Using GiST-Indexes to Speed up Visibility Calculations

Abstract

Given a user's GPS position, it is often useful to calculate which buildings (or objects) the user is able to see. This gives a candidate set of landmarks that can be used in navigation instructions (e.g. "go left, just past the fountain 150 meters in front of you") or as the subject of information presentation (e.g. "The neo-classical building to your right, was built in 1898 by the famous architect...").

In a server-based setting, where a system is tracking many simultaneous users, traditional ray casting techniques are too expensive and error prone. In this thesis we develop an indexing technique that enables the quick calculation of exactly which objects are visible from a given position in the 2D plane. The approach is to build polygons representing the 2D isovist of each polygon in the scene.

Then the problem of determining whether a building is visible from a given position, reduces to whether the position is within the $IsoVist$ polygon of the building. Such queries may then be rapidly answered at run time using a GiST based index. We develop variations of this solution and test them under various assumptions and present results.

Contents

List of Figures

List of Algorithms

List of Tables

List of Acronyms and Abbreviations

2D Two Dimensional Space

API Application Programming Interface

GPS Global Positioning System

GiST Generalized Search Tree

IDE Integrated Development Environment

JTS Java Topology Suite

JDBC Java Database Connectivity

KTH Royal Institute of Technology

LBS Location Based Services

OSM Open Street Map

DBMS Database Management System

PostgreSQL Open Source object-relational database system

PostGIS Spatial and Geographic Objects for PostgreSQL

SQL Structured Query Language

XML Extensible Markup Language

Chapter 1

Introduction

1.1 Background

The first algorithms for computing visibility were studied in the field of computer graphics during the late 60's for rendering visible scenes to the observer [1]. Since then, visibility computation has been useful in wireless network design, architectural design and, nowadays, Location Based Services (LBS). For example, in a pedestrian navigation [2] and exploration system it is important to determine if a user can actually see a building (or other objects).

A key concept from architecture is the notion of an *isovist*. Traditionally an isovist, in Two Dimensional Space (2D), is defined as the set of points on the plane that are visible from a given point [3]. A related notion developed in computational geometry is the *visibility polygon* which describes which parts of a polygon can be seen from a given point inside the polygon [4, 5, 6].

The notions of visibility, especially in the 2D plane, have been well explored. Fewer advancements have been made towards efficiently solving as many visibility queries as possible where the point of visibility is a real valued point determined at run time and the arrangements of polygons are held static. As a practical matter, usually we are interested in what objects can be seen from a given point, (e.g a pedestrian's position), not the infinite set of points defining the isovist.

A simple way to define which objects can be seen from a given position is using *ray casting* (or *ray shooting*) approaches [8, 9, 10, 11]. The idea is obvious, around the pedestrian's position shoot a series of rays. The first polygon (if any) that the ray intersects, is one of the visible objects. There are two parameters that are used to control the ray casting algorithm: (a) the delta angle which determines the granularity of the sampling (e.g. every 2° for a total of 180 rays) and (b) the max distance, which determines how long the rays are. These parameters trade-off efficiency with accuracy. Figure 1.1 shows a ray casting example with a fixed distance and a delta angle of 60°. In this case here the ray casing algorithm determines that buildings b_1, b_2 and b_4 can be seen.

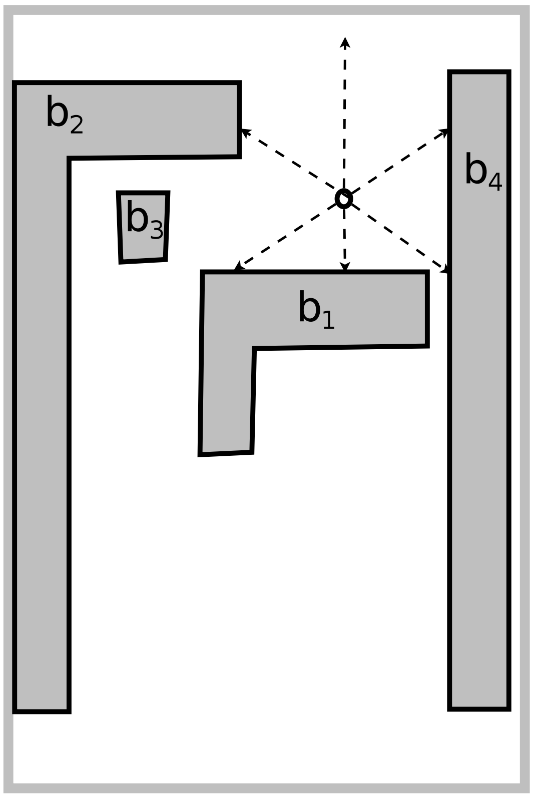


Figure 1.1: Visible objects determined via positionary casting.

