibility set of the visibility region [6] containing r. The chord visibility of $\overline{pq}$ is defined as the number of polygon vertices visible from $\overline{pq}$, that is the number of vertices of P present in $\mathcal{V}(\overline{pq})$. Formally we say $cv(\overline{pq}) = |V_{poly}(\overline{pq})|$, where $|V_{poly}(\overline{pq})|$ denotes the size of $V_{poly}(\overline{pq})$.

Two visibility chords are considered equal when they induce the same set of distinct visibility polygons, i.e. when their weak visibility polygons are (combinatorially) equal. Let U be the set of distinct visibility chords of P, i.e. no two chords in U see the same set of visibility polygons of P. The chord visibility of P is defined as the largest number of vertices that can be seen from a visibility chord $\overline{pq} \in U$, that is $cv(P) = \max_{\overline{pq} \in U} cv(\overline{pq})$. We extend the definition of chord visibility to also include average chord visibility. The average chord visibility of P is the average number of vertices that can be seen from a visibility chord $\overline{pq} \in U$, i.e. $cv_{avg}(P) = \frac{1}{|U|} \sum_{\overline{pq} \in U} cv(\overline{pq})$.

4.3.1 Diagonal Visibility

Instead of computing chord visibility, using the distinct chords of P, we can also compute it using the diagonals of P. We call this parameter the diagonal visibility of P. The diagonal visibility of a polygon is an approximation of its chord visibility value. The diagonals of P can be computed in two different ways. The first option is to take all $O(n^2)$ diagonals between all vertices of a polygon. The second option is to compute a triangulation of the polygon and use the triangle edges as diagonals. Both options will be looked in to during the second phase of the project. Since the second option depends on which triangulation is used, we will also look into different kinds of triangulation methods.

Given two vertices v and w of P, the diagonal visibility value $dv(\overline{vw})$ of diagonal $\overline{vw}$ is defined exactly like the chord visibility, so $dv(\overline{vw}) = |V_{poly}(\overline{vw})|$. Let D be the set of distinct diagonals of P. The diagonal visibility of P is defined as the largest number of vertices that can be seen from a diagonal $\overline{vw} \in D$, that is $dv(P) = \max_{\overline{vw} \in D} dv(\overline{vw})$. The average diagonal visibility of P is the average number of vertices that can be seen from a diagonal $\overline{vw} \in D$, i.e. $dv_{avg}(P) = \frac{1}{|D|} \sum_{\overline{vw} \in D} cv(\overline{vw})$.

4.4 Polygon Convexity

We introduce two polygon convexity parameters that both have a unique way of determining the convexity of a polygon. Our first convexity parameter is the percentage of vertices of P that lie on its convex hull. Our second parameter is the relative size of a largest convex sub-polygon of P compared to P.

The convex hull $\mathcal{CH}(P)$ of a polygon P is the smallest convex set that contains P, if P is a convex polygon then $\mathcal{CH}(P)$ is equal to P. Observe that when P is convex $\mathcal{V}(P) = \mathcal{CH}(P) = P$. The hull vertices parameter $hv(P)$ of a polygon P, with $0 < hv(P) \leq 1$, represents how many vertices of P lie on $\mathcal{CH}(P)$. When P is a convex polygon, $hv(P) = 1$. The less vertices of P lie on $\mathcal{CH}(P)$, the closer $hv(P)$ will be to 0. Let P be a polygon with n vertices and let h vertices of P lie on the convex hull, then $hv(P) = \frac{h}{n}$.

We denote the area of a polygon P by $||P||$. Let $C \subseteq P$ be a convex polygon contained by P with the property that $||C||$ is maximal. C consists of vertices of P and points on edges of P, if P is a convex polygon then C is equal to P. Observe that when P is convex $\mathcal{V}(P) = P = C$. The convex area parameter $ca(P)$ of P, with $0 < ca(P) \leq 1$ represents how large the area of C is compared to the area of P. When P is a convex polygon, $ca(P) = 1$, otherwise $ca(P) < 1$. Let P be a polygon and let C be the largest convex polygon contained by P, then $ca(P) = \frac{||C||}{||P||}$.

5 Research Questions

The goal of the research is to provide a more detailed analysis of visibility algorithms, and to express their worst-case time and space bounds not just in the input size n, but also in parameters measuring properties of the polygon. To this end we want to establish how our polygon properties relate to the complexity measures of a number of visibility algorithms. Ideally, we want to establish whether or not there is a correlation between those bounds and our properties. At the moment there are no results which indicate that such correlations indeed exist. In this study we intend to learn if such correlations exist or not.

To this end we will investigate how the properties influence the performance results of visibility algorithms. This provides a good indication of whether it is worth investigating how the parameters correlate to the complexity measures. To measure performance results we will look at two values. The running time of the algorithms in milliseconds, as this is related to the worst case running time. As well as the size of the computed visibility polygons, where the size is equal to $|\mathcal{V}(q)|$ (the number of vertices in the geometric representation of the visibility polygon), as this is related to the size bounds on these polygons.

We will look at three different algorithms, all of which run in linear time on simple polygons. The first algorithm will be the triangular expansion algorithm [4]. The second algorithm is the algorithm by Joe and Simpson [5]. The third and final algorithm algorithm is that of Asano [1].

To successfully complete the goals of our project, we defined the following research question. This is the general research question of the project.

How do properties of a simple polygon influence the complexity of visibility computations?

We also defined specific research questions for each property and complexity measure. For every pair of properties and measures we ask the following question: How does property **X** of a simple polygon influence complexity measure **Y**? Each property and their relevant measures are denoted in table 1. The X's denote which measures are investigated for which properties.

Property \ Complexity measure	Running time of visibility computations	Size of visibility polygons
Average/Maximum visibility polygon size	X	
Number of reflex vertices	X	X
(Average) Chord visibility	X	X
polygon convexity	X	X

Table 1: Polygon properties and their relevant complexity measures.

5.1 Hypotheses

For every research question we have some intuition on how its property correlates to its complexity measure. Based on this intuition we give a hypothesis for each research question. We will confirm or disprove all hypotheses based on the results of the research.

When the largest visibility polygon of a polygon P is small, a logical consequence would be that computing any visibility polygon of P is fast. However, when the average visibility polygon size is low this is not automatically the case. P can contain many small visibility polygons but also several large ones, in which case the average size is still low. Based on these observations we give the following hypotheses for the visibility polygon size parameter.

- The running time of visibility computations will be low when the maximum visibility polygon is small.
- The running time of visibility computations is likely to be low when the average visibility polygon is small.

When the number of reflex vertices of a polygon P is small, we can likely see a large subset of P from any query point q. This means $\mathcal{V}(q)$ would contain few windows and many parts of ∂P. Computing visibility in

such a polygon might be faster, since ∂P is already known and only a small number of windows are computed. Another consequence of few reflex vertices is that the size of the visibility polygon will likely be large, as q can see many vertices and edges of P. This also implies that visibility polygons are small when P has many reflex vertices. Based on these observations we give the following hypotheses for the reflex vertex parameter

- The running time of visibility computations will be low when there are few reflex vertices and high when there are many reflex vertices.
- Visibility polygons will be small when there are many reflex vertices and large when there are few reflex vertices.

When the chord visibility value of a polygon P is low, query points can only see a small part of P. This implies that we only have to consider a small part of P when computing $\mathcal{V}(q)$ for a query point q. This in turn suggests that computing $\mathcal{V}(q)$ might be faster when the chord visibility value of P is low. However, when the average chord visibility value is low, this is not automatically the case. P can contain many chords with low values and several chords of high value, in which case the average value is still low. We believe the size of visibility polygons in P is small when the chord visibility value is low, as a query point q can see few vertices and edges of P. This also implies that visibility polygons are large when P has a high chord visibility value. Based on these observations we give the following hypotheses for the chord visibility parameter.

- The running time of visibility computations will be low when the chord visibility value is low and high when the chord visibility value is high.
- The running time of visibility computations will likely be low when the average chord visibility value is low and it will high when the average chord visibility value is high.
- Visibility polygons will be small when the chord visibility value is low and large when the chord visibility value is high.

Computing visibility is easy when a polygon P is convex, as $\mathcal{V}(q) = P$ for any query point q. So it also makes sense that computing visibility is likely easy when P is an almost convex polygon. $\mathcal{V}(q)$ consists almost entirely of parts of ∂P when P is almost convex. This implies that computing visibility in an almost convex polygon is likely faster, as ∂P is already known and only a small number of windows are computed. It also implies that the visibility polygon is large when P is almost convex, since a query point q can see many vertices and edges of P. Based on these observations we give the following hypotheses for the polygon convexity parameter.

- The running time of visibility computations will be low when the original polygon has an almost convex shape.
- Visibility polygons will be large when the original polygon has an almost convex shape and small otherwise.

5.2 Diagonal Visibility

The diagonal visibility value is a somewhat different property as intuitively, it is an approximation of the chord visibility value of a polygon. However, currently there is no formal prove that shows this is indeed the case. As such its research questions and hypotheses are also a little different. It shares the questions on computation time and visibility polygon size with the other properties. However, there is also an additional question we want to investigate, regarding the relation between chord visibility and diagonal visibility. The following are the research questions for diagonal visibility.

- How does the diagonal visibility property of a simple polygon influence the running time of visibility computations?
- How does the diagonal visibility property of a simple polygon influence the size of visibility polygons?
- What is the relation between diagonal visibility and chord visibility?

We expect the relation between diagonal visibility, and the running time of visibility computations and the size of visibility polygons to behave like that of chord visibility. Furthermore we believe that when the diagonal visibility value of a polygon is low, the chord visibility is also likely to be low. Observe that the opposite may not be true. Diagonal visibility seems like a good approximation of chord visibility since they are closely related. Based on these observations we give the following hypotheses for the diagonal visibility parameter.

- The running time of visibility computations will be low when the diagonal visibility is low and high when the diagonal visibility is high.
- Visibility polygons will be small when the diagonal visibility is low and large when the diagonal visibility is high.
- When diagonal visibility is low, chord visibility is also low.
- Diagonal visibility is a good approximation for chord visibility.
- Diagonal visibility and chord visibility have the same upper bound on visible vertices.

5.3 Edge Cases

It is possible to construct edge cases for each polygon property such that our hypotheses wont hold. These edge cases occur in very specific polygons and are not representative of the behaviour of the properties in the majority of polygons. As such we will not take the edge cases into account when confirming or disproving the hypotheses.

An example of a specific edge case is illustrated in figure 3, this polygon is an edge case for the reflex vertices property. The polygon P depicted in figure 3 contains 15 vertices in total, 6 of which are reflex vertices. Almost half of the vertices of P are reflex vertices which is quite a lot. Observe that this type of polygon can be constructed for any odd number of vertices. The hypotheses suggest that computing visibility in P should be slow since there are many reflex verities. However, when we take a query point q as illustrated in the image, we only need to compute two windows which should not take much time. So, computing visibility is actually not slow in this specific example.

A polygon like P is constructed specifically as an edge case and it is probably not representative for polygons in practice. Which is why we label this and similar cases as edge cases.

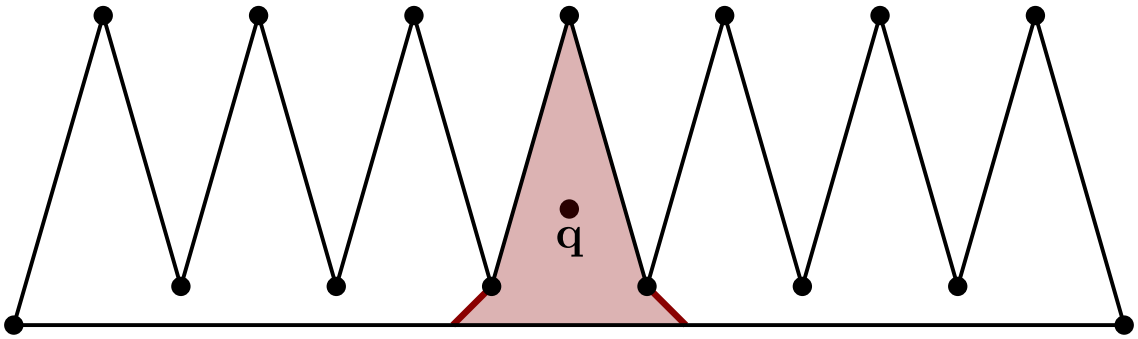


Figure 3: An edge case where the reflex vertices property doesn't behave as expected, $\mathcal{V}(q)$ is represented by the red area.

6 Visibility Algorithms

We compare three visibility algorithms in practice. The rotational sweep algorithm [1], the triangular expansion algorithm [4] and the sequential algorithm [5]. The algorithms are chosen because each one is implemented efficiently in CGAL. The algorithms are explained briefly below.

6.1 Rotational Sweep Algorithm

The first algorithm we use in the comparison is the rotational sweep algorithm [1]. The algorithm follows the rotational plane sweep paradigm. It defines a ray called the event line, denoted L, that originates from and rotates around q. Events take place when L intersects a new segment of ∂P and when L stops intersecting a segment of ∂P. Events are stored in an event list Q. The vertices of P are the event points of the algorithm, as this is where L starts and stops intersecting segments of ∂P. The status is defined as the segments that intersect L at the current position, stored in a status structure T. Both Q and T are stored in a balanced binary search tree.

First all vertices of the input polygon P are sorted and stored in Q according to their polar angles with respect to the query point q. This can be done using standard sorting methods in $O(n \log n)$ time. Then L will rotate around q, and the segments of ∂P intersecting L are stored in T based on their order of intersections with L. As the sweep proceeds, T is updated each time L starts or stops intersecting a segment of ∂P. A new vertex of $V(q)$ is generated each time the smallest element (segment closest to q) in T changes, i.e. when L intersects a segment closer to q then the current closest segment or when L stops intersecting the current closest segment. In the following, we present the major steps for computing $\mathcal{V}(q)$.

Algorithm 1 Compute the visibility polygon of q

Input. A polygon P and query point $q \in P$.
Output. The visibility polygon $\mathcal{V}(q)$ of q.

1: let L be the horizontal line originating from q and directed towards the right of q
2: compute the intersection points of L with the edges of P
3: sort the intersection points from left to right along L
4: initialize the binary search tree T with the points intersecting L.
5: sort the vertices of P according to their polar angle and store them in the event list Q
6: label the vertices as $v_1, v_2, ..., v_n$ and initialize the index i with 1.
7:
8: **if** both edges incident to v_i are in L **then**
9: Remove the edges incident to v_i from T
10: **if** there is a change in the leftmost leaf of T **then**
11: compute the intersection point z of $\overrightarrow{qv_i}$ and the edge currently in the leftmost leaf of T
12: add $v_i z$ to $\mathcal{V}(q)$
13: **end if**
14: go to step 30
15: **end if**
16:
17: **if** one edge incident to v_i is in L **then**
18: replace the edge of v_i currently in L with the edge of v_i not in L
19: go to step 30
20: **end if**
21:
22: **if** both edges incident to v_i are not in L **then**
23: Insert both edges of v_i into T
24: **if** there is a change in the leftmost leaf of T **then**
25: compute the intersection point z of $\overrightarrow{qv_i}$ and the edge currently in the leftmost leaf of T
26: add $v_i z$ to $\mathcal{V}(q)$
27: **end if**
28: **end if**
29:
30: **if** $i \neq n$ **then**
31: $i := i + 1$
32: go to step 8
33: **end if**
34:
35: **return** $\mathcal{V}(q)$

A sweep of L handles every event (vertex) once so it takes $O(n)$ time and when the sweep is finished $\mathcal{V}(q)$ is known. The running time of the algorithm is bound by the initial sorting step and is $O(n \log n)$. The event queue initially stores all vertices and gets shorter as the sweep progresses so it is never more then linear in size. The status structure stores a segment at most once, thus using $O(n)$ storage. The total amount of storage used by the algorithm is therefore $O(n)$.

6.2 Triangular Expansion Algorithm

The second algorithm we compare is the triangular expansion algorithm [4] which runs in linear time on a triangulated polygon. The algorithm has a worst case complexity of $O(n)$ for simple polygons, but is $O(n^2)$ for polygons with holes.

During a pre-processing phase the polygon is first triangulated. This can be done using any triangulation, a Delaunay triangulation is used in [4]. There are a number of algorithms available that can efficiently compute a Delaunay triangulation for a polygon. Given a query point q, the algorithm locates the triangle containing it by a performing a walk. The point q sees the entire triangle and it reports all edges that belong to ∂P. For every other edge, the algorithm starts a recursive procedure that expands the view of q through that edge into the next triangle. At first the view is restricted by the endpoints of the edge, but as the recursion proceeds the view can be restricted further. Each recursive procedure may split into two new procedures when a triangle can be expanded through both unexplored edges. This happens when neither unexplored edge of a triangle is part of ∂P. A polygon contains n vertices so this can happen $O(n)$ times; each call can reach $O(n)$ triangles, suggesting a worst case running time for the algorithm of $O(n^2)$.

Fortunately, a split into two new recursive procedures that reach the same triangle independently can only happen at holes of P. So, the worst-case running time of the algorithm is actually $O(nh)$, where h is the number of holes of the polygon. This implies a linear running time for simple polygons, which are the polygons that are of interest to us. Although the paper does not provide a complete proof of the linear running time for simple polygons, it does give a good intuition on why this is the case. We do believe that the algorithm runs in linear time for simple polygons. If a polygon has holes, then it is possible for $h = \Theta(n)$ giving a worst case complexity of $\Theta(n^2)$ for those polygons. The algorithm uses $O(n)$ space as the triangulation has linear size and there are at most $O(n)$ recursive calls on the stack at any time.

6.3 Sequential Algorithm

The third algorithm of our comparison is that of Joe and Simpson [5] which runs in $O(n)$ time and space.However, their algorithm does not rely on a triangulation of the polygon like the previous two algorithms. Which in theory makes it faster as we no longer have to compute a triangulation of our polygons.

The algorithm works by performing a sequential scan of the boundary of a polygon P. It uses a stack S of boundary points $s_0, s_1, ..., s_t$ such that when the algorithm is finished, the stack represents the visibility polygon. However, during the execution of the algorithm, S is not guaranteed to represent part of the final visibility polygon. When the current edge is processed, three possible operations can be performed: new boundary points can be added to the stack, obsolete boundary points can be deleted from the stack or a sequence of edges on ∂P is scanned until a certain condition is met. The algorithm uses states to process the ∂P and manipulate the stack.

In the following, we present the major steps for computing $V(q)$. Assume v_0 is the closest point of ∂P to to the right of q and that vertex v_1 is the next counter clockwise vertex of v_0. Push v_0 on S and initialize i by 1.

Algorithm 2 Compute the visibility polygon of q

Input. A polygon P and query point $q \in P$.
Output. The visibility polygon $\mathcal{V}(q)$ of q.

1: Push v_i onto S and $i := i + 1$
2: **if** $n = i + 1$ **then** go to step 51 **end if**
3: **if** v_i lies to the left of $\overrightarrow{qv_{i-1}}$ **then** go to step 1 **end if**
4:
5: **if** v_i lies to the right of both $\overrightarrow{qv_{i-1}}$ and $\overrightarrow{v_{i-2}v_{i-1}}$ **then**
6: scan from v_{i+1} in counterclockwise order until a vertex v_k is found such that $v_{k-1}v_k$ intersects $\overrightarrow{qv_{i-1}}$ in a point z
7: push z onto the stack
8: $i = k$
9: go to step 1
10: **end if**
11:
12: **if** v_i lies to the right of $\overrightarrow{qv_{i-1}}$ and to the left of $\overrightarrow{v_{i-2}v_{i-1}}$ **then**
13: let u denote the element on top of S and pop it from the stack
14: **while** u is a vertex and $v_{i-1}v_i$ intersects uq **do**
15: pop u from the stack
16: **end while**
17: **end if**
18:
19: **if** $v_{i-1}v_1$ does not intersect uq **then**
20: **if** v_{i+1} lies to the right of $\overrightarrow{qv_i}$ **then**
21: $i := i + 1$
22: go to step 19
23: **end if**
24: Let m be the point of intersection of $\overrightarrow{qv_i}$ and the edge containing u
25: **if** v_{i+1} lies to the right of $\overrightarrow{v_{i-1}v_i}$ **then**
26: push m on S
27: go to step 1
28: **end if**
29: scan from v_{i+1} in counterclockwise order until a vertex v_k is found such that $v_{k-1}v_k$ intersects mv_i
30: assign k to i
31: go to step 19
32: **end if**
33:
34: let w be the vertex immediately below u on the stack
35: let p be the point of intersection between $v_{i-1}v_i$ and uq
36: **if** $p \in qw$ or q, w and u are not collinear **then**
37: pop u from the stack
38: go to step 19
39: **end if**
40:
41: scan from v_{i+1} in counterclockwise order until a vertex v_k is found such that $v_{k-1}v_k$ intersects wp
42: push the intersection point onto S
43: assign k to i
44: go to step 1
45:
46: **return** $\mathcal{V}(q)$ by popping all vertices and points from S

Every vertex is visited once during the scan, so the algorithm takes $O(n)$ time. The stack contains at most $O(n)$ vertices at any point during the execution, so the algorithm uses linear space.

7 Chord Visibility Algorithm

In this section we present an algorithm to compute the chord visibility value of a simple polygon P. We show that the algorithm is correct and analyze the running time.

7.1 Correctness

We start by proving properties of chords and windows that are required for the chord visibility algorithm. We begin by studying the relation between vertices chords.

Lemma 1. Let r and r' be two points on a chord $\overline{pq}$, r and r' see different sets of vertices if and only if r and r' are on different sides of a window intersecting $\overline{pq}$.

Proof. We use a proof by contradiction. Suppose r and r' are not separated by a window. Now suppose that r can see a vertex v but r' cannot. We know that r lies in $\mathcal{V}(v)$ since it can see v. We know that r' lies in the in the pocket Q, of some window w collinear with v, as it cannot see v. Now, let $\overline{rr'}$ be the line segment formed by r and r'. The segment intersects $\partial\mathcal{V}(v)$ since $r \in \mathcal{V}(v)$ and $r' \in Q$. The boundary of $\mathcal{V}(v)$ contains sections of ∂P and windows collinear with v. Both r and r' lie in P, so the edge of $\partial\mathcal{V}(v)$ intersected by $\overline{rr'}$ must be a window. Specifically, the window intersecting $\overline{rr'}$ must be w since $r' \in Q$. We have therefore shown that r and r' are separated by a window and have reached a contradiction. □

Lemma 2. A vertex v is invisible from a chord $\overline{pq}$ if and only if there is a window w collinear with v that separates v from $\overline{pq}$.

Proof. A chord $\overline{pq}$ is by definition only visible from v when v can see at least one point on $\overline{pq}$. If v sees some subset $\overline{p'q'} \subseteq \overline{pq}$ then $\overline{p'q'}$ must lie inside $\mathcal{V}(v)$. If v cannot see $\overline{pq}$ then the entire chord must lie in the pocket Q of some window w collinear with v. When $\overline{pq}$ lies in Q, it is by definition separated from $\mathcal{V}(v)$ by w. Therefore, v is only invisible from $\overline{pq}$ when they are separated by a window w collinear with v. □

Let P_{super} of P be a polygon containing the vertices of P as well as the vertices of each $\mathcal{V}(v) \subseteq P$. Let $V(P)$ be the set of vertices of P. Each vertex w with $w \in V(P_{super}) \setminus V(P)$ is the endpoint of a window of some visibility polygon for a vertex $v \in P$. P_{super} consists only of vertices of P and endpoints of windows. If P_{super} is known then all windows of P and the vertices associated with them can be derived.

Lemma 3. Let e, e' and e'' be edges of P_{super} with e' and e'' adjacent. Let $\overline{c}$ be a chord between e and e', and $\overline{c'}$ be a chord between e and e''. The chord visibility of $\overline{c}$ and $\overline{c'}$ is the same if and only if the vertex u incident to e' and e'' is not the endpoint of a window.

Proof. We distinguish between two possible cases. One where we have two chords that have a common endpoint and another where we have two distinct chords. Each case is proved using a proof by contradiction.

First, let $\overline{c} = \overline{pq}$ be a chord between e and e', and $\overline{c'} = \overline{ps}$ be a chord between e and e''. Note that $\overline{pq}$ and $\overline{ps}$ have a common endpoint p. Now suppose that $\overline{pq}$ can see a vertex v and that $\overline{ps}$ cannot. We know from lemma 2 that $\overline{ps}$ is separated from v by a window w. We define w as a chord $\overline{rz}$ where r is the reflex vertex where the window originates and where z is the endpoint of the window. We also know v is by definition only visible from $\overline{pq}$ when v can see at least one point on $\overline{pq}$ and that some subset of $\overline{pq}$ must therefore lie in $\mathcal{V}(v)$. It follows from the definitions of $\overline{pq}$ and $\overline{ps}$ that $\overline{pq}$ cannot lie completely in $\mathcal{V}(v)$ since at least p is separated from v by w. Therefore, $\overline{pq}$ must intersect w. The endpoints q and s are separated by w since s is in the pocket of w and q is not. From which follows that either z or r lies on the section of ∂P_{super} between q and s not containing e. If either point lies between q and s on this section of ∂P_{super} then we have reached a contradiction as $\overline{pq}$ and $\overline{ps}$ can only be separated by u.