A database index is a data structure defined on particular columns of database tables to significantly speed up data retrieval operations from the disk and is the key component of a Database Management System (DBMS) [7]. The Generalized Search Tree (GiST) [17] data structure is used to build a variety of disk-based indexes such as B+ and R-Trees. The latter, is a dynamic index structure designed to optimize spatial searching [18]. The use of such indexes to describe visibility relations between objects and answer queries such as "Which buildings can be seen from point A?" is not a common practice, and their performance against traditional ray shooting methods has not been widely investigated.

1.2 Problem statement

Ray casting is a very basic algorithm for defining visible objects around a stationary object. Although it is very accurate, performance is not its strongest feature. This approach cannot provide sufficient results for a scenario where we want to define in real time 'what can be seen' from the point of view of pedestrians. A different approach to this problem is to use pre-computed spatial database indexes built on the isovist concept. We wish to investigate the extent to which such index based implementations can provide better solutions to the 'what is visible' problem in terms of run-time performance and accuracy over ray casting approaches.

1.3 Goal

The construction of such isovist indexes along with an evaluation of their performance is the main focus of this thesis. First, an algorithm for generating these indexes named $IsoVist$ index builder is described in detail. Then, the execution time of the algorithm under various input parameters is presented. Finally, the performance of $IsoVist$ index is analyzed to show that in fact, visibility queries can be answered more efficiently than with traditional ray casting under various circumstances.

1.4 Structure of the report

Chapter 2 presents the algorithms that we considered and implemented in this project. Chapter 3 focuses on technical details regarding the $IsoVist$ index builder algorithm. A series of experiments is performed in chapter 4 and their evaluation is presented and discussed. Finally the conclusions are included in chapter 5.

Chapter 2

Algorithms

2.1 Ray casting

Ray casting is a basic computer graphics rendering algorithm, first presented by Arthur Appel in 1968 [12]. From a given point p in the 2D plane rays are shot at an increasing angle. If the ray reaches a polygon in the plane, and is not intercepted by other objects, that polygon is visible from p. Usually ray casting is applied within a buffer radius around the point of interest. Considering a pedestrian in a crowded area, such buffer would simulate his field of vision.

In the scope of this thesis, a basic ray casting algorithm was implemented (Algorithm 1) and its performance was compared against the $IsoVist$ index builder. The algorithm identifies the visible buildings around a given point p in a scene. Rays are shot at an increasing angle a around p. The first object that is intersected by a ray is identified as visible. The generation of the rays is simulated using the following procedure. First a buffer of radius R is created around p. To capture all the buildings of a scene within the buffer, as maximum radius R_{max} is selected the diagonal of that scene, meaning, the buffer encloses the scene. Then the exterior ring of the buffer is segmented into tiny line segments. The starting and ending points of each segment are the endpoints of rays shot from p, where two consecutive rays form an angle of a.

Algorithm 1: Basic ray casting

```
  Input: point $x$, buffer $b$, polygons $S$
  Result: $V$ set of polygons visible from $x$
1 $buffer\_points$ := segment_buffer($b$);
2 foreach $p \in buffer\_points$ do
3     $ray \leftarrow$ segment($x, p$);
4     foreach $P \in \mathcal{S}$ do
5         if intersects($ray, P$) then
6             $I \leftarrow I \cup \{P\}$;
7     $n$ := nearest_intersection($x, I$);
8     $V \leftarrow V \cup \{n\}$;
```

The algorithm presented above is a basic implementation of ray casting. Details regarding `segment_buffer` can be found in Appendix C. Function `nearest_intersection` identifies the closest intersected polygon point by the `ray` from `x` and returns the polygon `n` where the intersected point belongs to.

2.2 Visibility Polygon

In computational geometry, the visibility polygon for a point p in the 2D plane among obstacles is the possibly unbounded polygonal region of all points of the plane visible from p. There are various algorithms that solve the visibility polygon problem. *Uniform ray casting* among them, that was explained in the previous section, is the most naive approach and it requires an infinite amount of rays to produce results with high accuracy.

Another approach which is common in simple applications such as video games is *ray casting to every vertex.* Instead of casting rays at increasing angles, they are cast only towards the endpoints of polygon edges, aiming to decrease the execution time[13]. A basic implementation of the Algorithm 2 is presented below.

Algorithm 2: Ray casting to every vertex

```
  Input: point $x$, obstacles $S$
  Result: $V$ set of polygons visible from $x$
1 foreach obstacle $b \in S$ do
2     foreach vertex $v$ of $b$ do
3         $l$ := ray($x, v$);
4         $r$ := distance($x, v$);
5         $\theta$ := angle of $v$ with respect to $x$;
6         foreach obstacle $b'$ in $S$ do
7             r := min($r$, distance($x, b'$));
8         $V \leftarrow V \cup$ {vertex($\theta, r$)};
```

The time complexity of this algorithm is $O(n^2)$ because for each ray shot to every of the n vertices it checks whether it intersects with any of the obstacles in the scene.

There are a lot of algorithms proposed over the years with lower complexity than ray casting approaches. Among these algorithms are the visibility polygon for point in a simple polygon[4, 14], for a point among line segments that don't intersect with each other[15] with a worst case time complexity $\Theta(n \log n)$, and the angular sweep or rotational plane sweep with a time complexity of $O(n \log n)$[16].

In the scope of this project, we are interested in the objects that are visible from a given point p in the 2D plane and not the infinite set of points that can be seen from p. We consider the approach of ray casting to every vertex to be sufficient to cover our visibility requirements, and we rely on it to implement the $IsoVist$ index builder, even though other algorithms can perform better.

2.3 IsoVist Index Builder

In the following sections we describe our approach, its advantages and limitations, and we explain in detail the necessary steps for constructing an isovist index.

2.3.1 Definitions

- **Visible Point:** Visible point is a point in the scene that can be seen from a given polygon point.
- **Shadow Point:** Shadow point is a point on an edge of a polygon collinear with two other points in the scene such that the shadow point is visible from both these points.
- **Reflection Point:** Reflection point is a point on an edge of a polygon that is collinear with a shadow and a polygon point and is visible from both the shadow point and the polygon.
- **Isovist:** Isovist, is a the set of points on the plane that are visible from a given point.
- **Conjecture:** Any point visible from a line segment in the scene is visible from either the end points of the segment or any reflection point or shadow point of the segment.

Preliminaries

The input of the index building algorithm is a scene S consisting of a set of n polygons $S = \{P_0, P_1, \ldots, P_n\}$[1]. Each polygon P_i consists of a sequence of m polygon points $p_{P_i} = \{p_{i1}, \ldots, p_{im}\}$ points which determine the clockwise definition

[1] P_0 is a distinguished inverse polygon that defines the scene boundary.

of a simple polygon. In the work here we assume that all polygons are simple, that is they have no holes.

From this we associate with each polygon P_i a set of k line segments $l_{P_i} = \{l_{i1}, ..., l_{ik}\}$ which define the line segments making up the polygon, namely the polygon edges. $\mathcal{P}$ defines all the polygon points in the scene ($\mathcal{P} = \{p_{P_1}, \ldots, p_{P_n}\}, P_i \in S$. $\mathcal{L}$ is the set of all polygon edges (line segments) in the scene ($\mathcal{L} = \{l_{P_1}, \ldots, l_{P_n}\}, P_i \in S$.