Second, let $\overline{c} = \overline{pq}$ be a chord between e and e', and $\overline{c'} = \overline{ps}$ be a chord between e and e''. Now suppose that $\overline{pq}$ can see a vertex v and that $\overline{st}$ cannot. We know again, from lemma 2, that $\overline{st}$ is separated from v by a window w, i.e. $\overline{st}$ is in the pocket Q of w. We define w as before, i.e. a chord with reflex vertex r and endpoint z. We also know, again by definition, that some subset of $\overline{pq}$ must lie in $\mathcal{V}(v)$. We know that e must be in Q since $s \in Q$, $s \in e$ and $e \in \partial P_{super}$. From which follows that $p \in Q$ because $p \in e$. Therefore, $\overline{pq}$ must intersect w. If $\overline{pq}$ intersects w then either p or q can be in the Q. If p is in the pocket then q and t are separated by w as t is in Q and q is not. If q is in the pocket then p and s are separated by w as s is in Q and p is not. From which follows that either, z or r lies on the section of ∂P_{super} between q and t not containing e, or that z or r lies between p and s on e. In either case we have reached a contradiction since p and s should be on the same edge, and q and t can only be separated by u. □

Theorem 1. All chords between two edges of P_{super} are combinatorially equal.

Proof. Note that this is a special case of lemma 3 where $e' = e''$, i.e. the chords start on the same edge and they end on the same edge. Again we distinguish between two possible cases. One where we have two chords that have a common endpoint and another where we have two distinct chords. Both are proved using a proof by contradiction.

First, let e and e' be edges of P_{super}. Let $\overline{pq}$ and $\overline{ps}$ be two chords between e and e' with common endpoint p. Now suppose that $\overline{pq}$ can see a vertex v and that $\overline{ps}$ cannot. We know by the same reasoning as lemma 3 that s is in the pocket of w while q is not and that the endpoints are therefore separated by w. Where w again is a chord with reflex vertex r and endpoint z. From which follows that either z or r lies on the section of ∂P_{super} between q and s not containing e. This section of ∂P_{super} can only be a part of e', so z or r lies between q and s on e'. If either point lies between q and s on e' then $\overline{pq}$ and $\overline{ps}$ are not chords between the same edges of P_{super} and we have reached a contradiction.

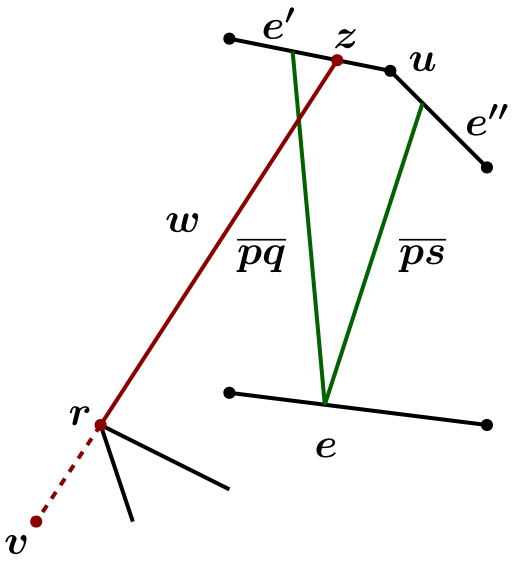
Second, let e and e' be edges of P_{super}. Let $\overline{pq}$ and $\overline{st}$ be two distinct chords between e and e'. Now suppose that $\overline{pq}$ can see a vertex v and that $\overline{st}$ cannot. We know by the same reasoning as lemma 3 that either p or q can be in Q. Where Q again is the pocket of a window w defined as a chord with reflex vertex r and endpoint z. If p is in the pocket then q and t are separated by w as t is in Q and q is not. If q is in the pocket then p and s are separated by w as s is in Q and p is not. From which follows that either, z or r lies on the section of ∂P_{super} between q and t not containing e, or that z or r lies between p and s on e. The section of ∂P_{super} between q and t can only be a part of e', so in this case z or r lies between q and s on e'. If an endpoint of w lies between p and s on e or between q and t on e' then $\overline{pq}$ and $\overline{st}$ are not chords between the same edges of P_{super} and we have reached a contradiction. □

We previously introduced the notion of the super polygon. From now on we differentiate between three different types of vertices in P_{super}

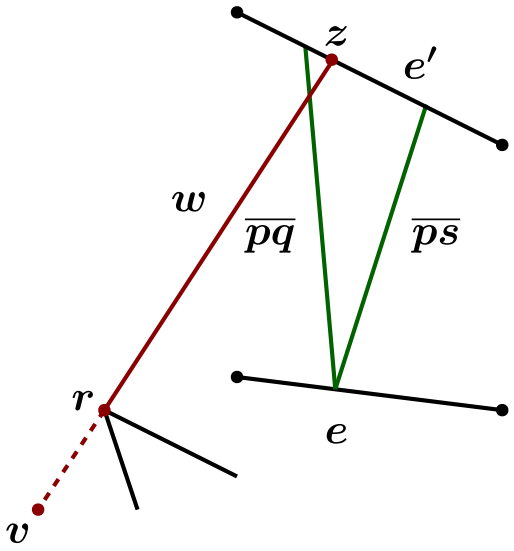
1. Vertices v with $v \in V(P)$ and v not reflex, i.e. regular vertices of P;
2. Vertices r with $r \in V(P)$ and r reflex, i.e. reflex vertices of P;
3. Vertices w with $w \in V(P_{super}) \setminus V(P)$, i.e. window vertices.

The following definitions are used to prove how the visibility changes around these different types of vertices. Let e, e' and e'' be edges of P_{super}. Let e' and e'' be neighbours with vertex u incident to both edges. Let $\overline{pq}$ be a chord between e and e' and let $\overline{st}$ be a chord between e and e''.

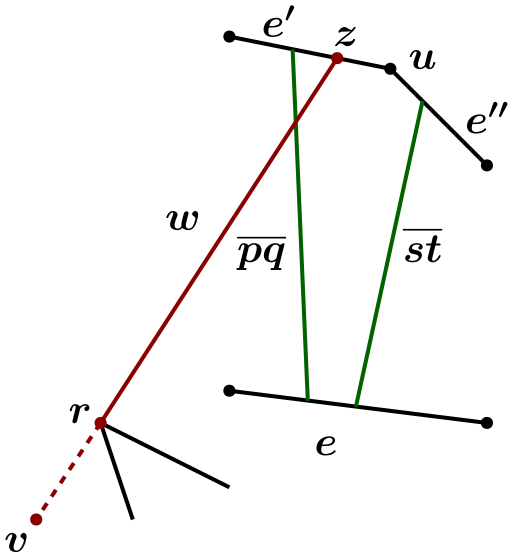
We say a super polygon is in general position when every window vertex is the endpoint of exactly one window. Note that even when a super polygon is in general position it can still have three or more collinear vertices due to the window vertices. We assume that P_{super} is in general position.



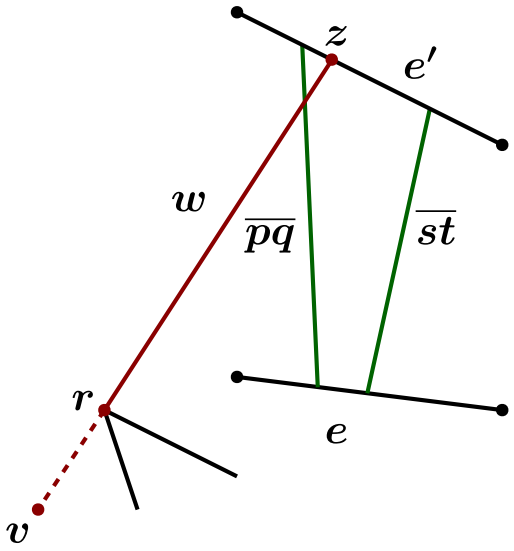
(a) When there is a point z on ∂P_{super} between the endpoints of $\overline{pq}$ and $\overline{ps}$ then there is some vertex v that is visible only to one chord



(b) A special case of figure 4a where the section of ∂P between the endpoints of $\overline{pq}$ and $\overline{ps}$ is a single edge.



(c) Even when $\overline{pq}$ and $\overline{st}$ have separate starting points, there only is some vertex v visible to one chord When there is a point z on ∂P_{super} between the endpoints of $\overline{pq}$ and $\overline{st}$



(d) A special case of figure 4c where the section of ∂P between the endpoints of $\overline{pq}$ and $\overline{ps}$ is a single edge.

Figure 4: A visual representation of why the chord visibility of $\overline{pq}$ and $\overline{st}$ only differs when $\overline{pq}$ intersects some window w as demonstrated in lemma 3 and theorem 1.

Lemma 4. The chord visibility of $\overline{pq}$ and $\overline{st}$ differs with at most one vertex when u is a window vertex.

Proof. We use a proof by contradiction. We know that u is the endpoint of exactly one window w collinear with some vertex v. Therefore the visibility polygons of $\overline{pq}$ and $\overline{st}$ can only differ with v. We also know that w separates e' and e''. From which follows that either $\overline{pq}$ or $\overline{st}$ must intersect w. Assume without loss of generality that $\overline{pq}$ intersects w. Now suppose that $\overline{pq}$ can see a vertex a and that $\overline{st}$ cannot. We know by the same reasoning as lemma 3 that either p or q can be in Q. Where Q is the pocket of a window w', defined as a chord with reflex vertex r and endpoint z, collinear with a. If p is in the pocket then q and t are separated by w' as t is in Q and q is not. If q is in the pocket then p and s are separated by w' as s is in Q and p is not. From which follows that either, z or r lies on the section of ∂P_{super} between q and t not containing e, or that z or r lies between p and s on e. From which follow a number of things. It is possible that z or r lies on the section of

∂P_{super} between q and t not containing e. It is also possible that z or r lies between p and s on e. Finally it is possible that u is an endpoint of w' as well as of w. In all cases we have reached a contradiction since p and s should be on the same edge, q and t can only be separated by u, and u can only be the endpoint of one window. □

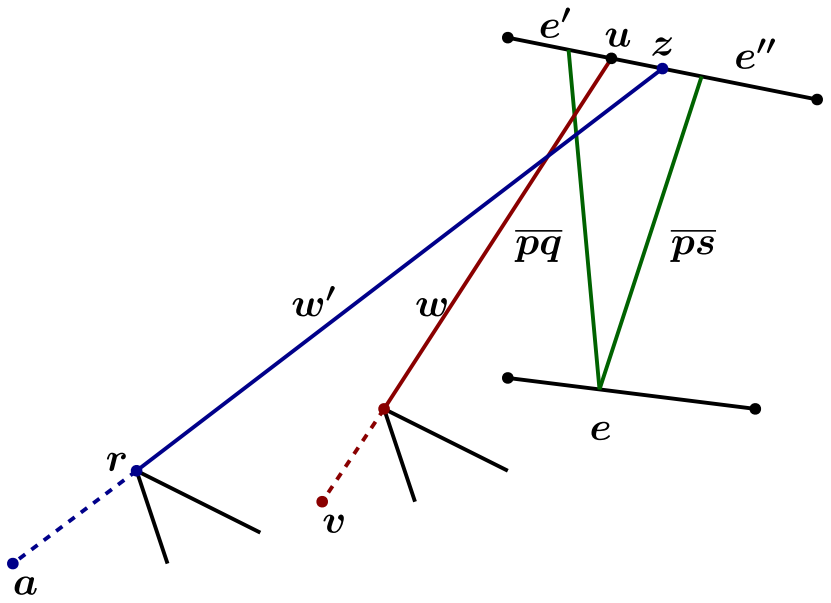


Figure 5: When $\overline{pq}$ and $\overline{ps}$ differ more vertices than just v, the vertex collinear with u, then e' and e'' are not neighbours.

Lemma 5. The chord visibility of $\overline{pq}$ and $\overline{st}$ is the same when u is a regular vertex of P.

Proof. A window is a chord $\overline{rz}$ where r is the reflex vertex where the window originates and where z is the endpoint of the window. Vertex u is a regular vertex of P so it is neither a reflex vertex nor a window vertex. It can therefore not be the endpoint of a window. From which follows that e' and e'' are not separated by a window. If $\overline{pq}$ and $\overline{st}$ are not separated by a window we can conclude from lemma 3 that both chords see the same set of vertices. □

Given an edge e of P_{super} we say H_e is a half-plane collinear with e. The inside of the half-plane can be directed either towards the interior or the exterior of the P_{super} as seen from e. We assume, unless otherwise noted that a half-plane H_e is always directed towards the interior of P_{super}.

Lemma 6. If e lies in $\mathcal{V}(r) \cap (H_{e'} \cap H_{e''})$ then there exists a chord $\overline{pq}$ from e to e' and a chord $\overline{st}$ from e to e''.

Proof. Let $H_{e'}$ be the half-plane collinear with e'. Let the inside of $H_{e'}$ be directed towards the interior of P_{super} as seen from e'. If $e \notin H_{e'}$ then e' is invisible from e. Similarly, let $H_{e''}$ be the half-plane collinear with e''. The inside of $H_{e'}$ is also directed towards the interior of P_{super} but now as seen from e''. If $e \notin H_{e''}$ then e'' is invisible from e. It follows that e must be in the intersection of $H_{e'}$ and $H_{e''}$ when both e' and e'' are visible from e. If e' and e'' are visible from e then r is also visible from e as r is incident to both edges. From which follows if e can see e' and e'' then e must also be in $\mathcal{V}(r)$. So when $e \in \mathcal{V}(r) \cap (H_{e'} \cap H_{e''})$, there is always some point $p \in e$ that can see some point $q \in e'$ and there is always some point $s \in e$ that can see some point $t \in e''$. From which we can conclude that if $e \in \mathcal{V}(r) \cap (H_{e'} \cap H_{e''})$ then there exists a chord $\overline{pq}$ from e to e' and a chord $\overline{st}$ from e to e''. □

Lemma 7. The chord visibility of $\overline{pq}$ and $\overline{st}$ is the same when u is a reflex vertex of P and $e \in \mathcal{V}(r) \cap (H_{e'} \cap H_{e''})$.

Proof. We use a proof by contradiction. Both half-planes $H_{e'}$ and $H_{e''}$ are not just collinear with e' and e'' but also with $w_{e'}$ and $w_{e''}$, the windows collinear with e' and e'' respectively. All windows $w_1, w_2, ..., w_n$, originating from u, and their corresponding half-planes $H_{w_1}, H_{w_2}, ..., H_{w_n}$ are outside of $H_{e'}$ and $H_{e''}$ respectively. Since $e \in H_{e'} \cap H_{e''}$, e is also in $\bigcap_{i=1}^{n} H_{w_i}$. From which follows that the chord visibility of $\overline{pq}$ and $\overline{st}$ is the same. Now suppose that $\overline{pq}$ can see a vertex v and that $\overline{st}$ cannot. We know by the same reasoning as lemma 3 that either p or q can be in Q. Where Q is the pocket of a window w', defined as a chord with reflex vertex r and endpoint z, collinear with v. If p is in the pocket then q and t are separated by w' as t is in Q and q is not. If q is in the pocket then p and s are separated by w' as s is in Q and p is not. From which follows that either, z or r lies on the section of ∂P_{super} between q and t not containing e, or that z or r lies between p and s on e. In either case we have reached a contradiction since p and s should be on the same edge, and q and t can only be separated by u. □

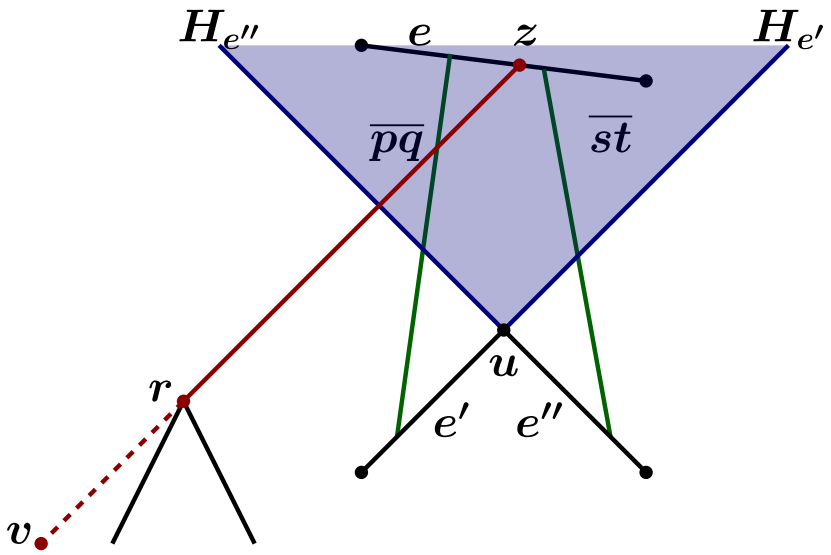


Figure 6: When e is in $\mathcal{V}(r) \cap (H_{e'} \cap H_{e''})$ then $\overline{pq}$ and $\overline{st}$ have the same chord visibility. There is no vertex v such that it is visible from $\overline{pq}$ and not from $\overline{st}$ since then there would be a point z between p and s or between q and t.

Combining lemmas 4, 5 and 7 gives the following theorem.

Theorem 2. The chord visibility of two chords in P_{super} that start on the same edge and that end on neighbouring edges differs with at most one vertex.

Everything we have proven so far is only true when the chords lie completely in the interior of P, i.e. the chords do not intersect ∂P. Now we will proceed to show what happens when a chord does intersect ∂P. We do not formally prove what happens but instead provide some intuition on what happens.

In the situation described above we always assumed that given three edges e, e' and e'' of P_{super}, with e' and e'' adjacent, there would be two chords $\overline{pq}$ between e and e', and $\overline{st}$ between e and e''. However this is not always the case, it is always possible for these chords to intersect ∂P which would make them invalid. Now we will discuss what happens when there is some chord $\overline{pq}$ between e and e' but no chord between e and e''. In this case there is some edge of P_{super}, e''' such that e''' is the first edge after e'' that has a valid chord with e. Now let $\overline{pq}$ be a chord between e and e', and $\overline{st}$ be a chord between e and e'''. Note that in this scenario $\overline{st}$ sees at least the same set of vertices as $\overline{pq}$.

Claim 1. A vertex $v \in P$ on the section of ∂P_{super} between e and e''' not containing e' and e'' is always visible from $\overline{st}$ when the endpoint u of a window w collinear with v lies on the section of ∂P_{super} between e'' and e''' not containing v.

If u lies between e'' and e''', and v lies between e and e''' then two tings can occur. Either v was already visible from $\overline{pq}$ or it was not. When v was already visible then it does not need to be added to the vertices visible from $\overline{st}$. If it was not then v can be added safely to the set of vertices visible from $\overline{st}$ because u and v are on different sides of $\overline{st}$, i.e. the window w ending in u intersects $\overline{st}$.

Claim 2. A vertex $v \in P$ on the section of ∂P_{super} between e'' and e''' not containing e is visible from $\overline{st}$ when $\overline{st}$ intersects $\mathcal{V}(v)$.

Any vertex v of the original polygon P that lies between e'' and e''' can see $\overline{st}$ if and only if some part of $\overline{st}$ lies in $\mathcal{V}(v)$. Given v and P_{super} we can retrieve all windows collinear with v and combine them with P to create $\mathcal{V}(v)$. When we have $\overline{st}$ and $\mathcal{V}(v)$ then it is simple to compute if they intersect and if v is indeed visible from $\overline{st}$.

Claim 3. There are no vertices of P other then those described in claims 1 and 2 that are visible from $\overline{st}$ but invisible from $\overline{pq}$.

We know for all vertices of P between e and e' if they are visible from $\overline{pq}$ and therefore also if they are visible form $\overline{st}$. We can also check for all vertices of P between e'' and e''' if $\overline{st}$ intersects their visibility polygons or not. Which only leaves the vertices between e and e'''. Any vertex v between e and e''' that cannot see $\overline{pq}$ but that can see $\overline{st}$ must have a window w intersecting $\overline{st}$ because otherwise it would be visible form $\overline{pq}$ as well. From which follows that the endpoint u of w must lie between e'' and e'''.

From which follows that there are no vertices other than those of claims 1 and 2 that can be seen from $\overline{st}$ but not from $\overline{pq}$. Additionally, this also tells us that we only need to examine the vertices P_{super} between e'' and e''' to determine which vertices of P are visible from $\overline{st}$.

7.2 Polygon Pre-processing

Given a polygon P we first need to construct the super polygon P_{super} of P since the chord visibility algorithm works on P_{super} and not on P. Computing P_{super} is done by simply computing $\mathcal{V}(v)$ for each $v \in P$ and adding the endpoints of the windows of $\mathcal{V}(v)$ to a copy of P. We also store for each vertex of P_{super} which, if any, windows end in that vertex.

Computing $\mathcal{V}(v)$ for each $v \in P$ can be done in linear time with a number of existing visibility algorithms. The vertices of $\mathcal{V}(v)$ that are not vertices of P can then simply be appended to a copy of the geometric representation of P, a circular list containing the vertices of P. The list can be sorted in $O(n \log n)$ time using a number of sorting algorithms when all new vertices are added. We need to compute n visibility polygons which takes a total of $O(n^2)$ time. A polygon contains $O(n^2)$ windows [6], so we need to add $O(n^2)$ window endpoints to P. Sorting all points of P_{super} therefore takes $O(n^2 \log n)$ time. The super polygon requires $O(n^2)$ space since it consists of the n original vertices of P as well as $O(n^2)$ window endpoints.

A window w is stored as a triplet of three vertices. The first is the reflex vertex r where w originates. The second is the window vertex u where w ends. The third is the vertex v of P collinear with w. Storing all windows in a data structure requires at least $O(n^2)$ space. We store the windows in a window list, a list where the windows that have vertex i of P_{super} as an endpoint are stored at index i of the window list. As a result we store each window twice, once for each endpoint. This data structure requires $O(|V| + |W|)$ space, where $|V|$ is the number of vertices of P_{super} and $|W|$ is the number of windows. Since both $|V|$ and $|W|$ are of quadratic size, the space required by the window list is $O(n^2)$. The window list allows us to retrieve the

windows of vertex i in $O(1)$ time. If i has more then one windows we can find the the window of i we need in $O(n)$ time.

The pre-processing step is bound by the sorting operation and therefore requires $O(n^2 \log n)$ time. Pre-processing a polygon P into the super polygon and a window list can be done in $O(n^2 \log n)$ time and $O(n^2)$ space.

7.3 Computing Chord Visibility

We present an algorithm that computes all chords originating from some edge e of P_{super}. The algorithm keeps e fixed throughout the procedure and performs a sequential scan along the other edges of ∂P_{super} starting at the right neighbour of e. The scan proceeds in counterclockwise order and terminates when e is reached. During each step of the procedure we keep track of two values. The first value is the maximum chord visibility value that was found so far. The second value is the total chord visibility value of all chords found so far.

During each step of the procedure we must check if there exists some chord $\overline{pq}$ between e and the edge e' that is currently being processed. We assume that p is always on e and that q is always on e'. For the algorithm we consider the edges of P_{super} to be open intervals, i.e. the endpoints are excluded from the edges.

When an edge e' of P_{super} is processed and the algorithm moves to e'', the next edge of P_{super}, then the algorithm might reach an edge such that no chord $\overline{pq}$ exists between e and e''. We can check if a chord exists in $O(1)$ time using the data structure presented in [12]. The data structure is a convex subdivision of the dual plane. Each convex region of the subdivision represents all chords between e and some other edge of P_{super}. If there exists a chord between e and e'' then there always exists a nonempty region for e'' in the data structure. Constructing the data structure for an edge e of P_{super} requires $O(n^2 \log n)$ time and $O(n^2)$ space.

Two things can happen when the algorithm moves from some edge e' to the next edge e'' and a valid chord $\overline{pq}$ exists between e and e''. We either add a single vertex to the set of visible vertices or we add no vertices. Observe that when the algorithm moves to e'' it crosses the vertex u of P_{super} incident to e' and e''. What happens to the chord visibility depends on u. If u is a window vertex then by lemma 4 the chord visibility can only change with one vertex as compared to the chord visibility during the previous step. If u is a regular vertex of the original polygon then by lemma 5 the chord visibility does not change as compared to the previous step. If u is a reflex vertex of the original polygon then by lemma 7 the chord visibility also does not change as compared to the previous step.

If the algorithm moves from some edge e' to the next edge e'' but no valid chord $\overline{pq}$ exists between e and e'', then we find the first edge e''' for which there exists a valid chord with e. The algorithm scans the boundary between e'' and e''' to compute how the chord visibility changes as compared to the chord visibility between e and e'. When e''' is reached the algorithm can proceed as before. In the following, we present the major steps for computing the chord visibility between a fixed edge e and all other edges of P_{super}.

Algorithm 3 Compute the chord visibility for a fixed edge e

Input. The super polygon P_{super}, the window List L and the edge e.
Output. The maximum and average chord visibility values of P.