Note that while the set $\mathcal{P}$ is finite, the set of distinct points that may be positioned in the scene which we denote as $\mathcal{X}$ is infinite ($\mathcal{P} \subset \mathcal{X}$). We say two points ($x$ and x') are *visible* to each other if the line segment $\texttt{segment}(x, x')$ between them does not *cross*[2] any line segment in $\mathcal{L}$. Formally $\texttt{visible}(x, x') \Leftrightarrow \neg\texttt{crosses(segment(x,x'),}l_i\texttt{)}, \forall l_i \in \mathcal{L}$. In addition to the `segment` constructor, we also use `ray` below. A $\texttt{ray}(x, p)$ denotes a line of infinite length that starts from point x and passes from point p.

Symbol	Description	Expression
P	Polygon	
S	Scene of polygons	$S = \{P_0, P_1, \ldots, P_n\}$
p_{P_i}	Polygon points of P_i	$p_{P_i} = \{p_{i1}, \ldots, p_{im}\}$
l_{P_i}	Edges of P_i	$l_{P_i} = \{l_{i1}, ..., l_{ik}\}$
$\mathcal{P}$	Set of all polygon points in S	$\mathcal{P} = \{p_{P_1}, \ldots, p_{P_n}\}, P_i \in S$
$\mathcal{L}$	Set of all polygon edges in S	$\mathcal{L} = \{l_{P_1}, \ldots, l_{P_n}\}, P_i \in S$
$\mathcal{X}$	All points in S	$\mathcal{P} \subset \mathcal{X}$

Table 2.1: Preliminaries used to describe the index builder algorithm.

2.3.2 Determining the isovist of a single point

Consider the basic problem of defining an isovist for a given point $x \in \mathcal{X}$ over the scene S. The basic strategy is to determine the points in $\mathcal{P}$ that are visible from x (figure 2.1), to order these points in a clockwise fashion around x, and then to compose an isovist polygon by uioning the sequence of triangles defined (figure 2.4).

Calculating visible points

For a given point x, algorithm 3 iterates through all polygon points $p \in \mathcal{P}$ in the scene to determine those visible. For every $p \in \mathcal{P}$ it creates the line segment $\texttt{segment}(x, p)$, and checks if it `crosses` any of the polygon edges $l \in \mathcal{L}$ of the scene. If no edge is crossed, p is added in V that is the set of visible polygon points of x.

[2]For two line segments to cross they must share exactly one point on both of their interiors. Note for two line segments that *touch* at end points, do not *cross*. Nor does a line segment that overlaps another line segment.

Algorithm 3: Visible points

```
Input: point $x$
Result: $V$ set of visible polygon points of $x$
1 foreach $p \in \mathcal{P}$, $p \neq x$ do
2     $possible := true$;
3     foreach $l \in \mathcal{L}$ do
4         if $crosses(segment(x, p), l)$ then
5             possible := false;
6             break;
7     if $possible$ then
8         $V \leftarrow V \cup \{p\}$;
```

After this step all visible polygon points around x have been computed. In figure 2.1 we can notice a segmentation of the space around the point of interest.

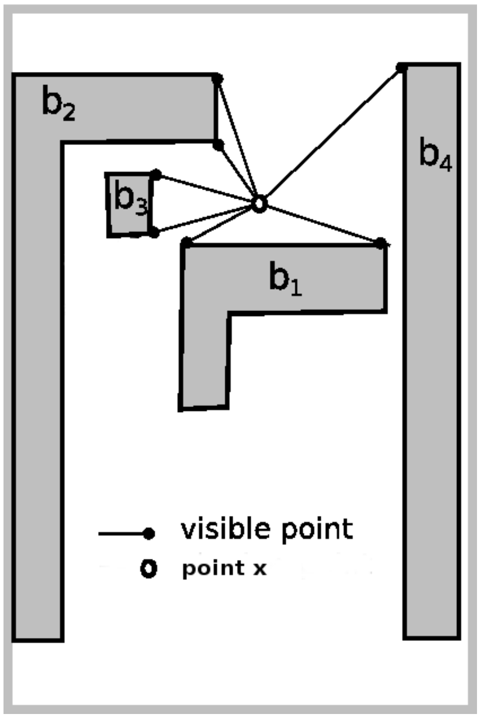


Figure 2.1: Visible points of point x.