1: $e' = e$
2: e'' is the right neighbour of e
3: $max_{vis} = 0$
4: $tot_{vis} = 0$
5: $num\_chords = 0$
6: S is the set of visible vertices
7:
8: **do**
9: **if** there exists a chord between e and e'' **then**
10: u is the vertex incident to e' and e''
11:
12: **if** u is a window vertex **then**
13: w is the window with endpoint u
14: v is the vertex of P collinear with u and w
15: **if** v is not in S **then** add v to S **end if**
16: **end if**
17: **else**
18: FindNextChord($P_{super}, L, S, e, e', e''$)
19: replace S with the set returned by FindNextChord
20: e'' is the edge returned by FindNextChord
21: **end if**
22:
23: **if** $|S| > max_{vis}$ **then** $max_{vis} = |S|$ **end if**
24: tot_{vis} += $|S|$
25: $num\_chords++$
26:
27: $e' = e''$
28: e'' is the next counter clockwise edge of P_{super}
29: **while** $e'' \neq e$
30:
31: $avg_{vis} = tot_{vis}/num\_chords$
32: **return** avg_{vis} and max_{vis}

Algorithm 4 Find the next valid chord originating from e

Input. The super polygon P_{super}, the window List L, the set of visible vertices S and the edges e, e', e''.
Output. The updated set of visible vertices S and the edge e'''.

```
1:  procedure FindNextChord(P_super, L, S, e, e', e'')
2:      e''' is the first edge of P_super, after e'', that has a valid chord with e
3:      pq is a chord between e and e'''
4:
5:      do
6:          u is the vertex incident to e' and e''
7:
8:          if u is a window vertex then
9:              w is the window with endpoint u
10:             v is the vertex of P collinear with u and w
11:
12:             if e''' is closer to e than e' then
13:                 if w does not intersect pq and v is not visible from pq then remove v from S end if
14:             else
15:                 if w intersects pq then add v to S end if
16:             end if
17:         else
18:             retrieve V(u)
19:
20:             if e''' is closer to e than e' then
21:                 if pq does not intersect V(u) then remove v from S end if
22:             else
23:                 if pq intersects V(u) then add u to S end if
24:             end if
25:         end if
26:
27:         e' = e''
28:         e'' is the next counter clockwise edge of P_super
29:     while e' ≠ e'''
30:
31:     return S and e'''
32: end procedure
```

We discussed previously that we can check in $O(1)$ time if there is a chord between e and some other edge of P_{super}. The algorithm also requires us to efficiently compute whether a vertex $v \in P$ is visible from some chord between e and an edge e'. We can perform this check by constructing $\mathcal{V}(v)$ and checking if there is an intersection with the chord, which can be done trivially in $O(n)$ time using existing visibility algorithms. Fortunately it is possible to make these checks more efficiently using the data structure from [13]. Here researchers present a data structure on P that we can query given a line segment $s \in P$ and a vertex $v \in P$ to test if s intersects $\mathcal{V}(v)$. Building the data structure for P_{super} requires $O(n^2 \log n)$ space and querying the data structure takes $O(\log^2 n)$ time.

The algorithm processes each edge of P_{super} at most twice and there are $O(n^2)$ edges. We represent S, the list of visible vertices, as a tuple containing a list representing the visible vertices and an integer value representing the number of visible vertices. The list is of length n and stores a 0 or 1 at every position. Each position corresponds to the vertex of the geometric representation of P at that same position, i.e. the indices of the geometric representation match those of the list of S. Adding a vertex to S takes $O(1)$ time as we only have to flip the value at the position corresponding to the vertex and increase the number of visible vertices with one. Removing a vertex from S is done similarly in $O(1)$ time as well. Checking which window ends in a vertex can be checked in $O(1)$ time using the window list. Of course this takes $O(n)$ time for reflex vertices but we need to check only for window vertices, so for the algorithm we can indeed conclude it only needs $O(1)$ time. From which follows that it takes at most $O(\log^2 n)$ time to process an edge. The total running time of

the algorithm is therefore $O(n^2 \log^2 n)$.

The algorithm can be extended to compute the maximum and average chord visibility for a polygon P by repeating it for every edge $e \in P_{super}$ and taking the maximum value of all maximums and average value over all averages. There are $O(n^2)$ edges in P_{super} and the algorithm takes $O(n^2 \log^2 n)$ time for each edge so computing the maximum and average values for P requires $O(n^4 \log^2 n)$ time.

We believe that it is also possible to build the subdivision for an edge of P while all regions of the subdivision represent edges of P_{super}. This way we would no longer have to build the sub division for every edge of the algorithm and it would suffice to build n subdivisions during pre-processing. Thus speeding up future algorithms.

8 Implementation

An implementation of several polygon parameters is combined in a computer application with the purpose of testing if there exists some correlation between the parameters and the running times of visibility algorithms. All algorithms are implemented in C++ using the Computational Geometry Algorithms Library (CGAL), an open source library of robust geometric algorithms and data structures. In this section we provide details about the implementation and the library. We also discuss some important implementation details in more depth.

8.1 CGAL

The Computational Geometry Algorithms Library (CGAL) is a joint project of different universities in Europe. The project started in 1996 and new versions of CGAL are still in development. The latest release CGAL 5.1 was released in September of 2020. Among its major design goals are robustness, generality and efficiency.

CGAL achieves its robustness through exact computations of geometric predicates and constructors, which often require the use of special number types. A geometric predicate typically computes the sign of an expression. A constructor produces a new geometric object, i.e. the intersection point of two lines. CGAL also allows the use of more common number types such as the machine float or double, but then the algorithms are not always guaranteed to produce the correct output.

CGAL contains a package for visibility computations. Included in this package are efficient implementations of the triangular expansion algorithm [4], the sequential algorithm by Joe and Simpson [5] and the rotational sweep algorithm [1]. Naturally algorithms to compute our polygon properties are not included. The visibility algorithms are not implemented as part of the project since efficient implementations are already available in CGAL.

8.2 Application Overview

The main application is written in C++ and compiled using Cmake. It has been compiled and tested for both Windows and Linux 64 bit operating systems. The application takes a single polygon P in the geoJSON or graphML format as input and returns data about the calculated properties and the performance of CGAL's visibility algorithms on that polygon.

The application is divided into three main packages Algorithms, IO and Progress. The algorithms package contains the implementations off the different parameter algorithms and is where the performance of CGAL's visibility algorithms is measured. The IO package contains all code related to IO. The Progress package contains the implementation of a basic progress bar to track the program during execution.

All packages work completely independent from each other but do expect certain input types, each package functions properly as long as the right type of input data is provided. This allows the packages to be modified, updated or replaced in the future without the need to change anything outside of that package. The application code is structured as follows.

```
src/
├── Algorithms/
│   ├── Algorithms.h
│   ├── Parameters.cpp
│   └── Visibility.cpp
├── IO/
│   ├── IO.h
│   ├── Json_parser.cpp
│   └── XML_parser.cpp
├── Progress/
│   └── ProgressBar.h
├── data_types.h
├── precomp.h
└── main.cpp
```

In addition to the main application that computes the polygon data we also made some smaller scripts to aid during the experiments. The scrips are available for Windows as well as Linux. Firstly there is a script to pre-process geoJSON data if required. It is possible that several polygons are stored in an array in a single geoJSON file, this script takes those polygons out of the array and stores them as separate files. This is required since the main application only takes one polygon as input. Secondly there is a script that takes the JSON data generated by the main program and converts it into a spreadsheet. Thus converting the data into a format much more suitable for analysis. Both scripts are written in JavaScript with Node.js and compiled as stand alone executables. Finally there is a command line script that given a folder containing a polygon dataset, executes the main C++ application for each polygon and when all polygons are processed runs the script that converts the output data into a spreadsheet.

8.3 Algorithms Package

The two main parts of the application are implemented in this package. The polygon properties we defined previously are computed here. The performance of CGAL's visibility algorithms is measured here as well. Which is done by computing $\mathcal{V}(v)$ for each vertex v of the polygon with each of the visibility algorithms in CGAL. The result of the performance test is the total amount of time taken by each of the algorithms as well as the minimum, maximum, mean and median visibility polygons of the test.

The application computes four different polygon properties. It computes the number of reflex vertices of the polygon. It computes how close the polygon is being a convex polygon. It computes the size of the largest visibility polygon, note that this does not have to be the largest polygon from the performance test. Finally it computes the diagonal visibility of the polygon.

The number of reflex vertices is computed by scanning the boundary of the polygon. During the scan we check for each vertex v if the edges incident to v greater then π on the inside of P.

We introduced two different polygon convexity parameters in chapter four but only one of them is implemented in the application, the hull vertices parameter. This parameter is value between 0 and 1 that represents how many vertices of P are on the convex hull. To compute this parameter we simply compute the convex hull of P and divide the number of hull vertices by the total number of vertices.

To compute the size of the largest visibility polygon of P, we first compute the visibility decomposition of P [6]. Next we iterate over all visibility regions and compute their respective visibility polygons, note that this way we compute all combinatorially distinct visibility polygons of P. During this process we keep track of the size of the largest polygon encountered so far. We also track the total amount of vertices of all visibility polygons encountered so far, we use this value to compute the average size. When the process is finished we are left with the maximum and the average visibility polygon size of P.

To compute the diagonal visibility we first need to compute the diagonals of P. The diagonals are computed using a constrained Delauney triangulation of P. This results in a set of "nice" diagonals according to the definition of the Delauney triangulation. Note that the Delauney triangulation does not contain all diagonals

of P but instead is a subset of all diagonals. Next we iterate over all diagonals of the triangulation and compute which vertices of P are visible from each diagonal. This is done by simply scanning the boundary of P and computing $\mathcal{V}(v)$ for all vertices, checking if the diagonal is (partially) inside each visibility polygon. During the process we keep track of the maximum diagonal visibility encountered so far. We also track the total diagonal visibility encountered so far, we use this value to compute the average value. When the process is finished we are left with the diagonal visibility value as well as the average diagonal visibility value.

8.4 IO and Progress Packages

The IO and Progress packages are not part of the core functionality of the application but instead contain important utility functions. The progress package is the smallest of the twp and it contains a basic ascii progress bar. The application lacks a graphical interface and runs from the command line, ascii progress bars are a nice and convenient way to keep track of the progress made by the application.

The IO package handles input and output for the application. We use the "nlohmann/json" library for handling JSON (geoJSON) data and the "Libxml2" library for handling XML (graphML) data. The package is capable of parsing polygons stored in geoJSON and graphML formats into the internal representation. During parsing, it also checks if the polygons are indeed simple polygons. The IO package also takes the data generated by the performance test and property calculations, appending it to a JSON file containing output of previous runs of the application. The output is written as JSON so the results can be easily interpreted or used by other applications.

9 Experiments

The application discussed in the previous section is used to compute data on a range of polygons. In this section we discuss the setup and dataset on which these experiments are executed. We also discuss the results of the experiments.

9.1 Experimental Setup

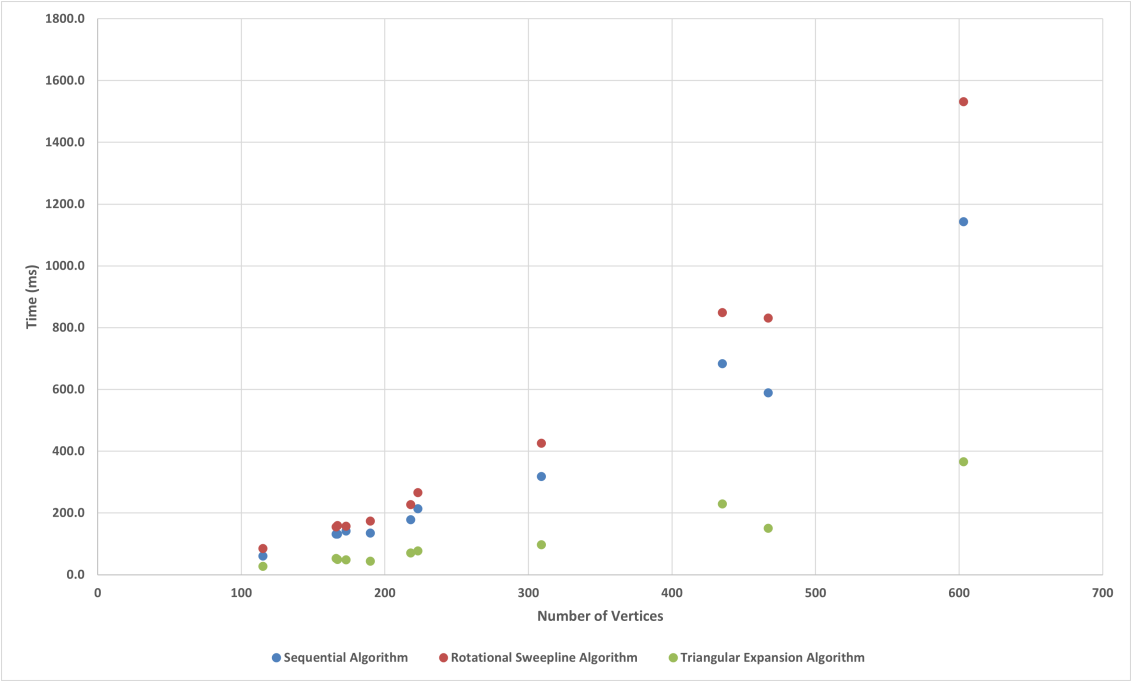
The experiments are executed on an Intel Core i7-920 at 2.67 GHz with 12 GB of system memory running a Linux 64 bit (Ubuntu 20.04.2) operating system. The algorithms used are from version 5.0.2 of CGAL. None of the algorithms use parallelization. We used two different data sets, one with polygons from the real world and another with generated polygons. The size of the polygons varies from 10 to about 5000 vertices.

The real world dataset consists polygons representing the Dutch provinces [14]. This set contains 11 simple polygons and one polygon with holes (North Brabant). We use the 11 simple polygons as the real world dataset.

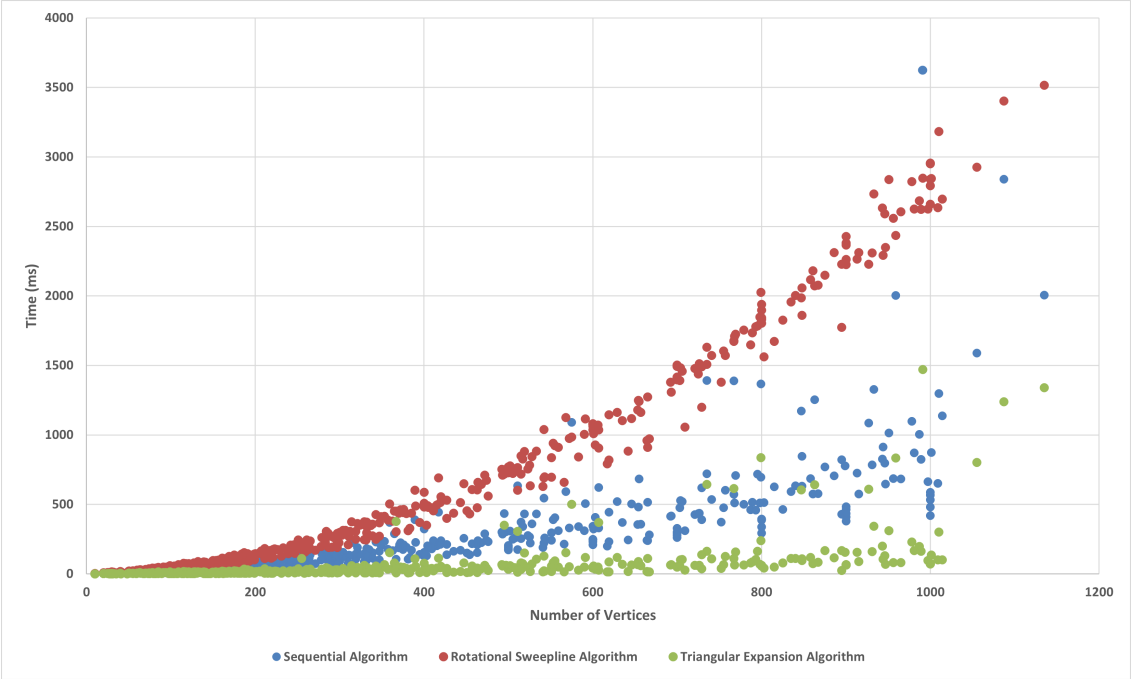
The Salzburg database [15] aims to provide a repository of random inputs for geometric algorithms. The database contains simply-connected and multiply-connected polygons in two dimensions. We use a subset containing 517 of the simply-connected polygons as the generated data. Most polygons in this subset contain 1000 vertices or less. We also added polygons of 2000, 3000, 4000 and 5000 vertices to obtain data on larger polygons as well. The polygons in the dataset are generated by different generators. We use polgons generated by the SPG, SRPG and RPG generators. The SPG algorithm constructs a simple polygon on a given point set by combining a line sweep algorithm with 2-opt moves. A 2-opt move takes a part P' of ∂P and reverses the order of the vertices of P' in ∂P. The super random polygon generator (SRPG) generates simply-connected and multiply-connected polygonal areas by means of a regular grid that consists of square cells. The random polygon generator (RPG) includes various heuristics to generate random polygons for a given set of vertices.

9.2 Results

In this subsection we discuss the most significant results of the experiments. The complete set of results can be found in Appendix A. The real world dataset contained too few polygons to deduce a meaningful correlation from the results, therefore we will mostly focus on the results of the generated polygons.



(a) Computation times of the real world dataset.



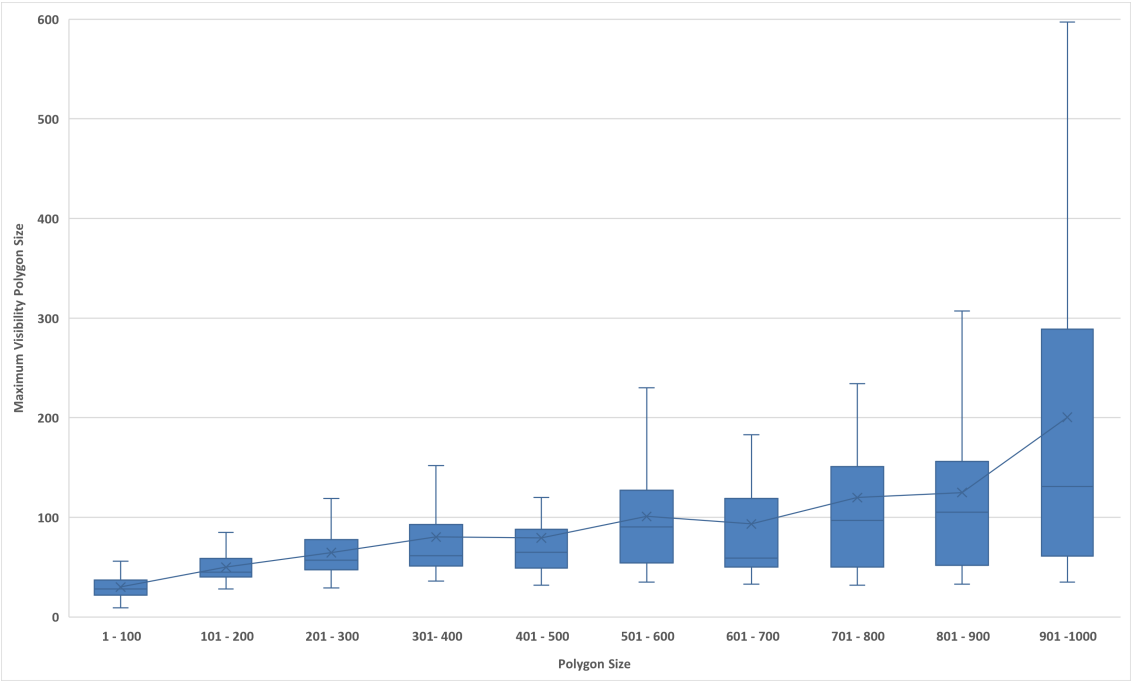
(b) Computation times of all generated polygons smaller than 2000 vertices.

Figure 7: Time to compute $\mathcal{V}(v)$ for all $v \in P$, for each visibility algorithm, plotted against the Number of vertices of P.

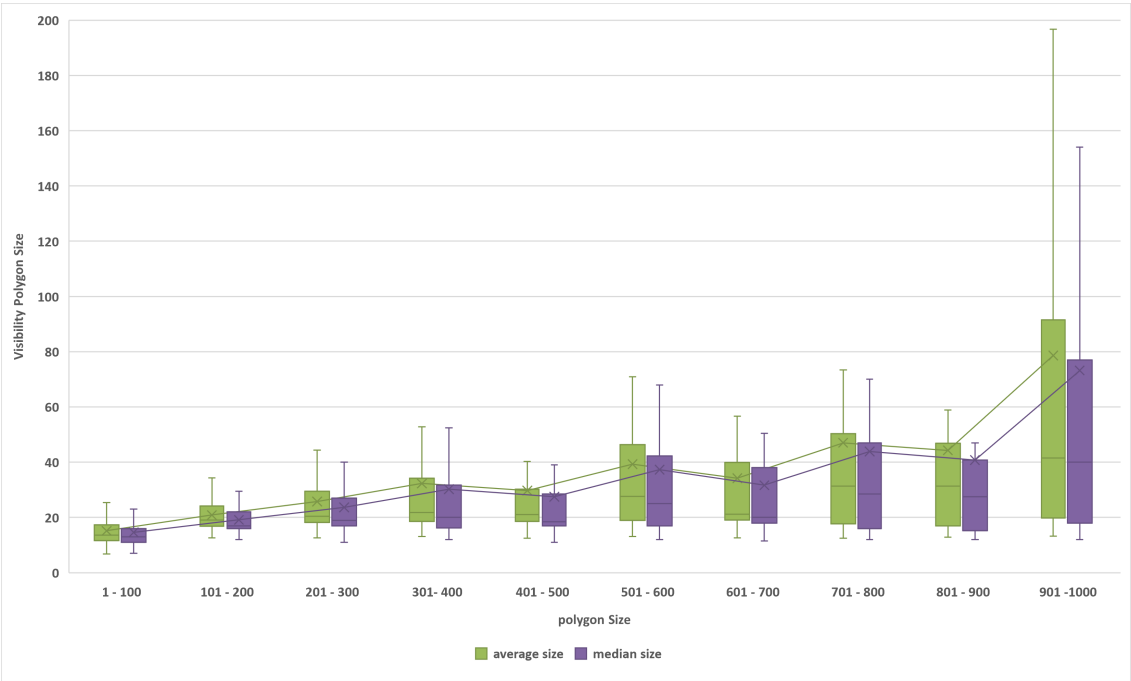
For the generated polygons, we can observe that the triangular expansion algorithm is faster than the other algorithms for the majority of polygons. The sequential algorithm is slower overall than the triangular expansion algorithm even though they both run in $O(n)$ time. This can be explained by the output sensitive

nature of the triangular expansion algorithm. The running time is actually $O(nh)$ where h is the number of holes of the polygon. The algorithm runs efficiently since there are no holes in the polygons and therefore has a running time similar to $O(n)$. However the results suggest that it might be faster in practice. The rotational sweep algorithm is clearly the slowest algorithm which we expected as is the only algorithm with a $O(n \log n)$ running time.

For the real world polygons, we can also observe that the triangular expansion algorithm is faster then the sequential algorithm and the rotational sweep algorithm. However the results of the sequential and rotational sweep algorithms are to close to really tell which one is faster.



(a) Size difference of the maximum $\mathcal{V}(v)$ in P.



(b) Size difference of the average and median $\mathcal{V}(v)$ in P.

Figure 8: Size differences of the maximum, average and median $\mathcal{V}(v)$ in a polygon P for generated polygons of similar size.

We can observe that when the size of the polygons grows, the size of their visibility polygons differs more. This is because large polygons with many reflex vertices can have small visibility polygons while large polygons with few reflex vertices can have large visibility polygons. While this is true for polygons of all sizes it is much more apparent in large polygons. Note that we did not discuss the minimum visibility polygons, this is because their size does not differ for most polygon sizes.

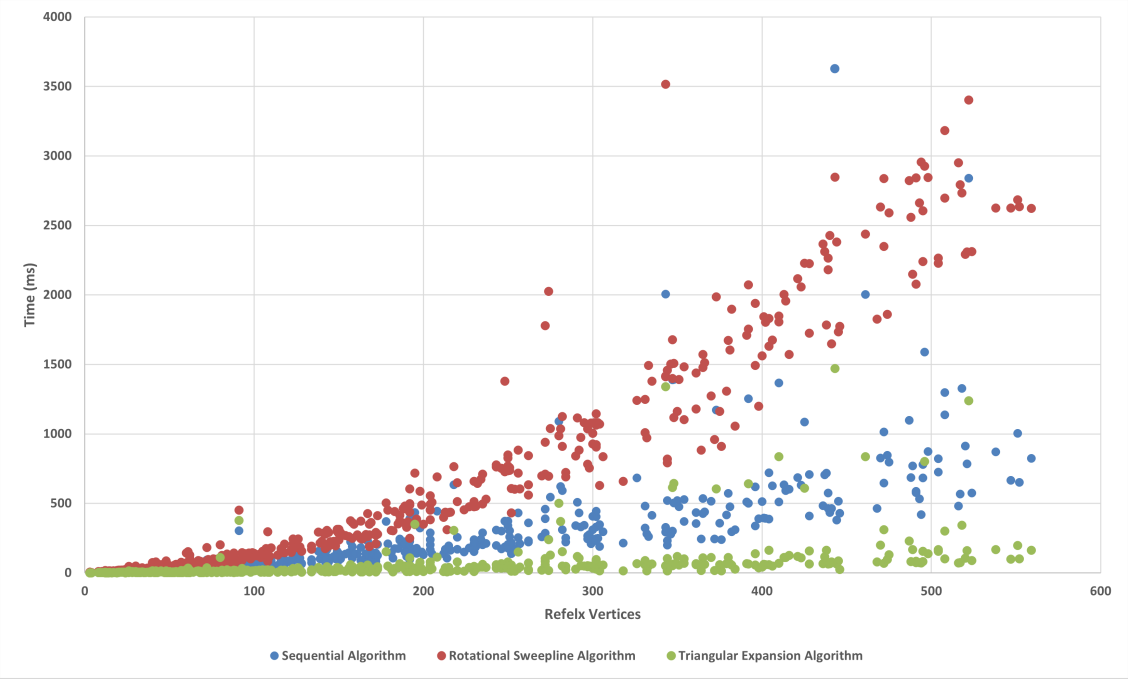


Figure 9: Time to compute $\mathcal{V}(v)$ for all $v \in P$, for each visibility algorithm, plotted against the Number of reflex vertices of P for generated polygons of less than 2000 vertices.

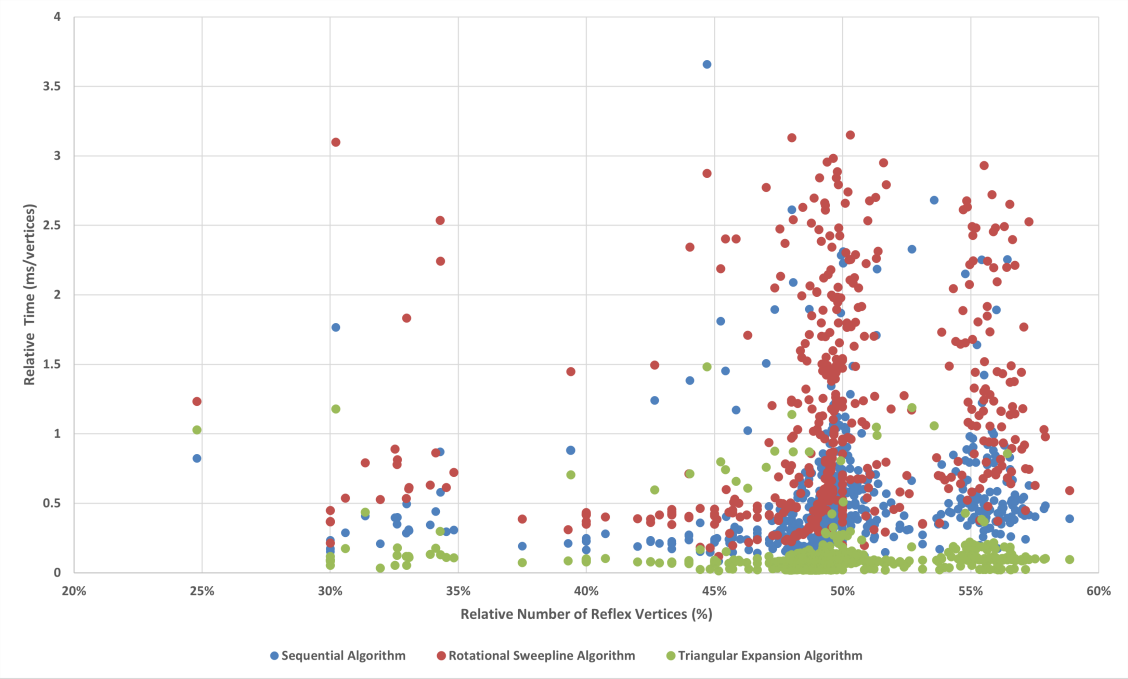


Figure 10: Relative time to compute $\mathcal{V}(v)$ for all $v \in P$, for each visibility algorithm, plotted against the relative number of reflex vertices of P for all generated polygons excluding outliers. The values are normalized by the number of vertices of P

We can observe from figure 9 that when the number of reflex vertices of a polygon increases, the time to compute $\mathcal{V}(v)$ for all v in a polygon P also increases. A possible explanation for this is that when a polygon has many reflex vertices, it also has many vertices. From which follows that the time to process these polygons is higher. Therefore we normalize the values on both axis by dividing them by the size of the polygons to remove the effect the polygon size has on this parameter. The result of the normalization is shown in figure 10. This figure shows two peaks in relative computation time when around 50% and 55% of the polygon vertices are reflex vertices. There is however no steady increase in relative computation time when the number of reflex vertices increases like there was in the non-normalized figure.

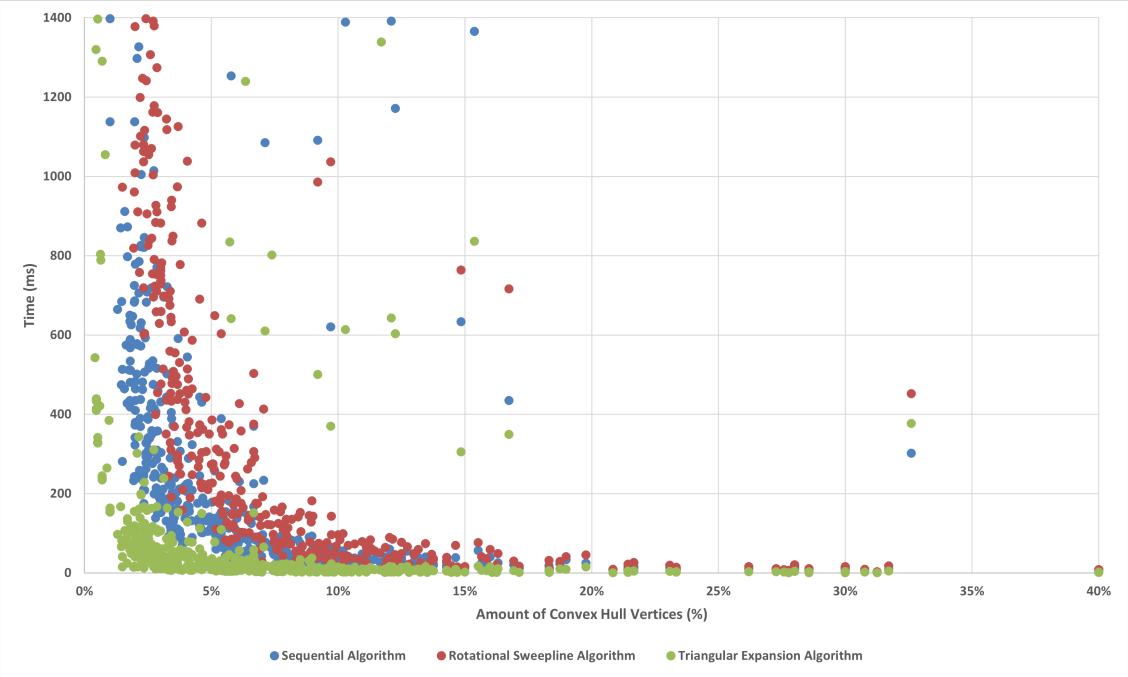


Figure 11: Time to compute $\mathcal{V}(v)$ for all $v \in P$, for each visibility algorithm, plotted against the percentage of convex hull vertices of P for all generated polygons excluding outliers.

Figure 11 shows that the time to compute $\mathcal{V}(v)$ for all v in a polygon P decreases as the percentage of convex hull vertices increases. A likely explanation for this is that when a polygon is more convex then there are less reflex vertices. If a polygon has few reflex vertices then few windows are computed when a visibility polygon is constructed, i.e. visibility polygons consist mostly of sections of the polygon boundary. We also normalized the computation time by dividing it by the number of polygon vertices but this resulted in a figure similar to the one without normalized time. This figure can be seen in appendix A.

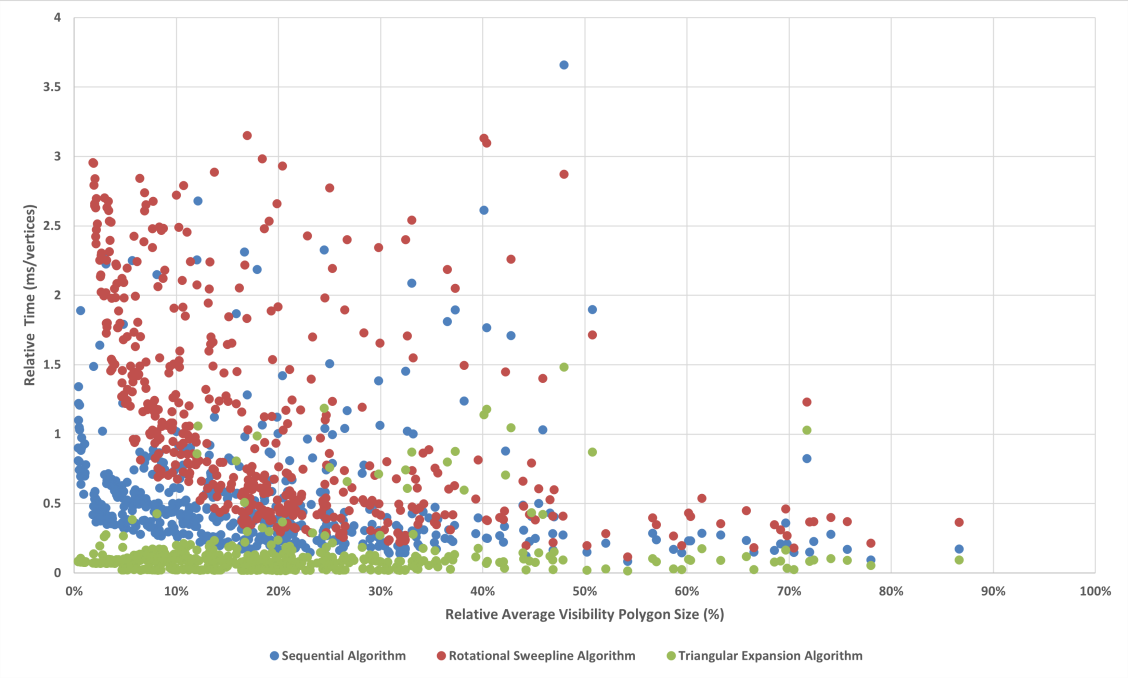


Figure 12: Relative time to compute $\mathcal{V}(v)$ for all $v \in P$, for each visibility algorithm, plotted against the relative average size of a visibility polygon in P for all generated polygons excluding outliers.

From figure 12, we can observe that when the average size of a visibility polygon inside a polygon P increases, the time it takes to compute $\mathcal{V}(v)$ for all $v \in P$ decreases. Note that the data in this graph is normalized by dividing both values with the size of the polygons. A possible reason for this decrease is that when the average size of a visibility polygon in P is large, P has few reflex vertices. The amount of time spent on a vertex of P is low when there are few reflex vertices because fewer windows need to be computed for a visibility polygon. The results of the normalized maximum visibility polygon size are similar but a little more scattered and are therefore omitted. The non-normalized results of average and maximum visibility polygon size show some increase in time as the size grows but are too scattered to provide meaningful insight.

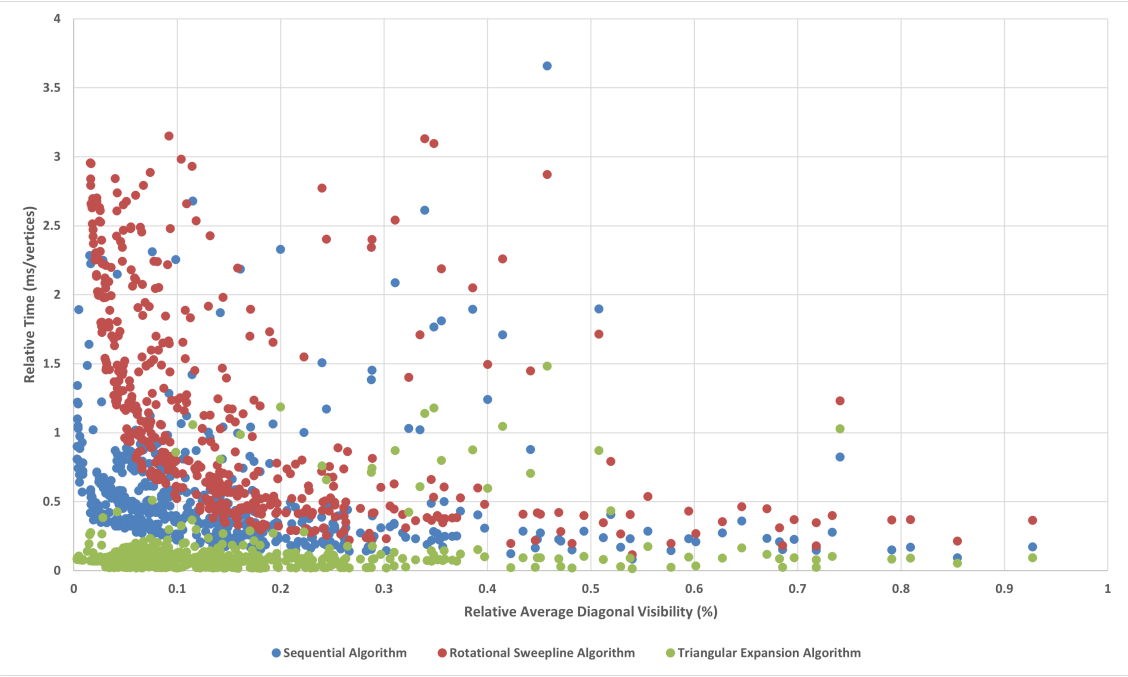


Figure 13: Relative time to compute $\mathcal{V}(v)$ for all $v \in P$, for each visibility algorithm, plotted against the relative average diagonal visibility of P for all generated polygons excluding outliers.

We can observe from figure 13 that when the average diagonal visibility of a polygon P increases, the time it takes to compute $\mathcal{V}(v)$ for all $v \in P$ decreases. Note that the data in this graph is normalized by dividing both values with the size of the polygons. A possible reason for this is that when the diagonal visibility of P is large, P has few reflex vertices to block the view from a diagonal. The amount time spent on a vertex of P is low when there are few reflex vertices because fewer windows need to be computed for a visibility polygon. The results of normalized diagonal visibility are similar but a little more scattered and are therefore omitted. The non-normalized results of (average) diagonal visibility show some increase in time as the size grows but are too scattered to provide meaningful insight.

10 Conclusion

We presented polygon properties that can be used in the analysis of visibility algorithms. We proved several properties about chords and their visibility inside a polygon P. We used these proofs to present a novel algorithm to compute the chord visibility property. Using $O(n^2 \log n)$ space and $O(n^4 \log^2 n)$ time we can compute the chord visibility of a polygon P by scanning the boundary of the polygon.

We implemented algorithms to compute the visibility polygon size, reflex vertices, diagonal visibility and polygon convexity properties. We computed the properties on a number of polygons to see if there is some correlation between each property and the time it takes to compute a visibility polygon in practice. We hypothesized that the running time of visibility computations is low when a polygon has a near convex shape. The experiments confirmed that the computation time indeed decreases when the percentage of convex hull vertices of a polygon rises. This is also the case when the data is normalized.

We initially believed that the time it takes to compute a visibility polygon would decrease when the size of the maximum and average visibility polygon would decrease. However, the results showed that the time it takes to compute a visibility polygon decreases when the average visibility polygon size increases, but only when both values are normalized by the visibility polygon size. The results for the maximum visibility polygon size show the same relation. This suggests the hypotheses are incorrect but further confirms the results of the convex hull vertices property.

We also believed that the time it takes to compute a visibility polygon would decrease when the diagonal visibility is low. However, the results showed that the time it takes to compute a visibility polygon decreases when the average diagonal visibility increases, but again only when both values are normalized by the visibility polygon size. The results for the maximum visibility polygon size show the same relation. This suggests again that our hypotheses are incorrect but further confirms the results of the convex hull vertices property.

Our hypothesis for the final property, the number of reflex vertices of a polygon, was that the computation time would increase as the number reflex vertices increases. The non-normalize results of this property suggest that this is the case but they are influenced by the size of the polygons. The normalized results however do not show a clear increasing or decreasing relationship between the computation time and the number of reflex vertices. As a result we can neither confirm nor disprove the hypothesis of this property.

We conclude from the results that the convex hull vertices property is the most significant. This the only property that showed the same relationship with computation time in both the regular and normalized results. It is also the only property for which the results confirmed the hypothesis. The results also suggest that there is a correlation between the convex hull vertices property, the visibility polygon size property and the diagonal visibility property. Therefore the visibility polygon size and the diagonal visibility are also significant properties. The reflex vertices property is the least significant.

There are several directions for future work. We believe that it is possible to compute chord visibility more efficiently. One direction of future work would be to improve upon the current algorithm. It also seems that there is correlation between convex hull vertices, visibility polygon size and diagonal visibility properties, both with each other and with the computation time. A second direction would be to further investigate the correlation between the properties as well as the correlation between the properties and the computation time. A third direction of future work would be to design a visibility algorithm that uses the properties to compute visibility more efficiently.

Bibliography

[1] Tetsuo Asano. An efficient algorithm for finding the visibility polygon for a polygonal region with holes. *IEICE TRANSACTIONS*, 68(9):557–559, September 1985.

[2] Luis Barba, Matias Korman, Stefan Langerman, and Rodrigo I. Silveira. Computing a visibility polygon using few variables. *Computational Geometry Theory and Applications*, 47:918–926, 2014.

[3] Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[4] Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Krolley. Effcient computation of visibility polygons. March 2014.

[5] B. Joe and R.B. Simpson. Corrections to lee's visibility polygon algorithm. *BIT Numerical Mathematics*, 27:458–473, 1987.

[6] Prosenjit Bose, Anna Lubiw, and J. Ian Munro. Efficient visibility queries in simple polygons. *Computational Geometry Theory and Applications*, 23:313–335, 2002.

[7] B. Aronov, L.J. Guibas, M. Teichmann, and L. Zhang. Visibility queries and maintenance in simple polygons. *Discrete and Computational Geometry*, 27:461–483, 2002.

[8] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E.Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, February 1989.

[9] Matias Korman, André van Renssen, Marcel Roeloffzen, and Frank Staals. Kinetic geodesic voronoi diagrams in a simple polygon. In *Proc. 47th International Colloquium on Automata, Languages and Programming*, Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020. To Appear.

[10] Simon Hengeveld and Tillmann Miltzow. A practical algorithm with performance guarantees for the art gallery problem. *arxiv.org*, 2020.

[11] Fabian Klute, Meghana M. Reddy, and Tillmann Miltzow. Local complexity of polygons, 2021.

[12] Bernard Chazelle and Leonidas J. Guibas. Visibility and intersection problems in plane geometry. *Discrete Computational Geometry*, 4(6):551–581, December 1989.

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

[14] Hietbrink Joost. Geojson data of the netherlands, July 2015.

[15] Günther Eder, Martin Held, Steinþór Jasonarson, Philipp Mayer, and Peter Palfrader1. On generating polygons: Introducing the salzburg database. March 2020.

[16] Dan Halperin. Robust geometric computing in motion. *The International Journal of Robotics Research*, 21(3):891–921, March 2002.

[17] Mark de Berg, Herman Haverkort, and Constantinos P. Tsirogiannis. Visibility maps of realistic terrains have linear smoothed complexity. *Journal of Computational Geometry*, 1(1):57–71, 2010.

[18] Esther Moet, Marc van Kreveld, and A. Frank van der Stappen. On realistic terrains. *Computational Geometry Theory and Applications*, 41:48–67, 2008.

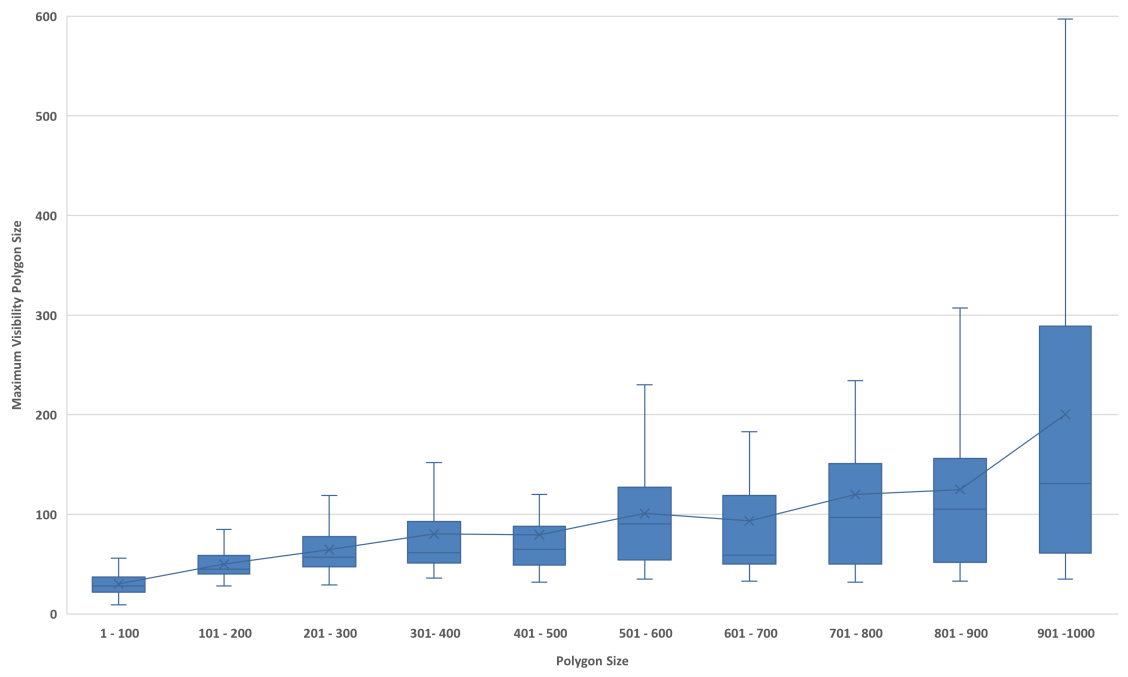
Appendices

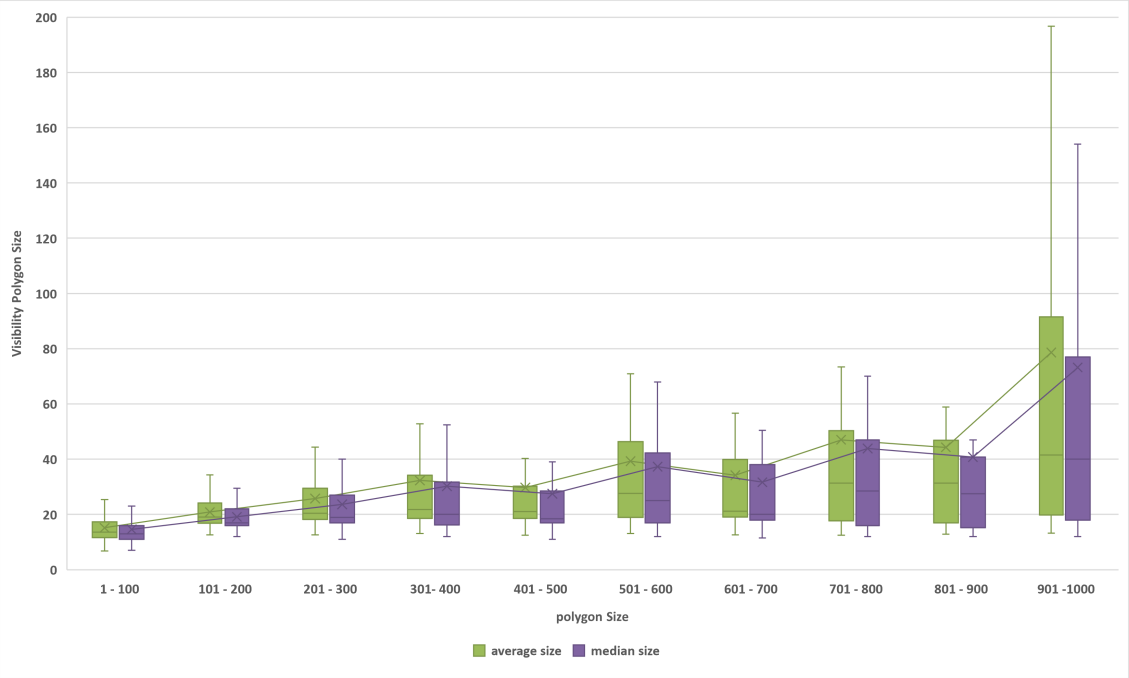
A Experimental Data

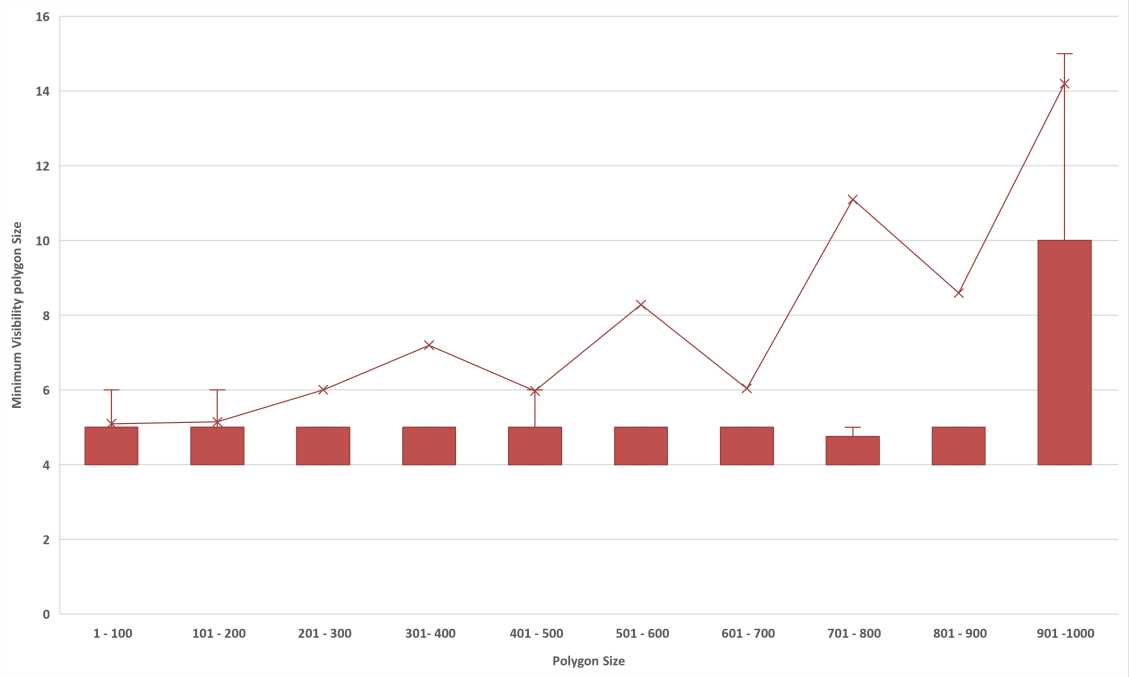
A.1 Generated Polygons

A.1.1 Visibility Polygon Size

The maximum, minimum, average and median visibility polygon size of all polygons, divided into groups of approximately the same size.