Calculating shadow points

There is one complication, illustrated in figure 2.2. Only visible points cannot define the visible space around a given point. This shows the necessity of calculating what we call *shadow points*. When the eye focuses on a corner of a building (point p)

from vantage point x it may see other objects behind p. This happens when the segment `ray`(x, p) does not cross into the polygon P at p. We simulate a `ray`(x, p) of infinite length by extending the `segment`(x, p) to the scene boundary polygon P_0.

In such a case there exists a point p' which is the exact point where `ray`(x, p) crosses some line segment $l \in \mathcal{L}$. Since the `ray` is extended to the enclosing polygon P_0 of the scene, it possibly crosses many edges including a segment of P_0. We pick as shadow point p' the closest shadow candidate to p.

The algorithm 4 calculates the shadow points of the visible points generated by algorithm 3.

Algorithm 4: Shadow points

```
   Input: $x$, $V$ visible points of $x$
   Result: $S$ set of shadow points of $x$
1  foreach $p \in V$ do
2      if Corner(x,p) then
3          $dist$ := inf;
4          foreach $l \in \mathcal{L}$ do
5              if $crosses(ray(x, p), l)$ then
6                  $p'$ := ray$(x, p) \cap l$;
7                  if $dist(x, p') < dist$ then
8                      $closest := p'$;
9                      $dist := dist(x, p')$
10         $S \leftarrow S \cup \{closest\}$;
```

It is now clear that visible and shadow points start to form triangles (figure 2.2) within which the point x can be seen.

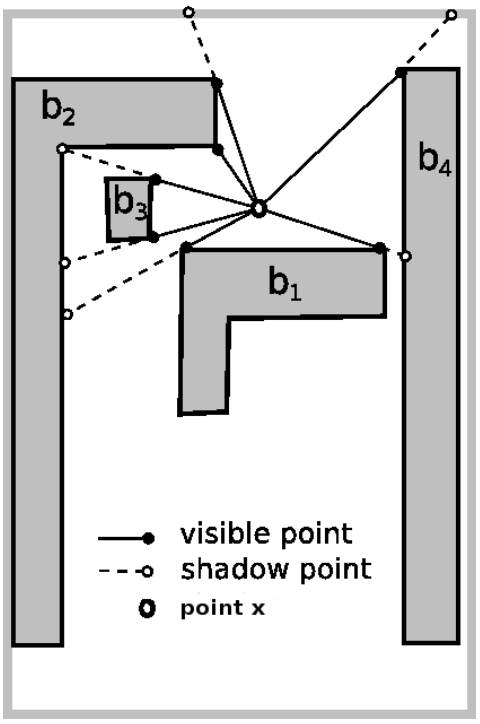


Figure 2.2: Segmentation of space by the visible and shadow points of point x.

Order Rays

To construct the isovist of x the segments of visible space defined by visible and shadow rays have to be joined. The following algorithm explains how these rays are ordered in a clockwise fashion to produce polygons which then form the point isovist.

Visible and shadow rays that originate from the same polygon point p define the viewing angle that the point occupies. Each polygon point connects two adjacent edges. Their respective rays define the minimum and maximum viewing angle for this point. Every other ray is located in between those two. All ordered rays, are combined in pairs to form triangles of visible space which later on are unioned to construct a polygon representing the point isovist.

Algorithm 5 selects the adjacent edges of p and labels them as min and max rays by comparing their angle from the x axis. The angle of each ray from the x axis ranges from 0 to 2π ($0 \leq \theta_{r_i} \leq 2\pi$). By subtracting the angle of min ray from all rays, min is shifted to be the x axis. To achieve a clockwise order from max to min, min is 0 if no ray has an angle greater than max, else min is 2π. Finally, it performs a quick-sort on the rays' angles.

Algorithm 5: Order Rays

```
  Input: $p \in P$, $R$ visible and shadow rays of $p$
  Result: $\mathcal{R}$ sorted rays of $p$
1 $r_{min}, r_{max} \in R$;
2 foreach $r_i \in R$ do
3     $\theta_{r_i} := \theta_{r_i} - \theta_{r_{min}}$;
4 if ($\theta_{r_i} > \theta_{r_{max}}$) then $\theta_{r_{min}} \leftarrow 2\pi$;
5 else $\theta_{r_{min}} \leftarrow 0$;
6 $\mathcal{R}$ := quicksort($R$);
```

In figure 2.3, all shadow and visible rays are sorted and are between the min and max rays (1 and 14) that define the maximum viewing angle. It is obvious that two consecutive shadow or visible rays can form a triangle since their endpoints are on the same polygon edge. For example shadow rays 1 and 4 end on the bounding box, and visible rays 3 and 5 end at the same polygon edge. This ensures that the union of all triangles succeeds, resulting to the point isovist.