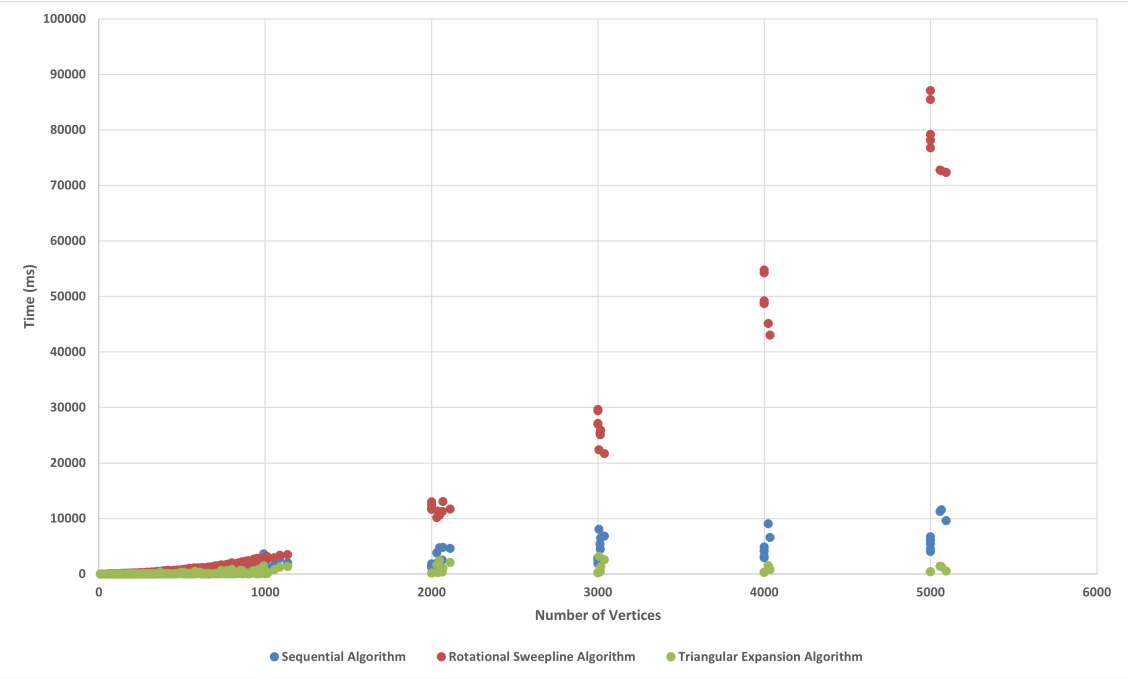


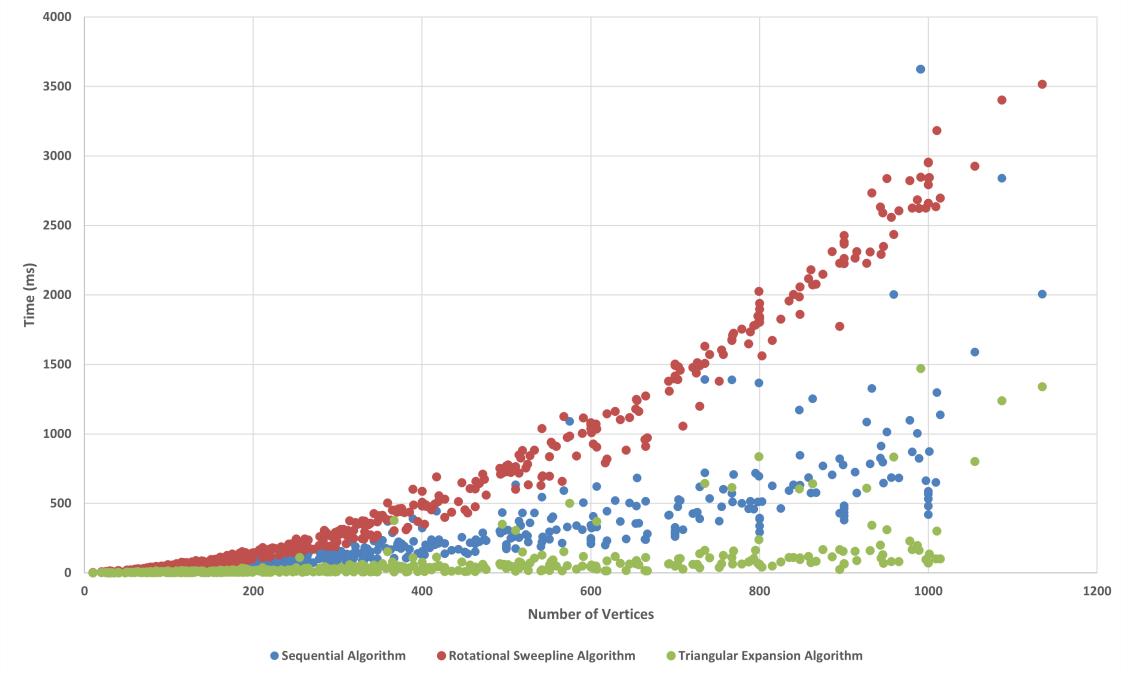

Visibility Polygon Size
200
180
160
140
120
100
80
60
40
20
0
1 - 100
101 - 200
201 - 300
301- 400
401 - 500
501 - 600
601 - 700
701 - 800
801 - 900
901 -1000
polygon Size
average size
median size


Minimum Visibility polygon Size
16
14
12
10
8
6
4
2
0
1 - 100
101 - 200
201 - 300
301- 400
401 - 500
501 - 600
601 - 700
701 - 800
801 - 900
901 -1000
Polygon Size

A.1.2 Computation Time

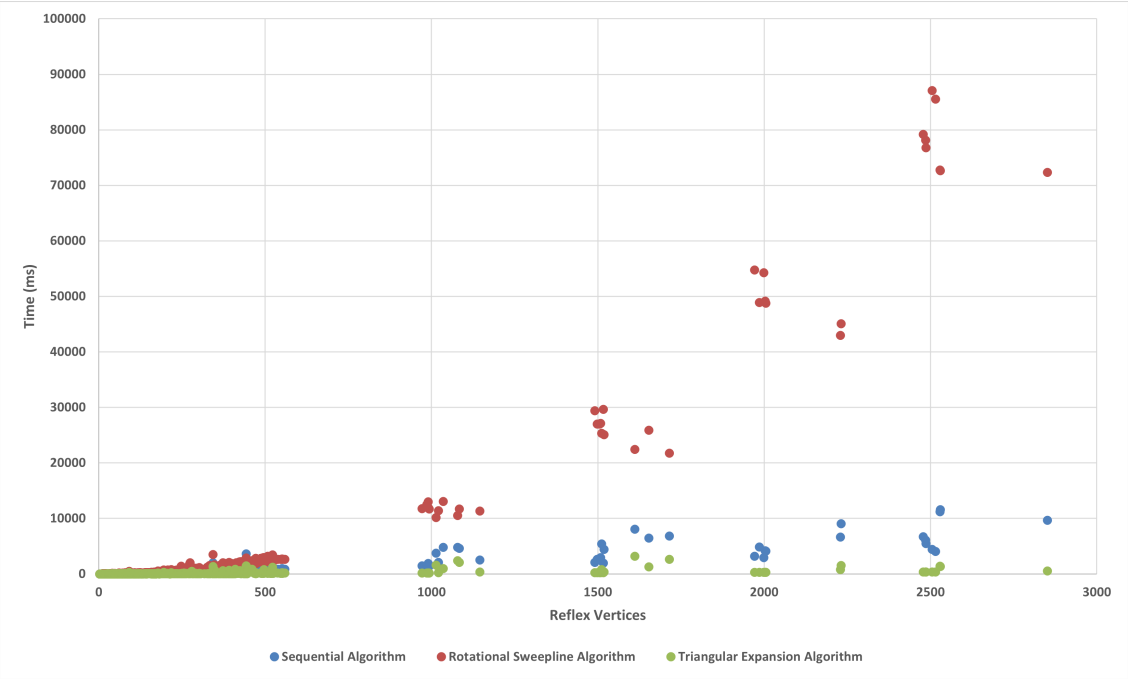
The computation times for all polygons and for the polygons smaller than 1200 vertices.

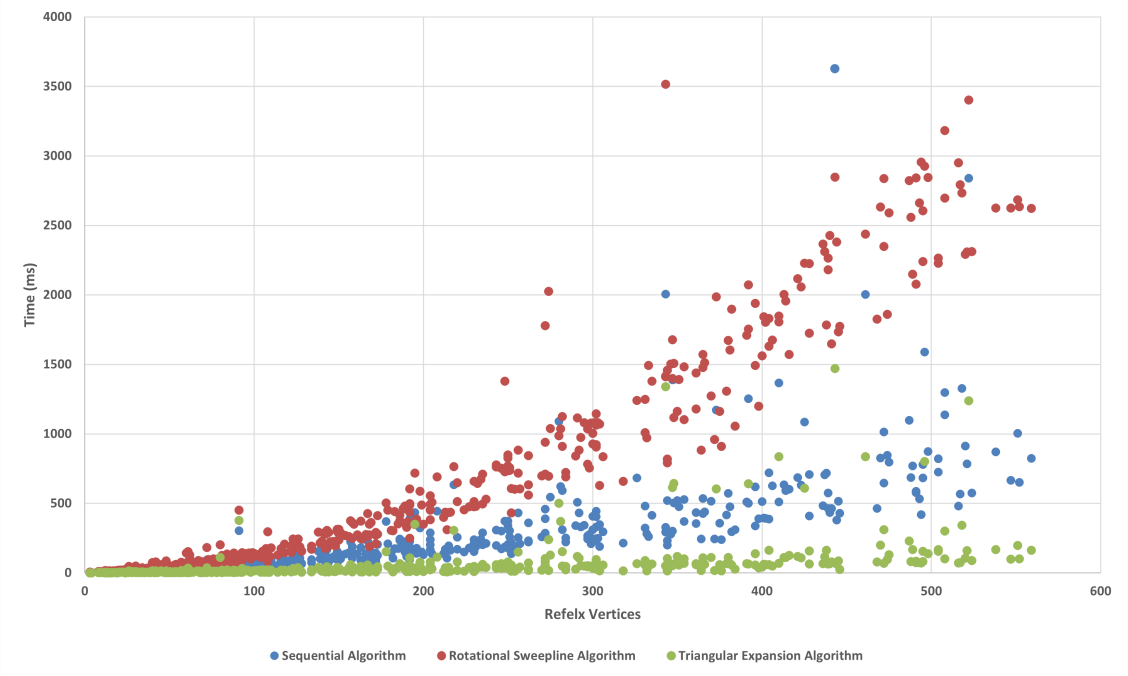




A.1.3 Reflex Vertices

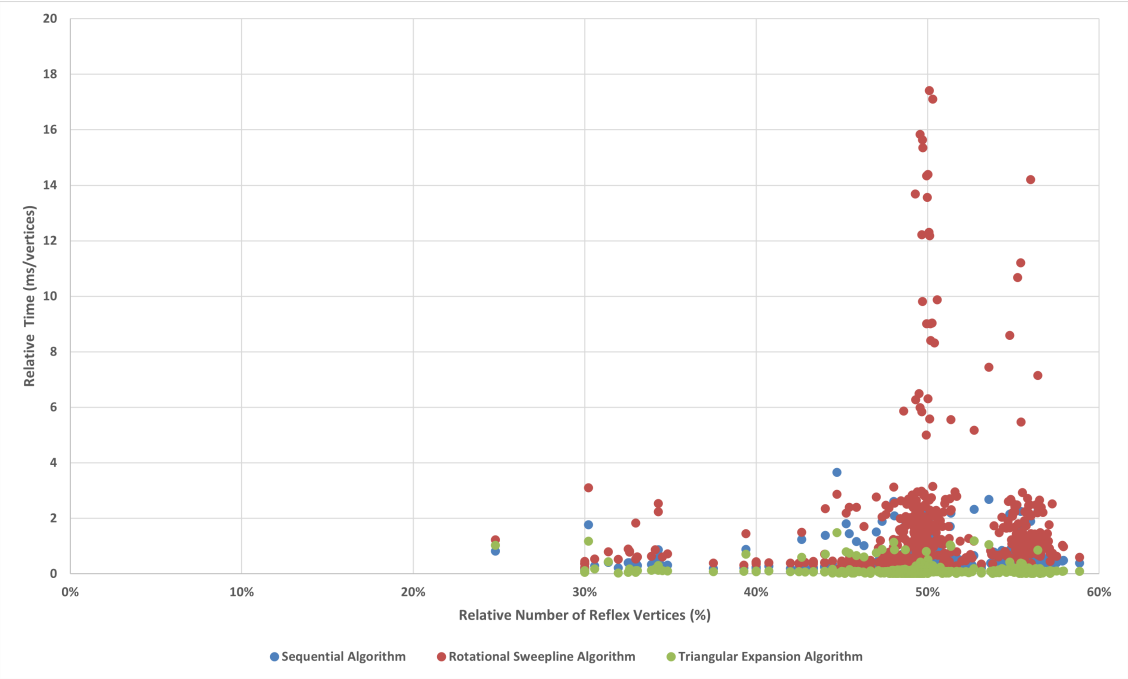
The number of reflex vertices for all polygons and for the polygons smaller than 1200 vertices.

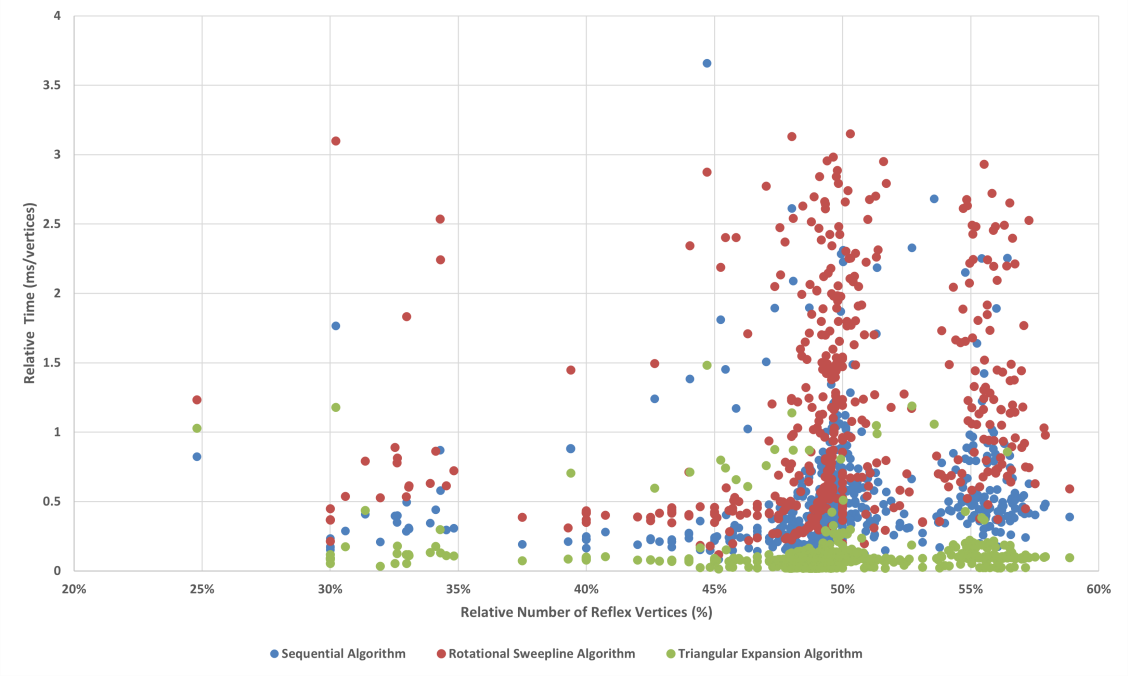




A.1.4 Normalized Reflex Vertices

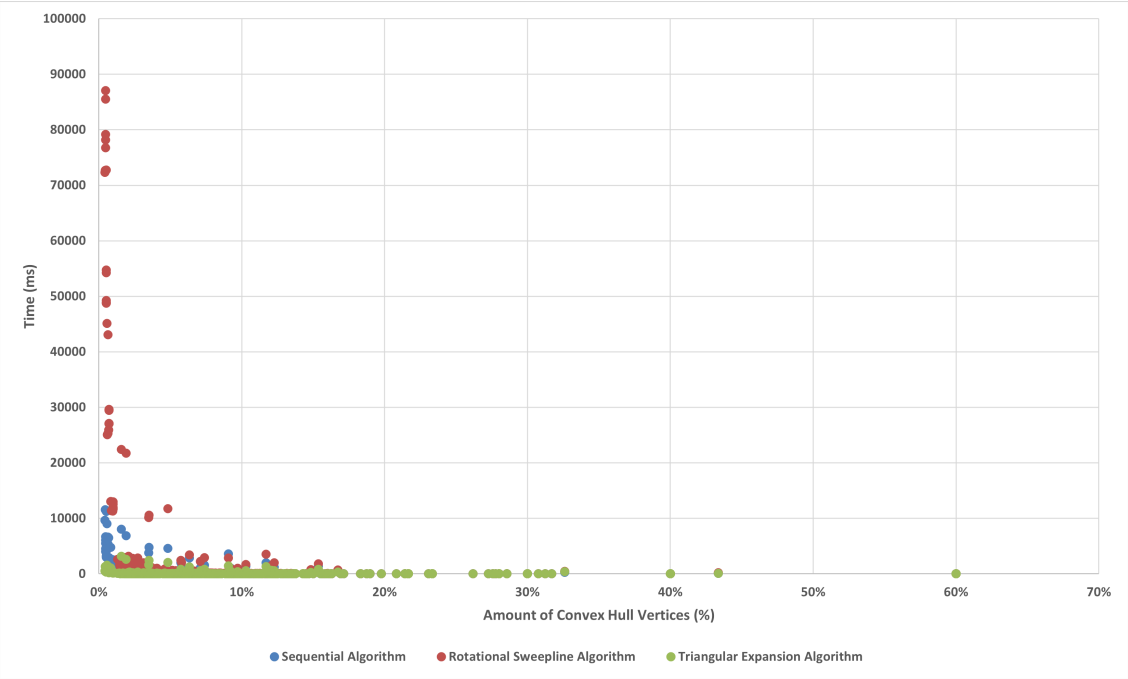
The relative number of reflex vertices for all polygons and without outliers.

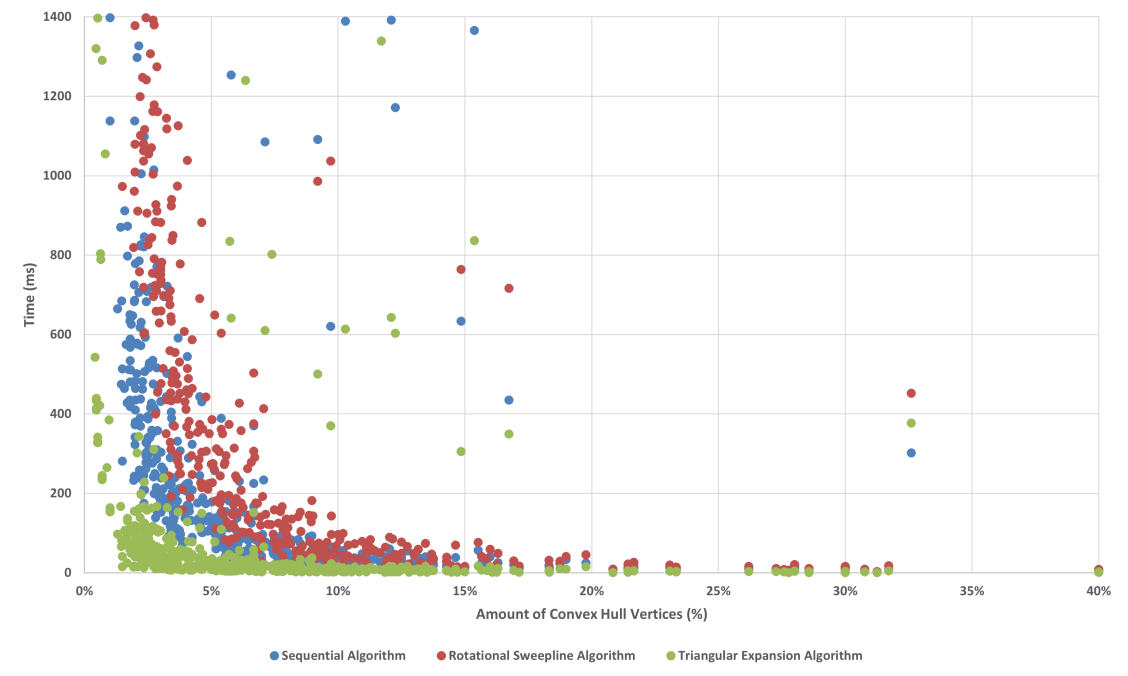




A.1.5 Polygon Convexity

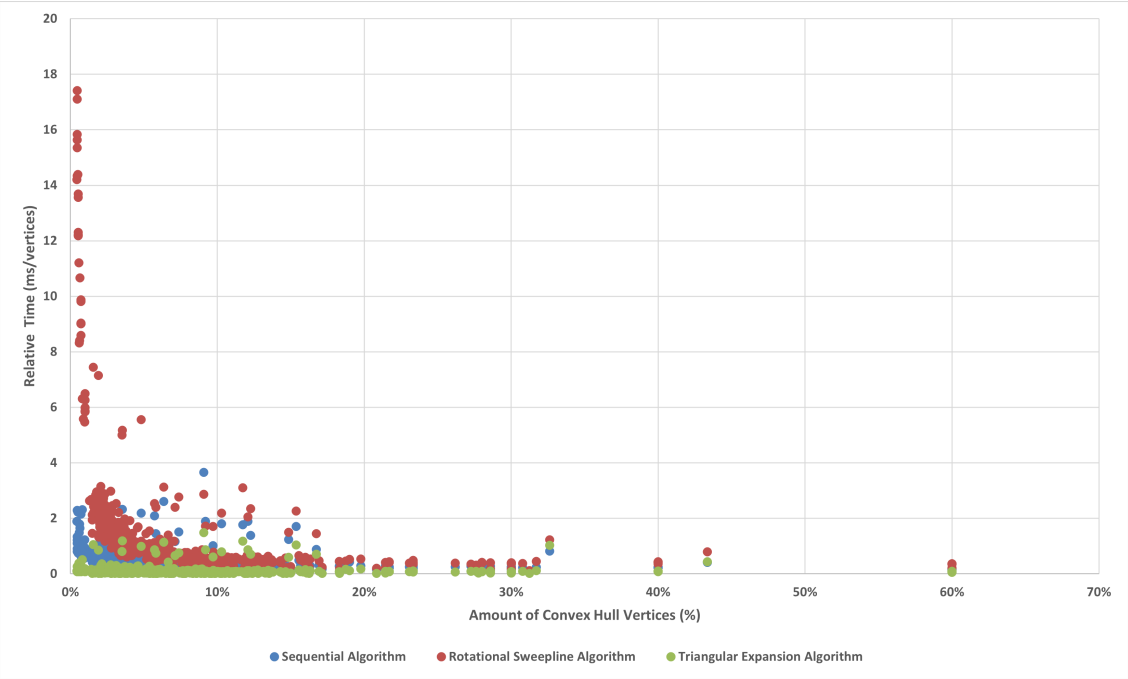
The amount of convex hull vertices for all polygons and without outliers.

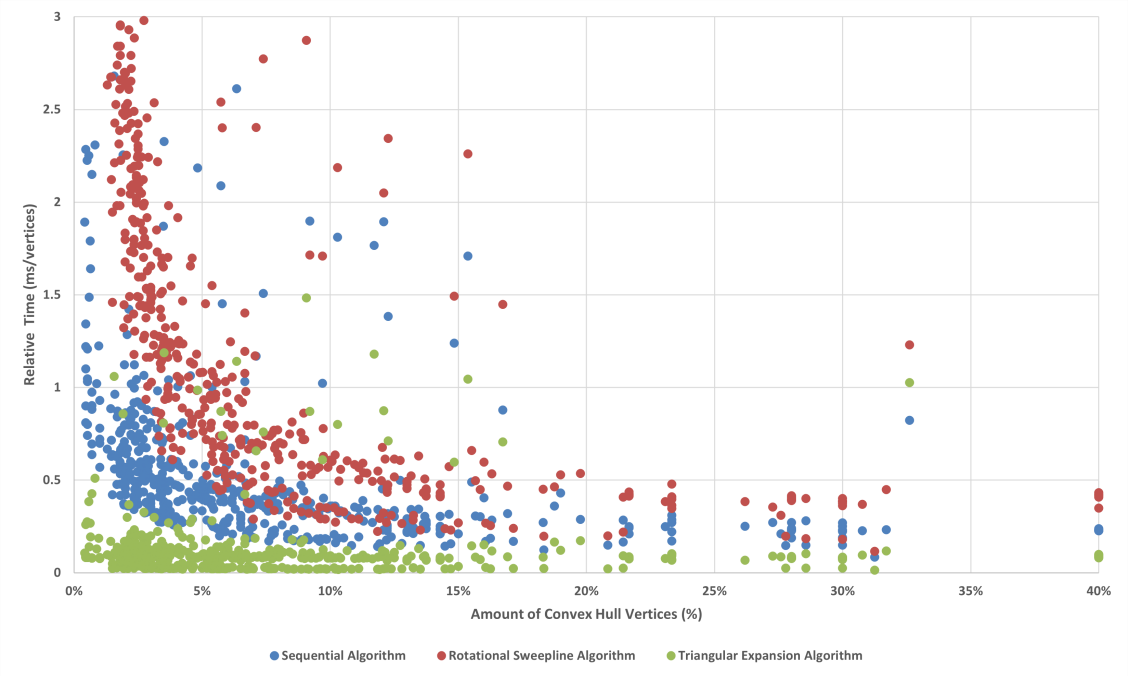




A.1.6 Normalized Polygon Convexity

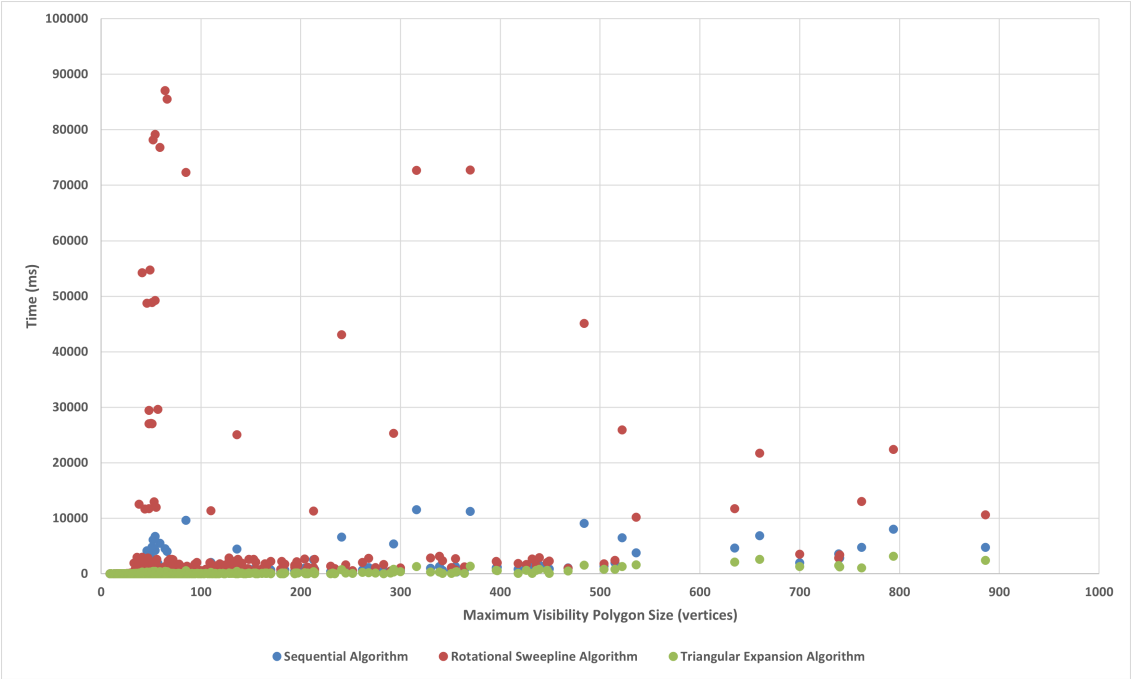
The relative amount of convex hull vertices for all polygons and without outliers.

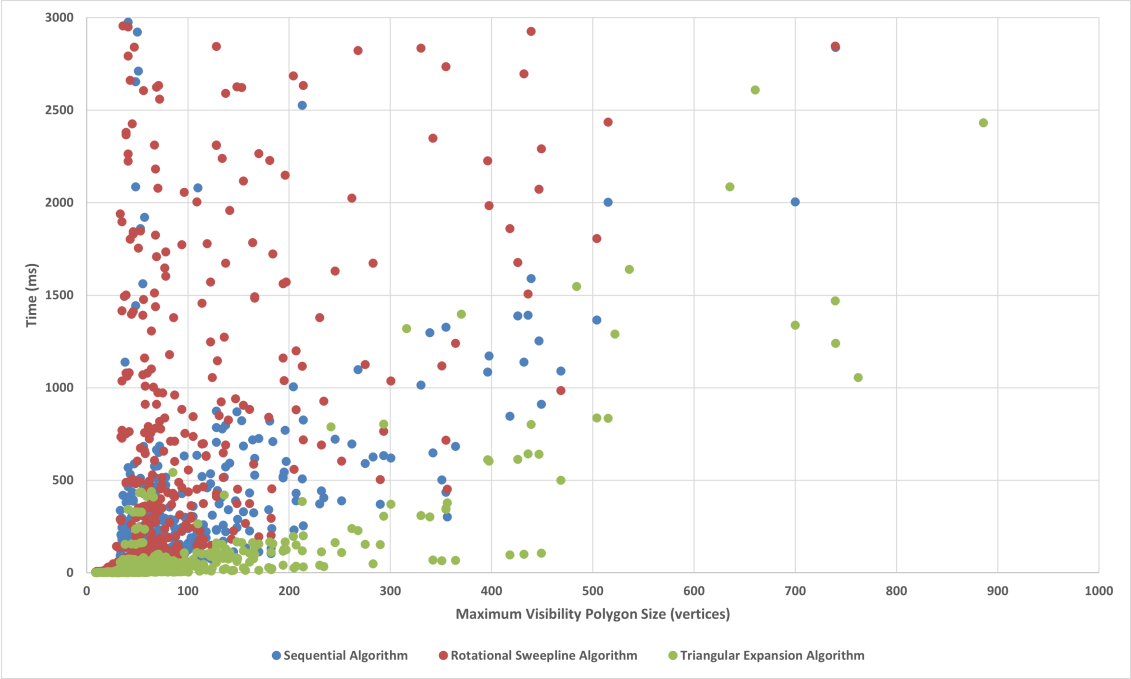


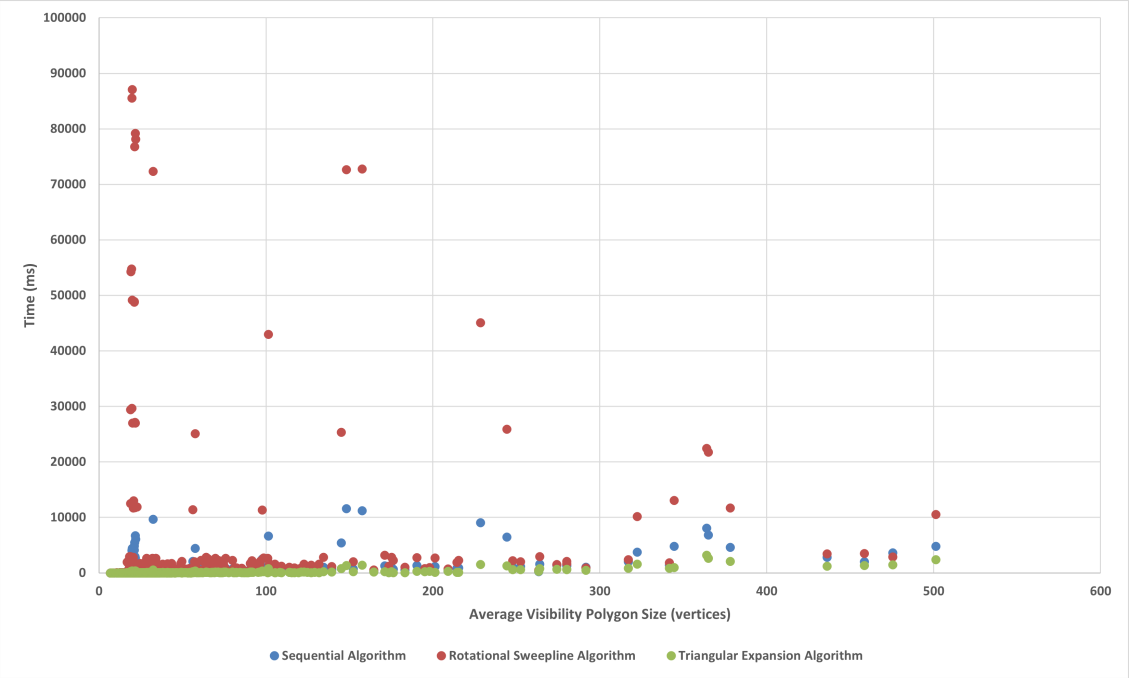


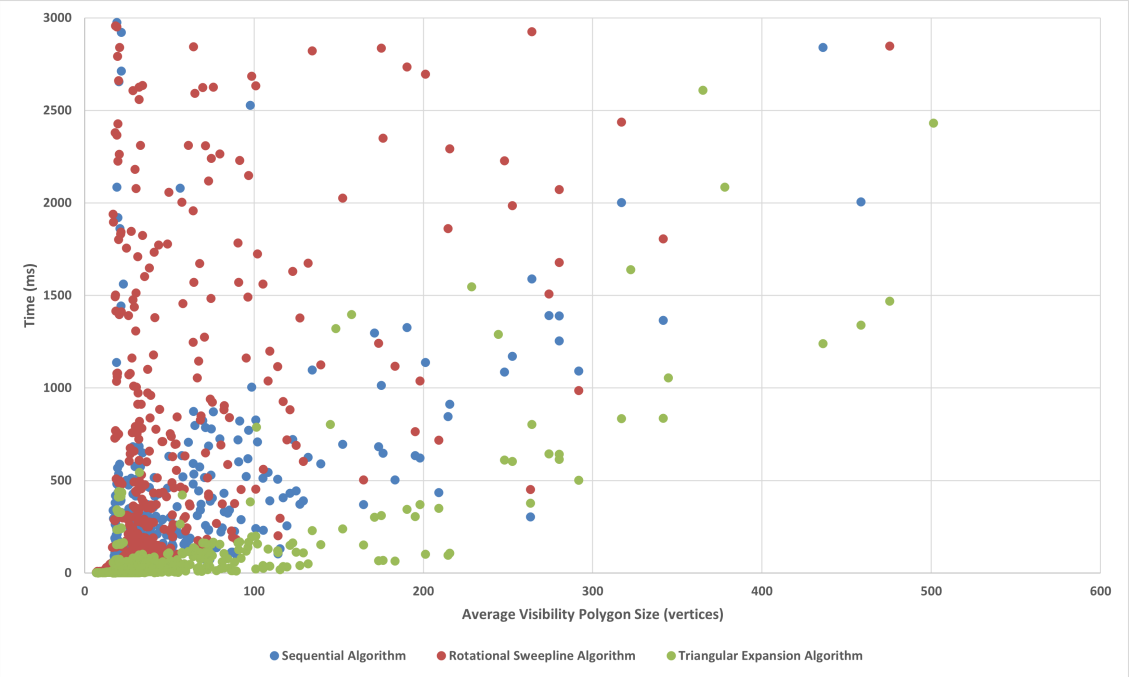
A.1.7 Visibility Polygon Size

The maximum visibility polygon size for all polygons and without outliers. Also the average visibility polygon size for all polygons and without outliers.



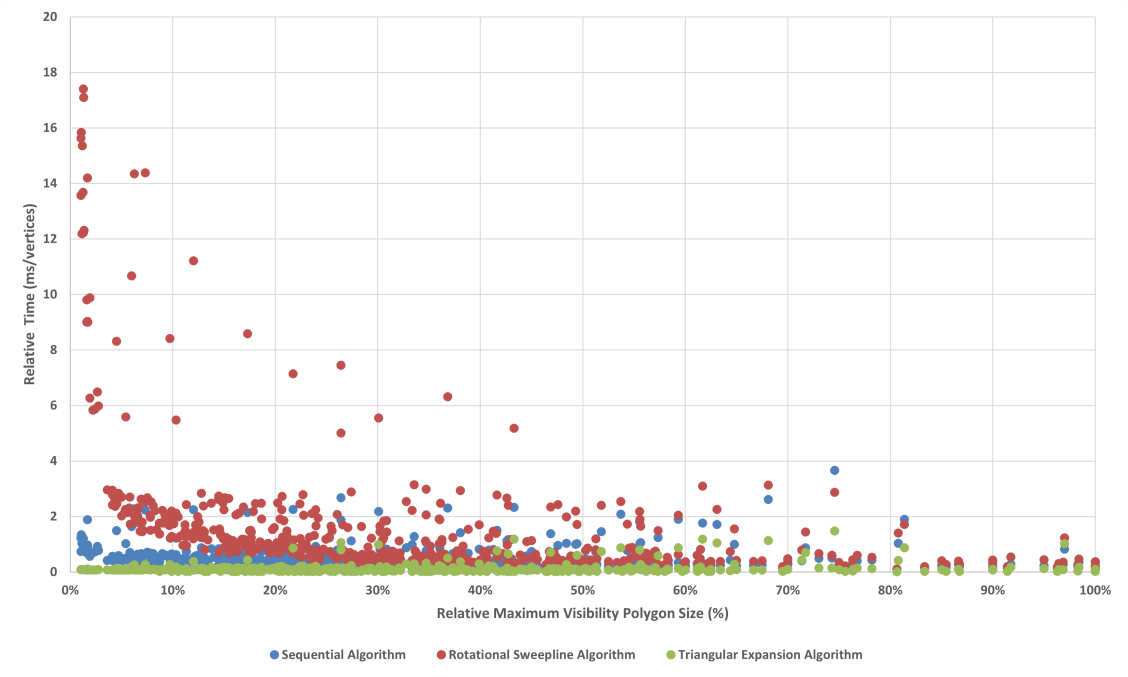


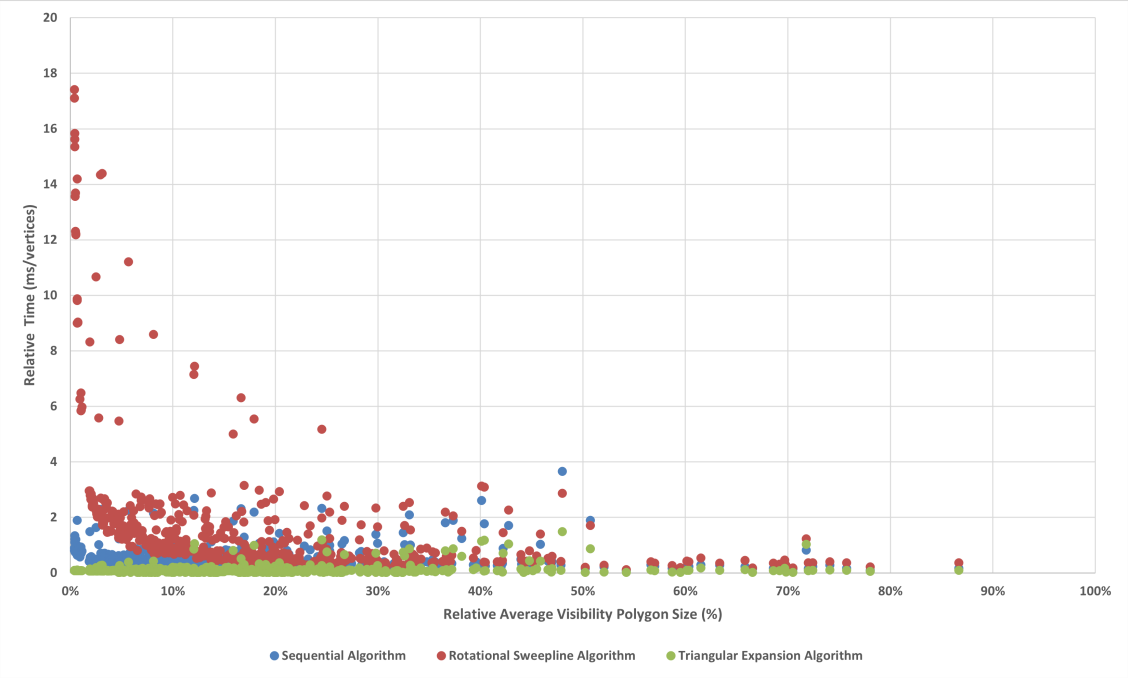

100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Time (ms)
0
100
200
300
400
500
600
Average Visibility Polygon Size (vertices)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

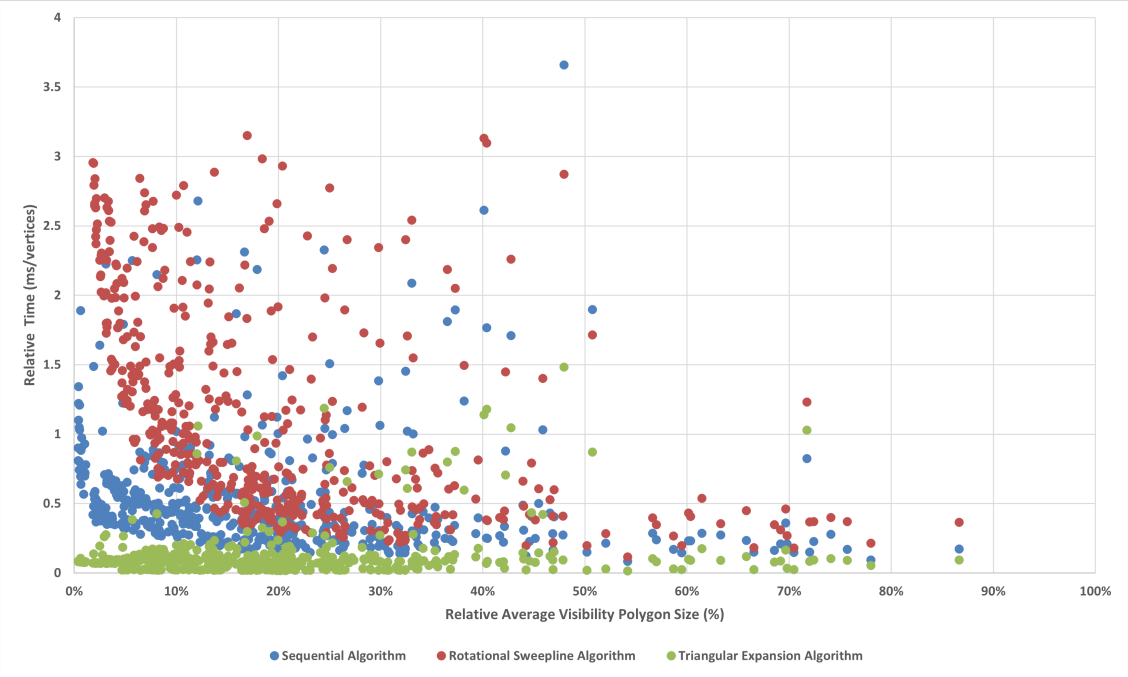

3000
2500
2000
1500
1000
500
0
Time (ms)
0
100
200
300
400
500
600
Average Visibility Polygon Size (vertices)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

A.1.8 Normalized Visibility Polygon Size

The relative maximum visibility polygon size for all polygons and without outliers. Also the relative average visibility polygon size for all polygons and without outliers.

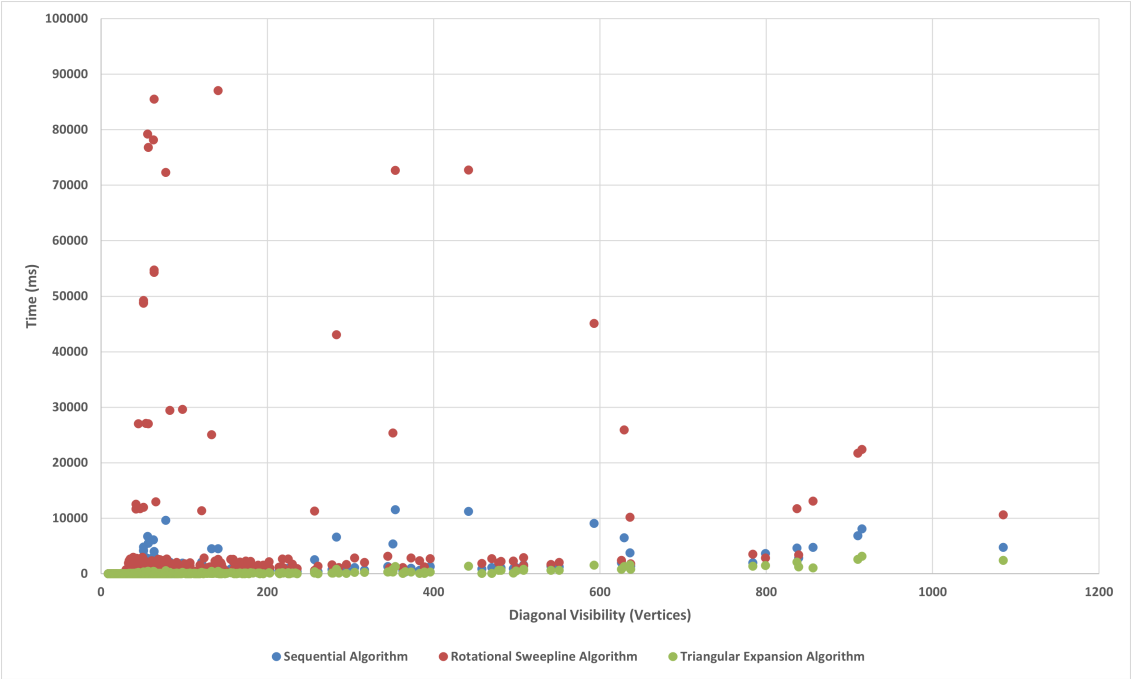


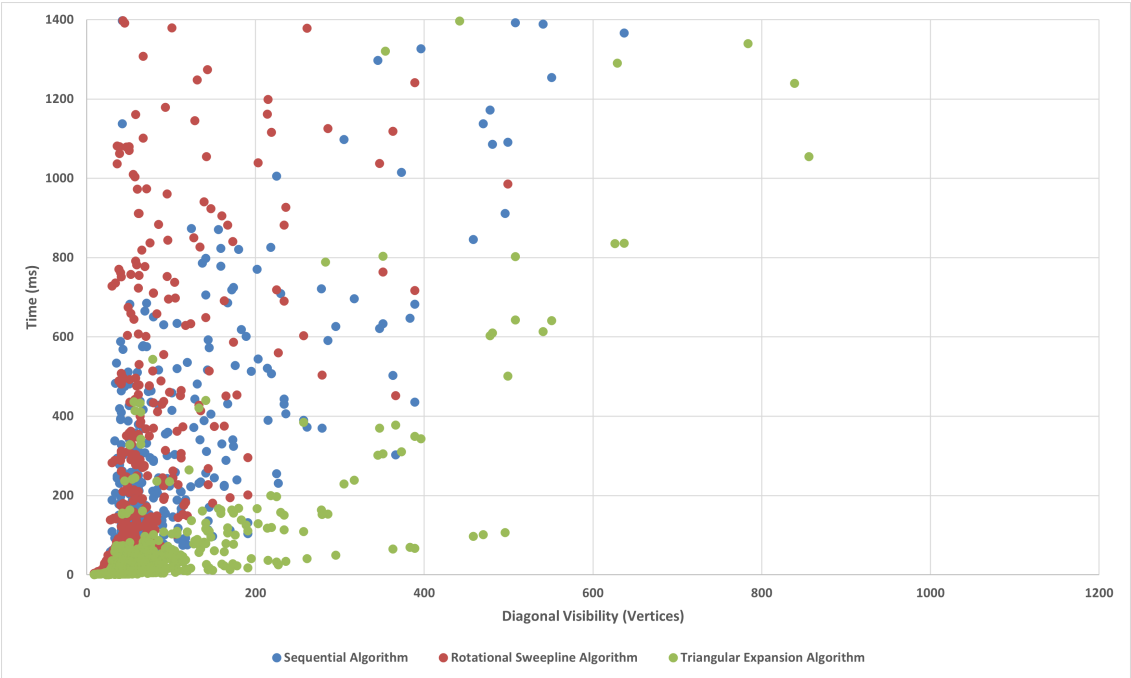

Relative Time (ms/vertices)
20
18
16
14
12
10
8
6
4
2
0
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Relative Average Visibility Polygon Size (%)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

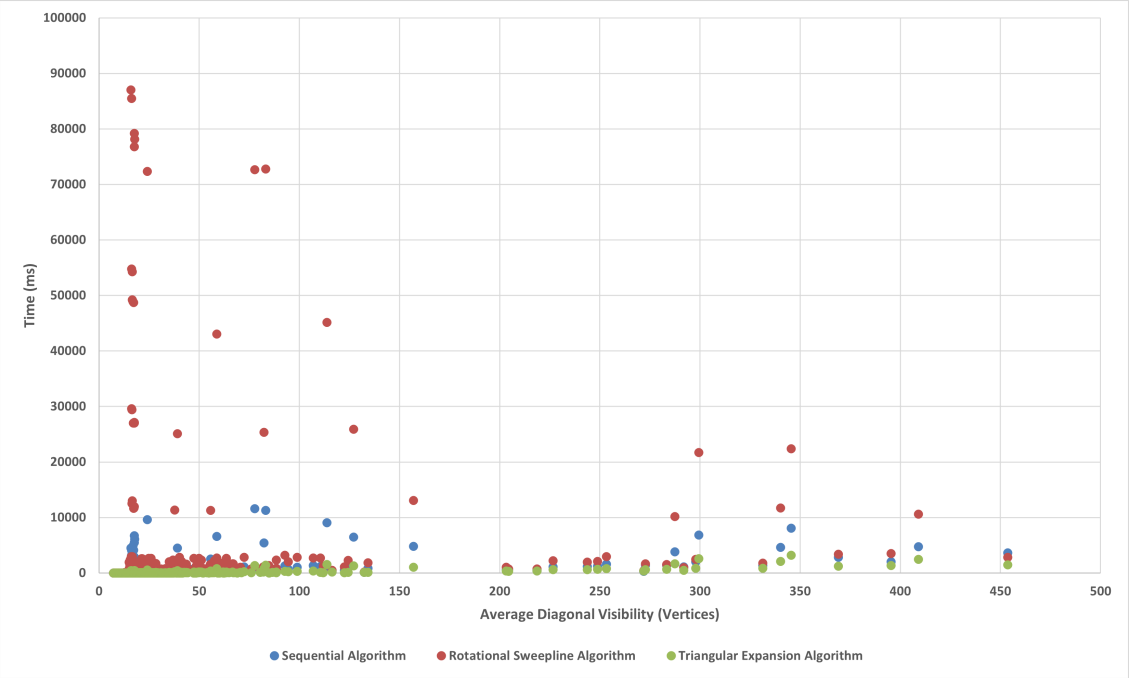

Relative Time (ms/vertices)
4
3.5
3
2.5
2
1.5
1
0.5
0
0%
10%
20%
30%
40%
50%
60%
70%
80%
90%
100%
Relative Average Visibility Polygon Size (%)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

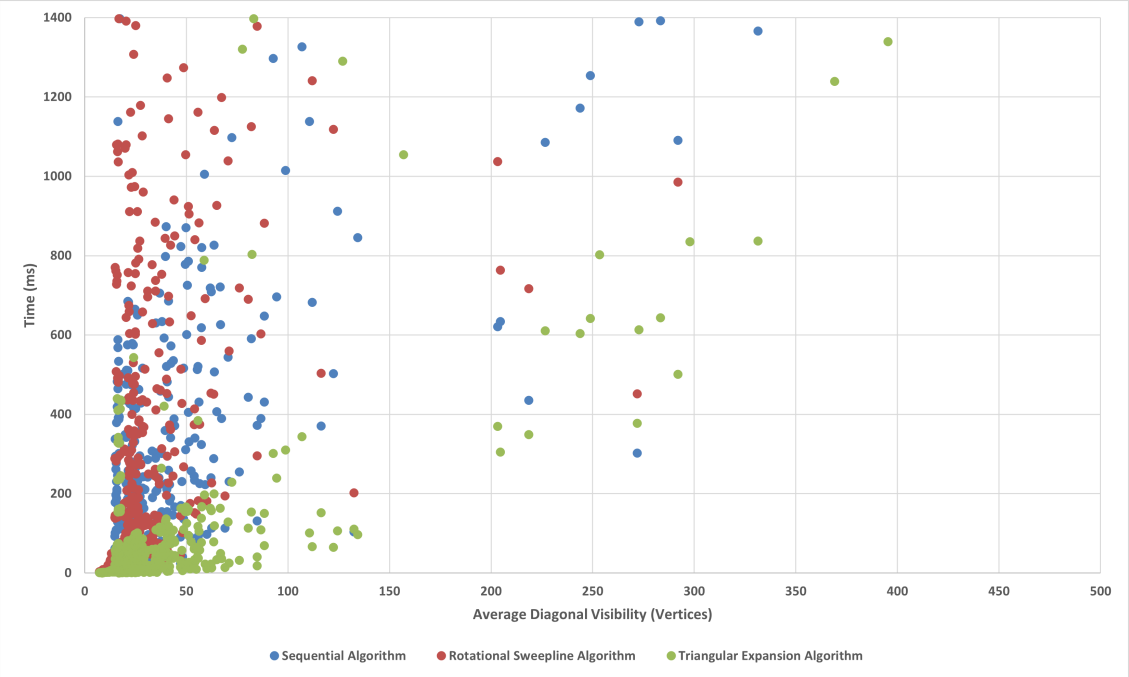
A.1.9 Diagonal Visibility

The diagonal visibility for all polygons and without outliers. Also the average diagonal visibility for all polygons and without outliers.



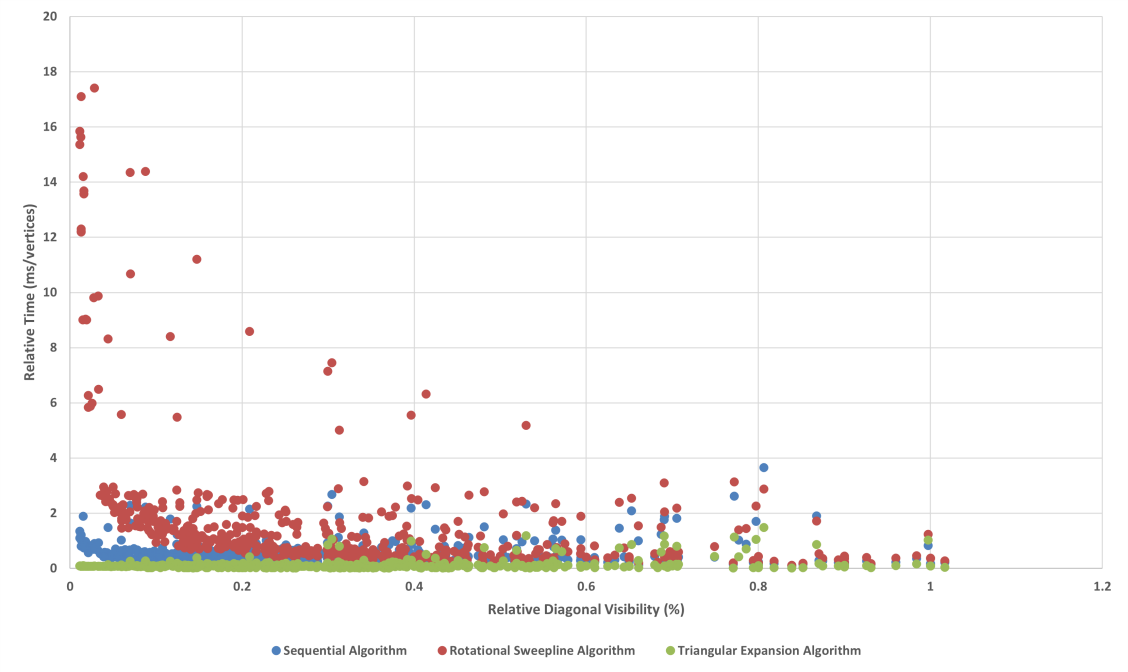


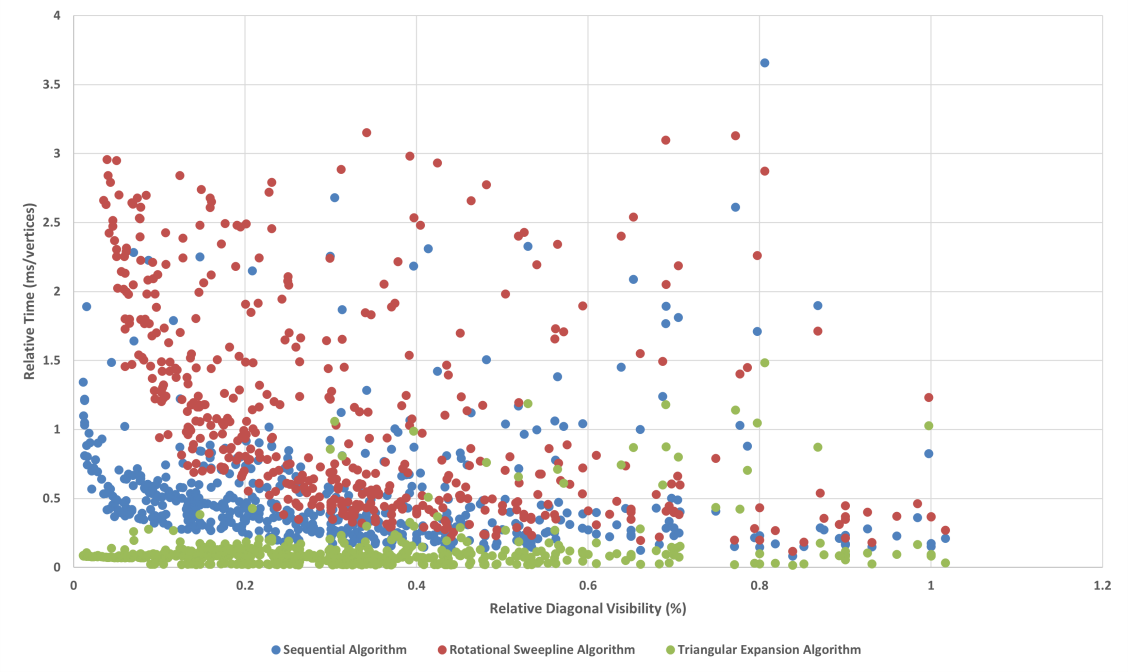

Time (ms)
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
0
50
100
150
200
250
300
350
400
450
500
Average Diagonal Visibility (Vertices)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

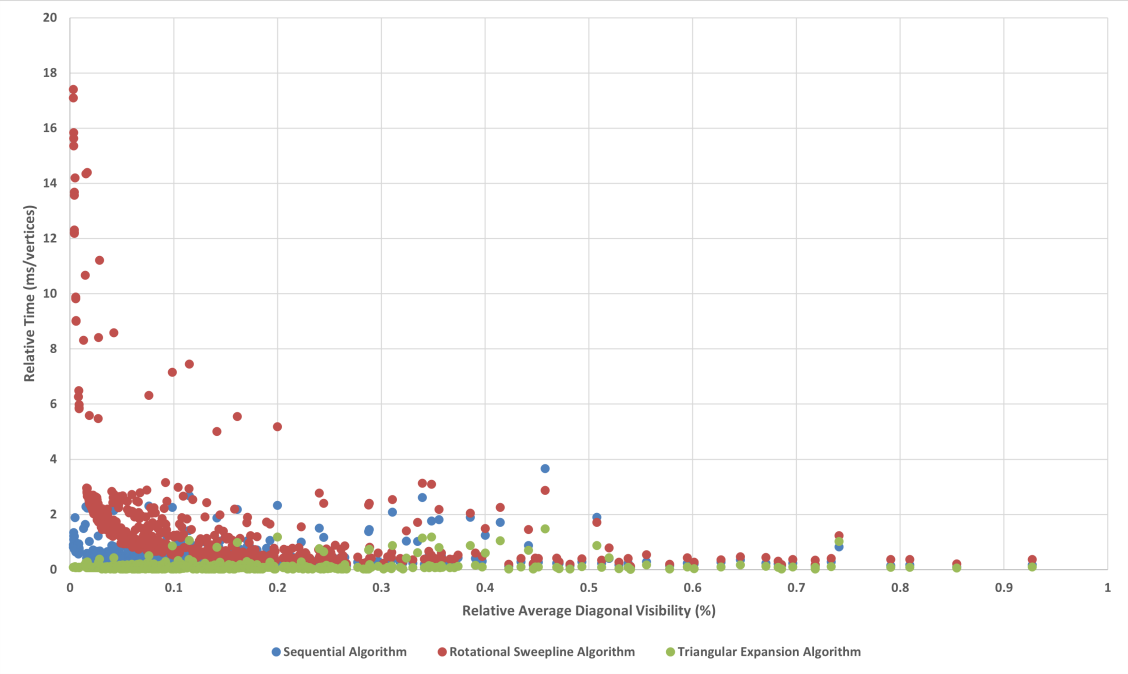

Time (ms)
1400
1200
1000
800
600
400
200
0
0
50
100
150
200
250
300
350
400
450
500
Average Diagonal Visibility (Vertices)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

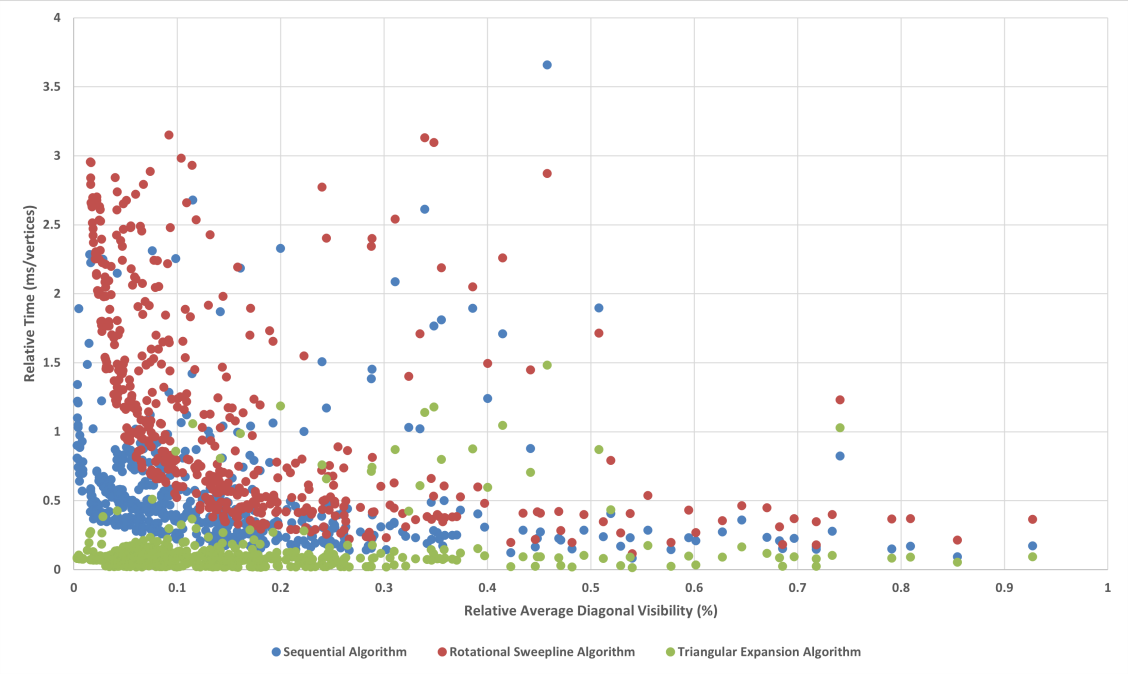
A.1.10 Normalized Diagonal Visibility

The relative diagonal visibility for all polygons and without outliers. Also the relative average diagonal visibility for all polygons and without outliers.





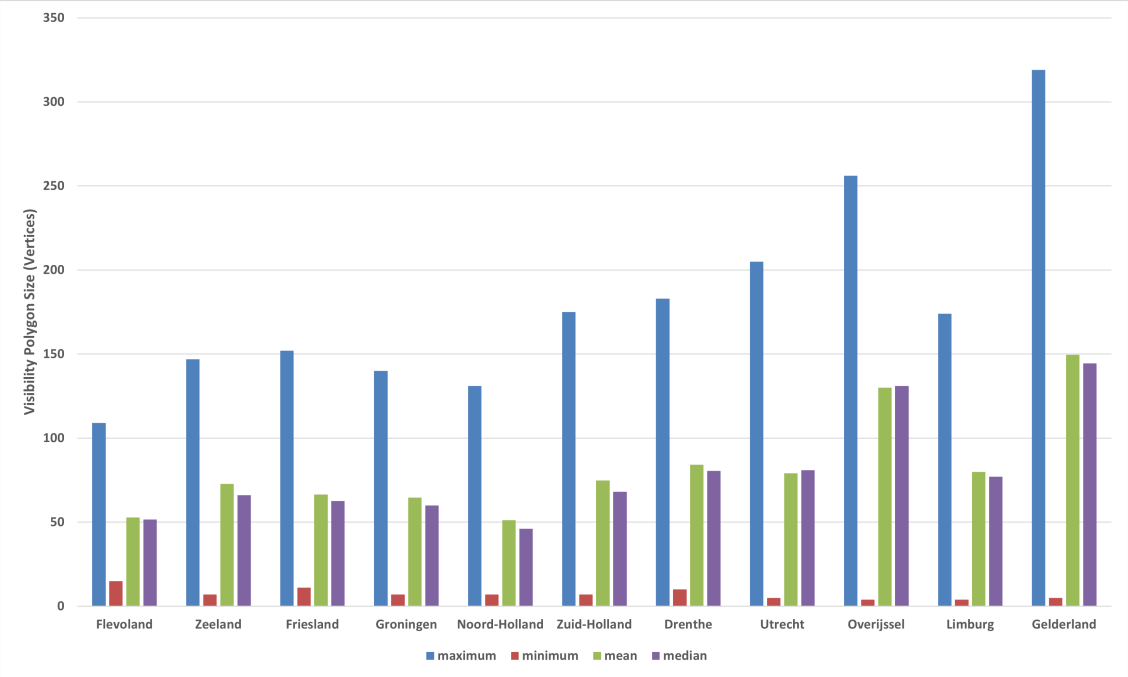

Relative Time (ms/vertices)
Relative Average Diagonal Visibility (%)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm


Relative Time (ms/vertices)
Relative Average Diagonal Visibility (%)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

A.2 Generated Polygons

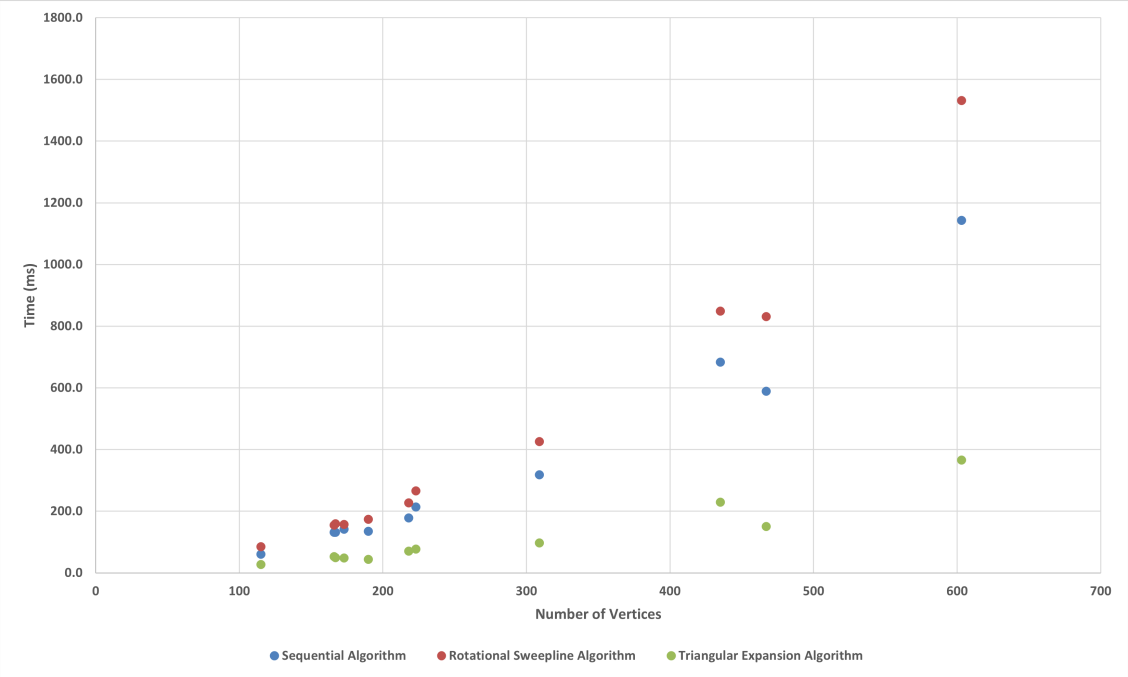
A.2.1 Visibility Polygon Size

The maximum, minimum, average and median visibility polygon size of all polygons.



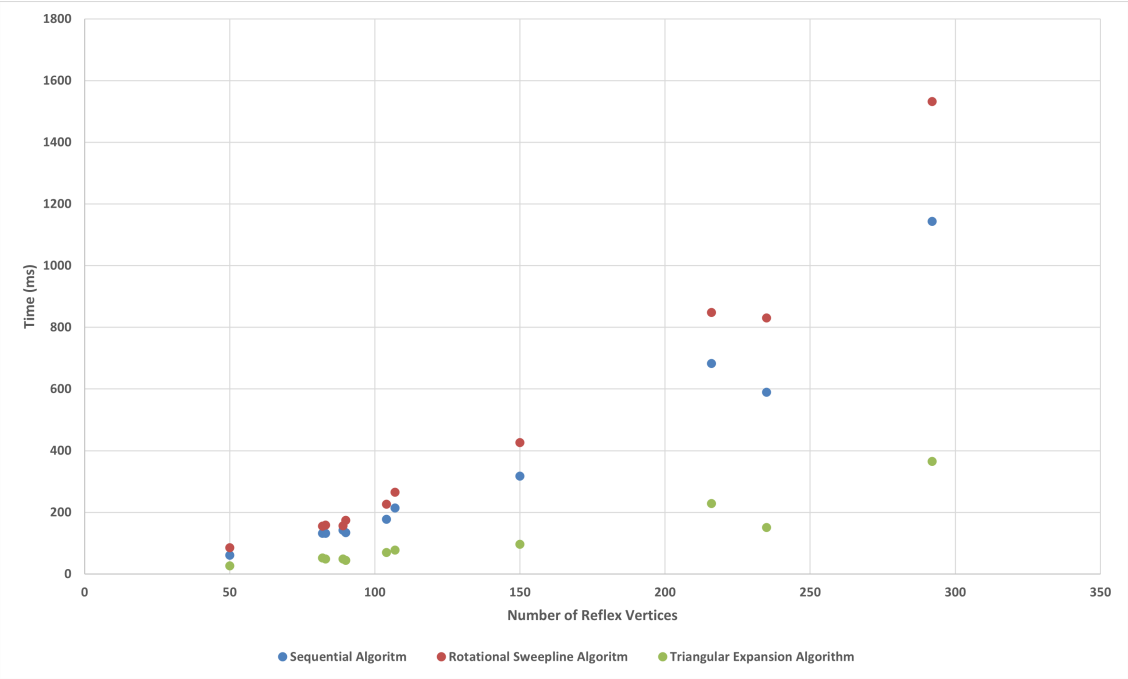
A.2.2 Computation Time

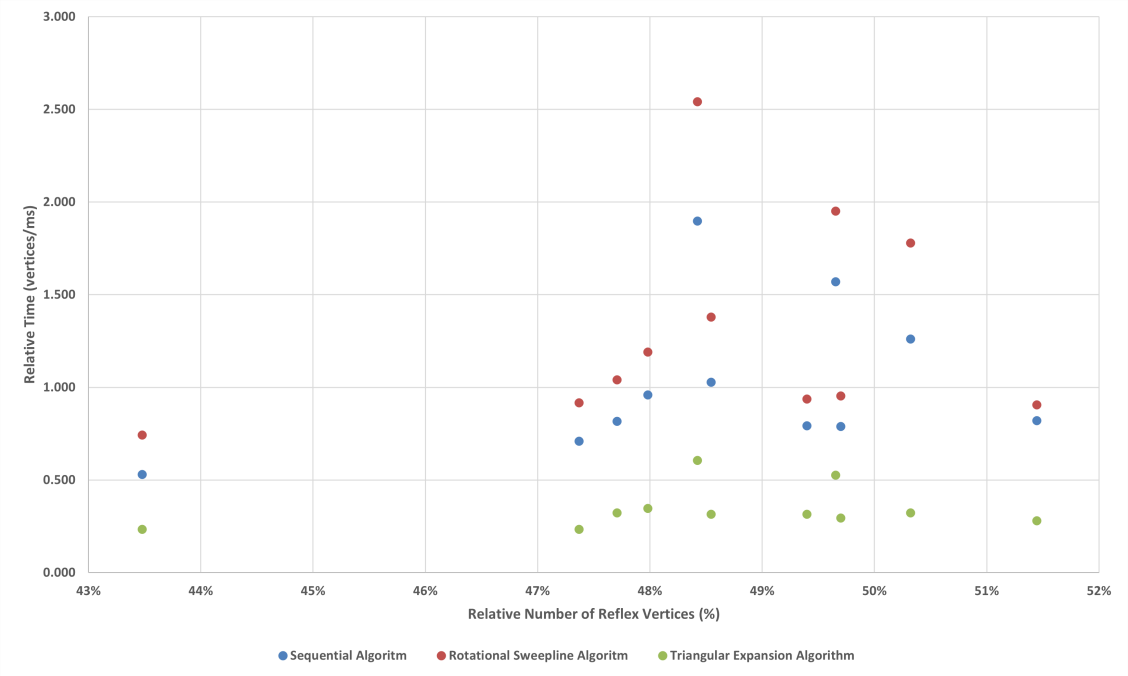
The computation times for all polygons.



A.2.3 Reflex Vertices

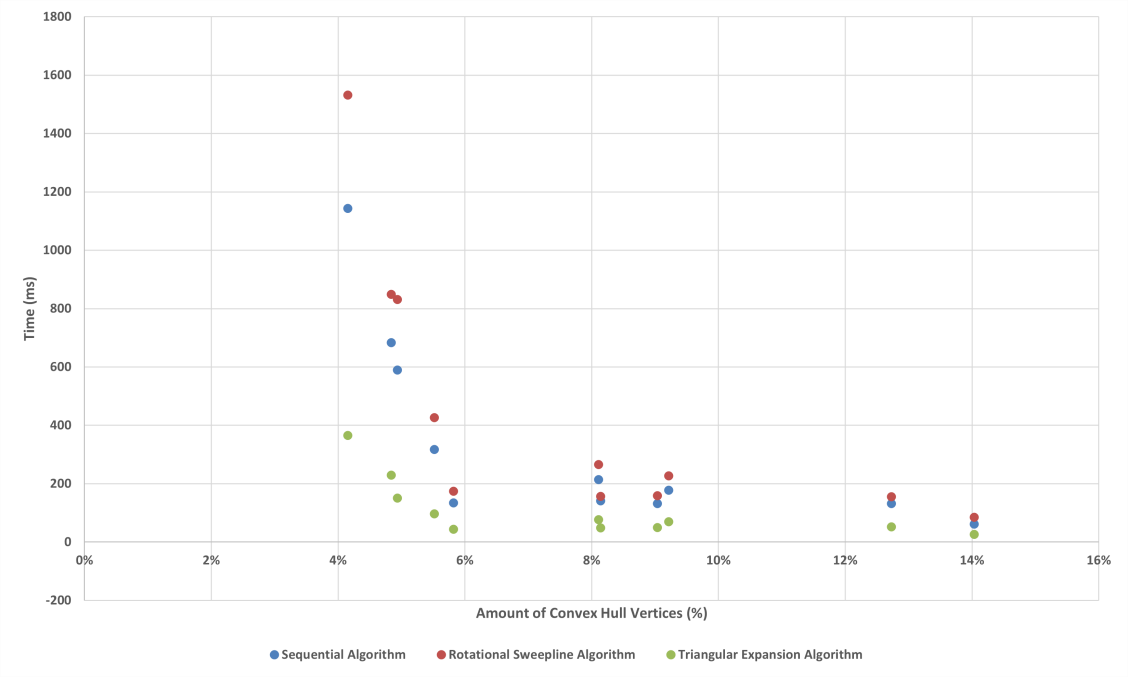
The number of reflex vertices for all polygons and the relative number of reflex vertices for all polygons.

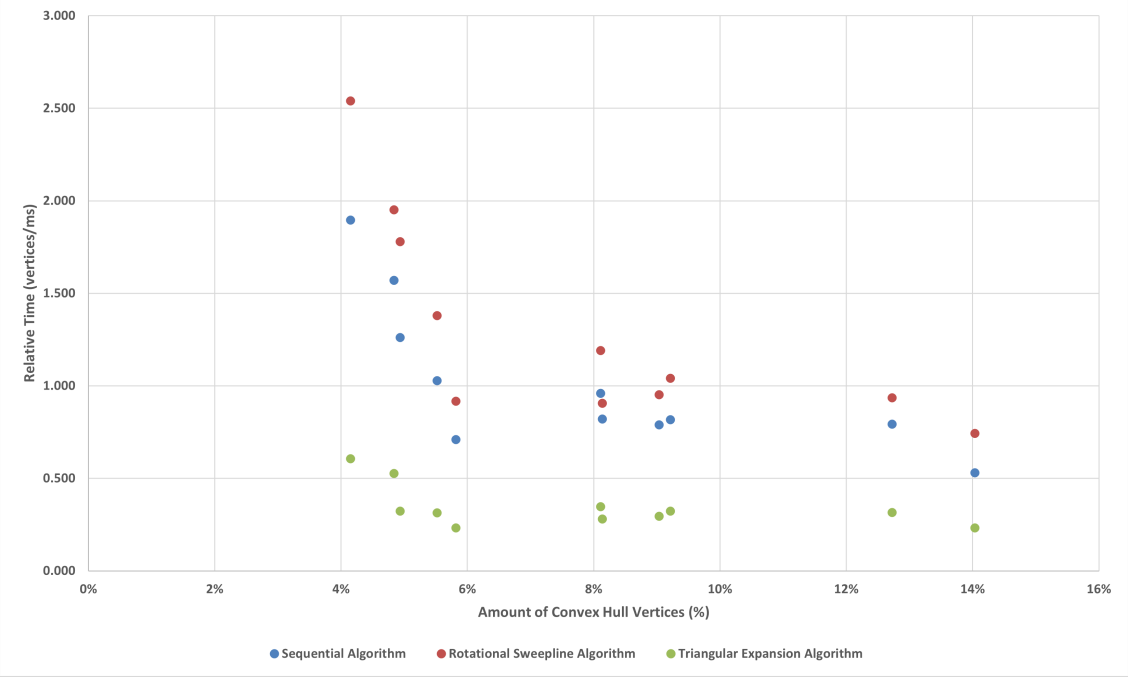




A.2.4 Polygon Convexity

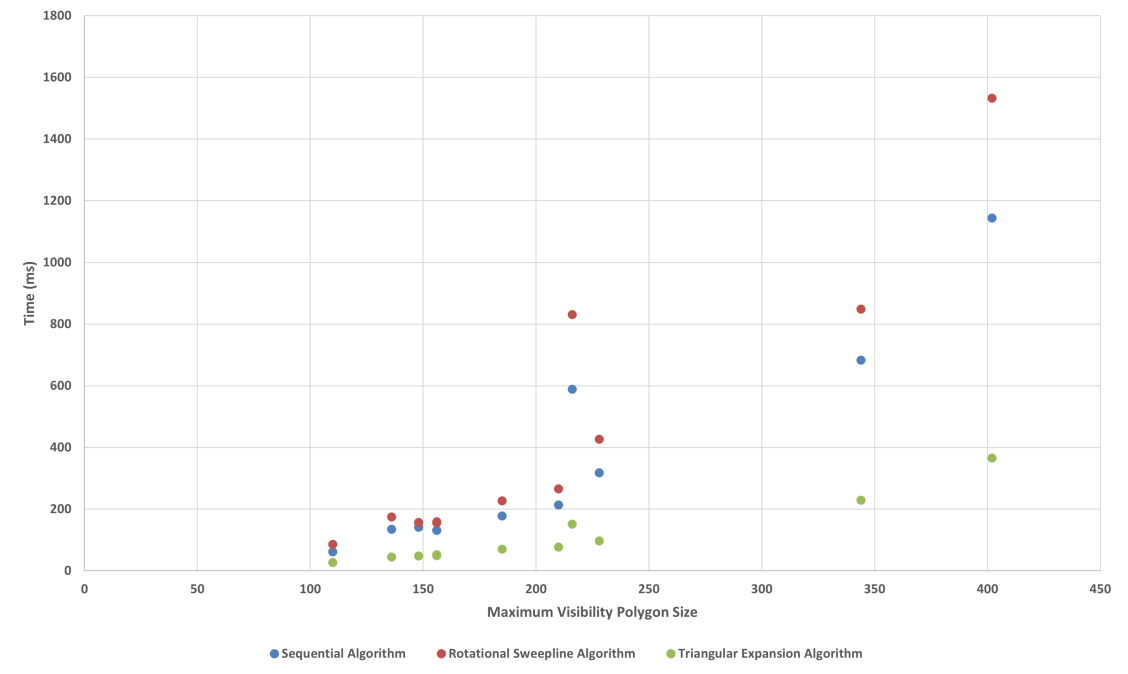
The amount of convex hull vertices vertices for all polygons and the relative amount of convex hull vertices for all polygons.

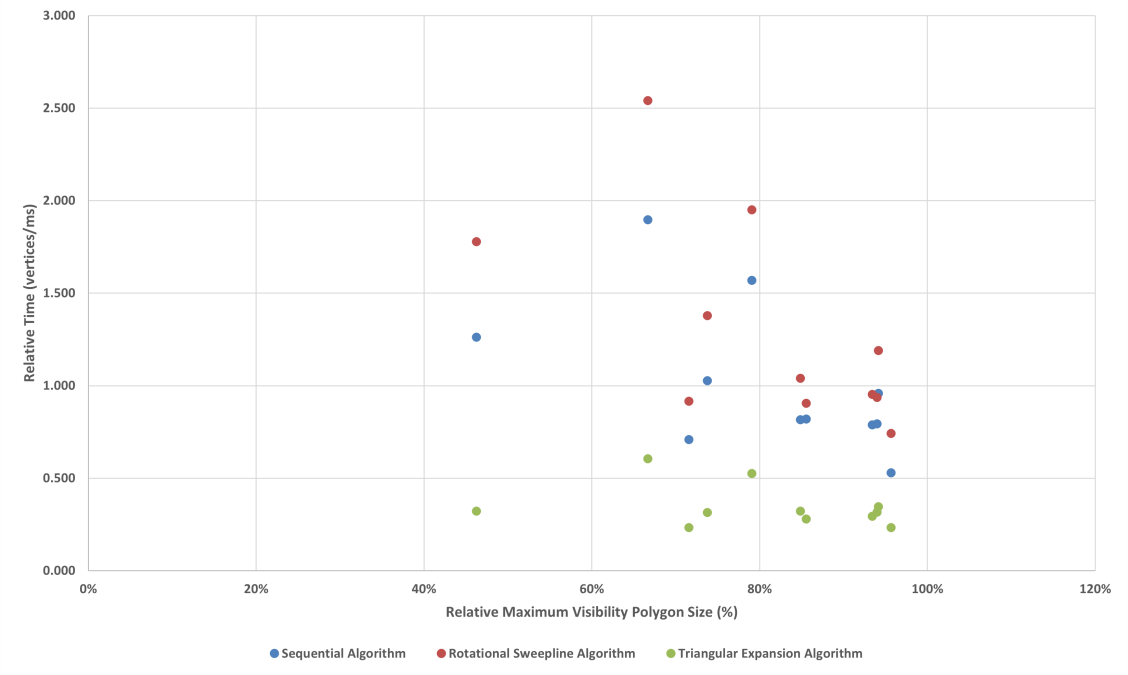


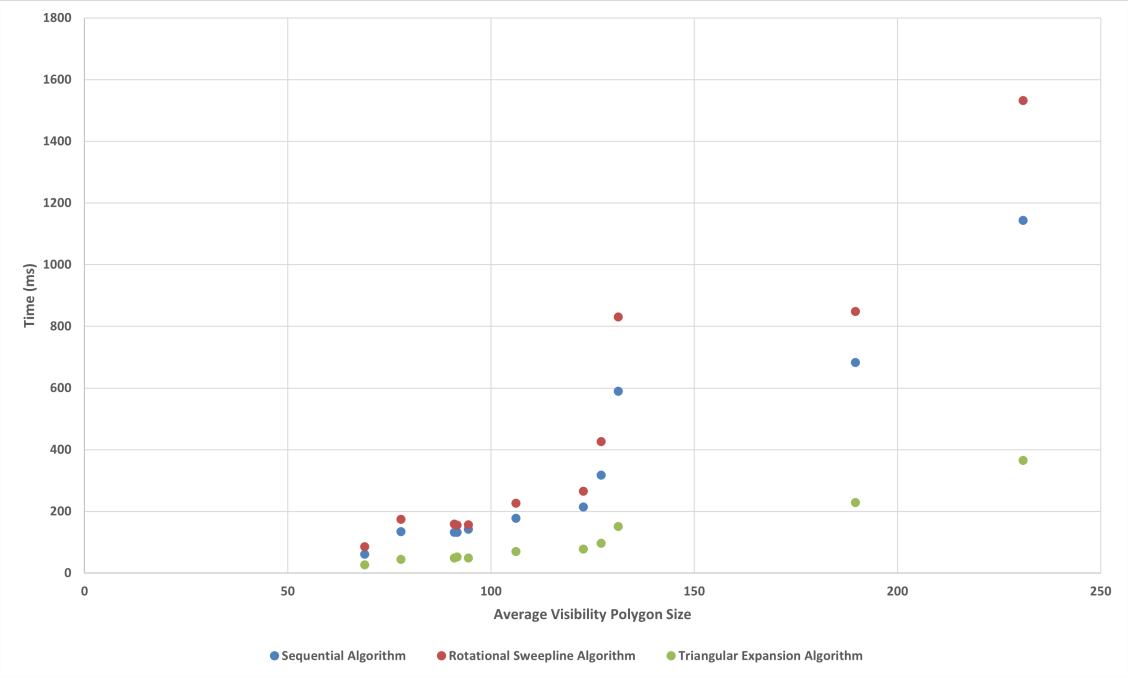


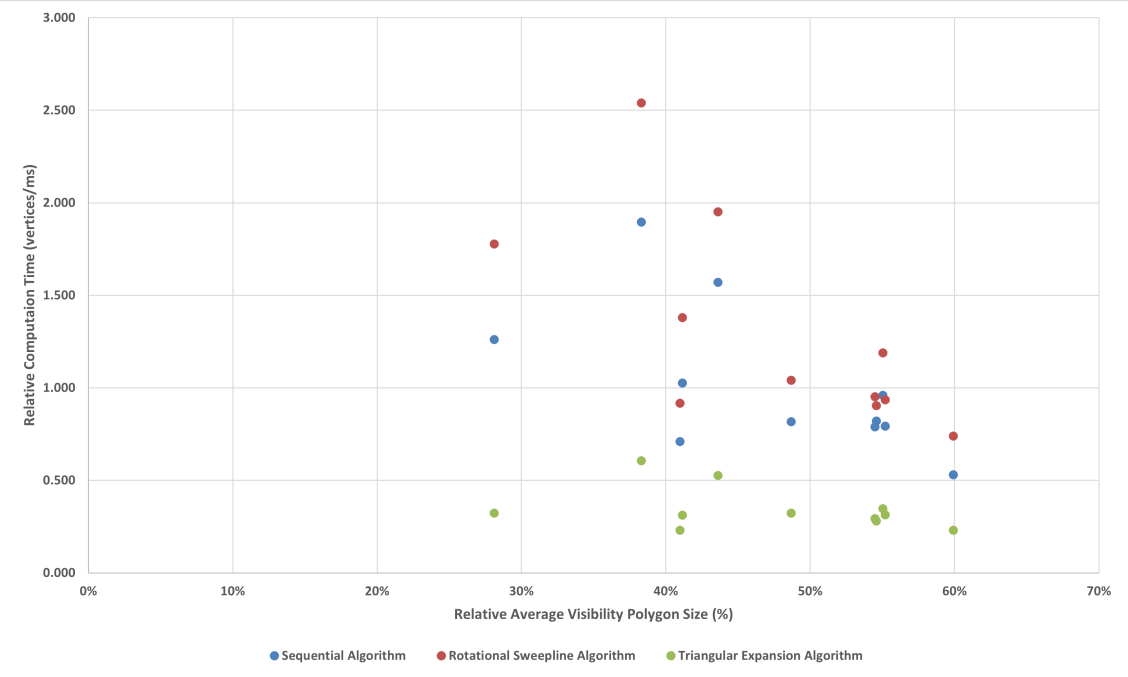
A.2.5 Visibility Polygon Size

The maximum visibility polygon size for all polygons and the relative maximum visibility polygon size for all polygons. Also the average visibility polygon size for all polygons and the relative average visibility polygon size for all polygons.



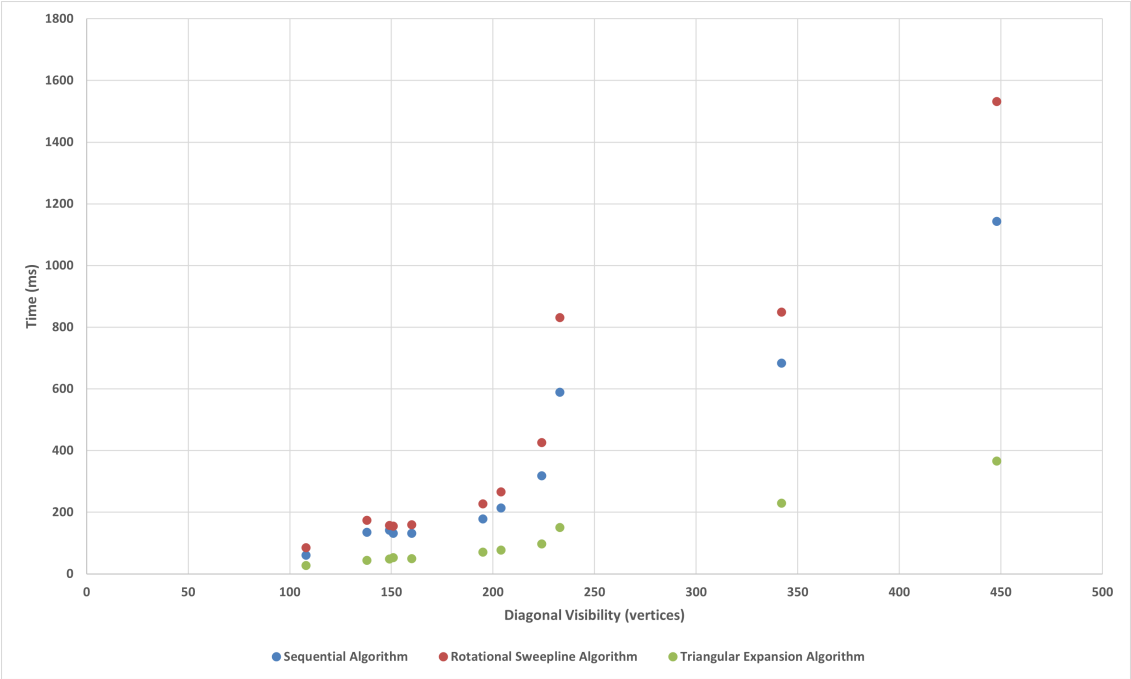


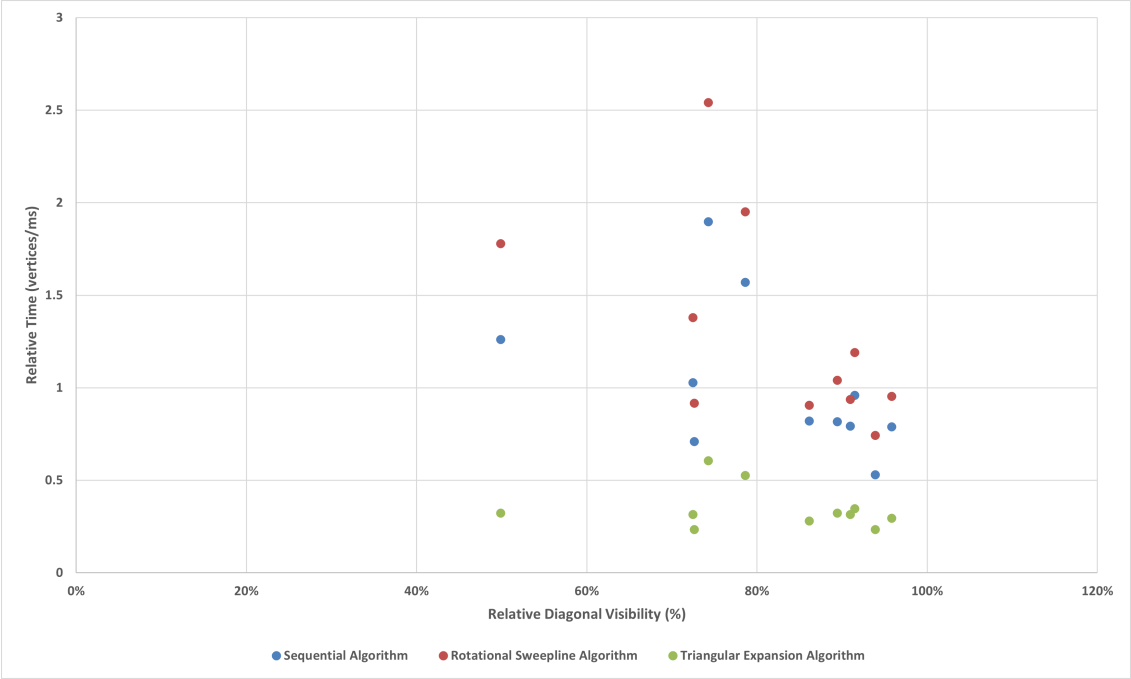

1800
1600
1400
1200
1000
800
600
400
200
0
Time (ms)
0
50
100
150
200
250
Average Visibility Polygon Size
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

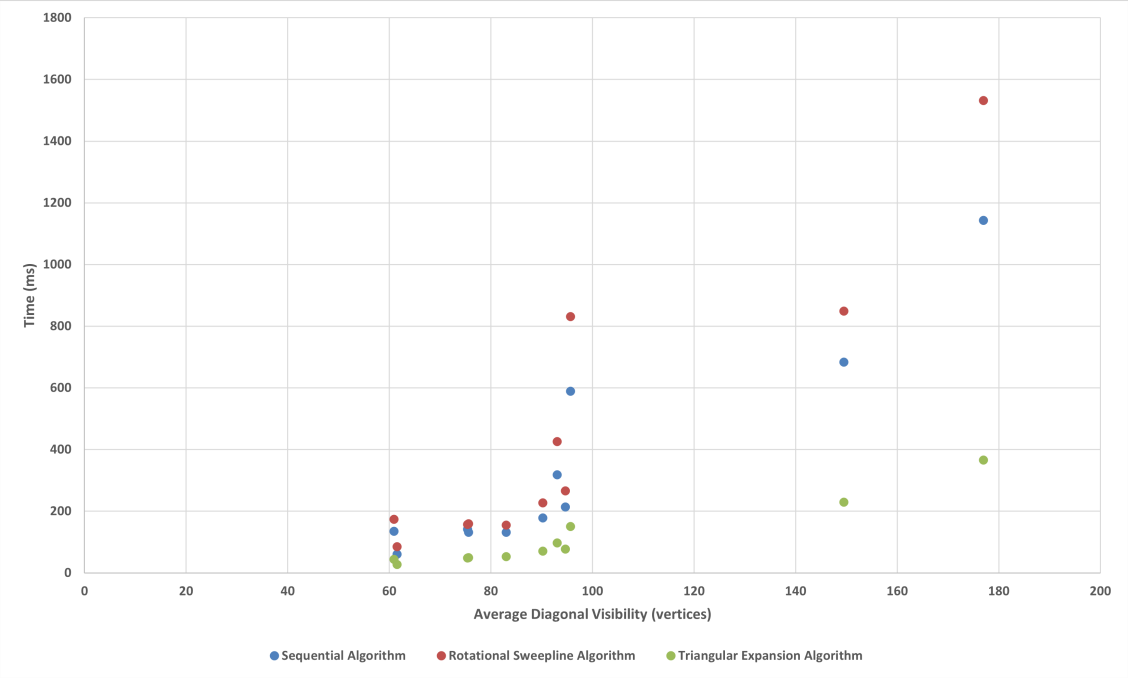

3.000
2.500
2.000
1.500
1.000
0.500
0.000
Relative Computaion Time (vertices/ms)
0%
10%
20%
30%
40%
50%
60%
70%
Relative Average Visibility Polygon Size (%)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm

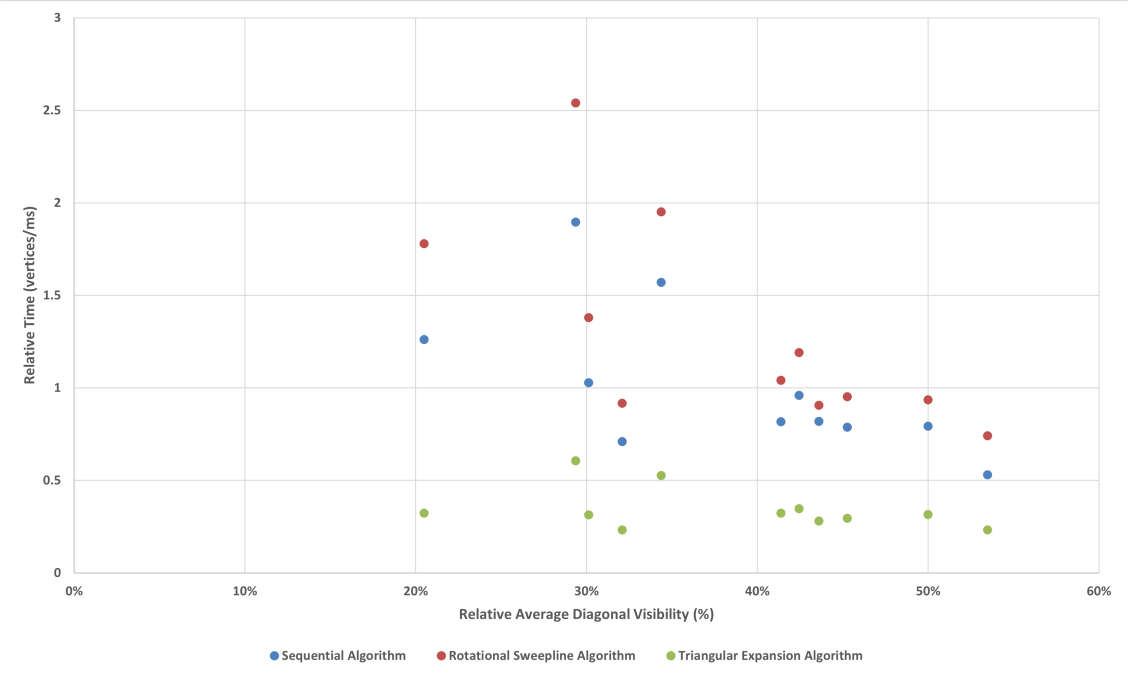
A.2.6 Diagonal Visibility

The diagonal visibility for all polygons and the relative diagonal visibility for all polygons. Also the average diagonal visibility for all polygons and the relative average diagonal visibility for all polygons.






Time (ms)
1800
1600
1400
1200
1000
800
600
400
200
0
0
20
40
60
80
100
120
140
160
180
200
Average Diagonal Visibility (vertices)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm


Relative Time (vertices/ms)
3
2.5
2
1.5
1
0.5
0
0%
10%
20%
30%
40%
50%
60%
Relative Average Diagonal Visibility (%)
Sequential Algorithm
Rotational Sweepline Algorithm
Triangular Expansion Algorithm