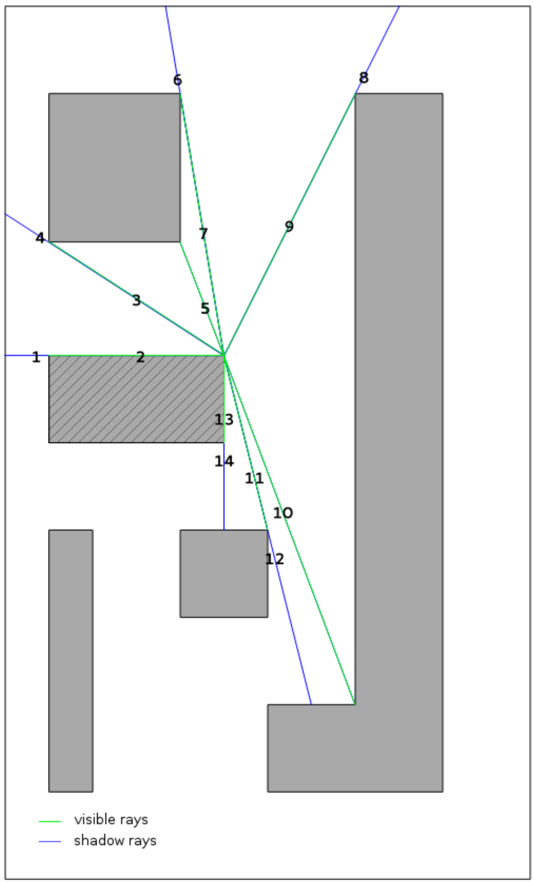


Figure 2.3: All rays of a polygon point ordered in clockwise fashion.

Synthesizing the point isovist

Visible and shadow points define ray segments that originate from point x at a given angle θ. The rays are ordered on θ clockwise, and this defines a set of triangles that can be combined in pairs to construct a polygon representing the point isovist.

In more detail, each triangle represents the space between the two selected visible or shadow points and x. Two neighboring rays that can form a triangle [3] are either both visible or both shadow rays and their endpoints are on the same polygon edge. Following the clockwise order, all triangles are produced and unioned with each other, resulting a polygon which represents the point isovist.

After all shadow and visible rays are sorted on θ clockwise, they are stored in a map structure where key is θ and each key is associated with one or more rays. This happens because a shadow `ray`(x,p) will always have the same θ with the visible `segment`(x,p).

[3] Under very rare conditions of co-linearity, a so called *antennae* may be defined [16]. This algorithm passes over such antennas, considering them to be invisible.

Algorithm 6 shows how pairs of rays are selected in a clockwise fashion to produce the point isovist. In each iteration, all rays with angle θ_1 (line 4) are selected (`getAll`(θ_{r_i})) and then the rays with $\theta_2 > \theta_1$ (line 5). Finally, the afore mentioned sets of rays are combined in pairs (line 6) to produce a triangle T. Figure 2.4 shows the triangles that are unioned with each other to produce the polygon that is the point's isovist.

Algorithm 6: Point Isovist

```
   Input: $x$, $R_x$ sorted rays of $x$
   Result: $Isv_p$ point isovist
1  $checked, Isv_p \leftarrow \emptyset$;
2  foreach $(r_i \in R_x)$ do
3      if $(r_i \notin checkedRays)$ then
4          $current \leftarrow$ getAll$(\theta_{r_i})$;
5          $next \leftarrow$ getNextOf$(\theta_{r_i})$;
6          $T \leftarrow$ combine_pairs$(current, next)$;
7          $checked \leftarrow checked \cup current$;
8          $Isv_p \leftarrow Isv_p \cup T$;
```

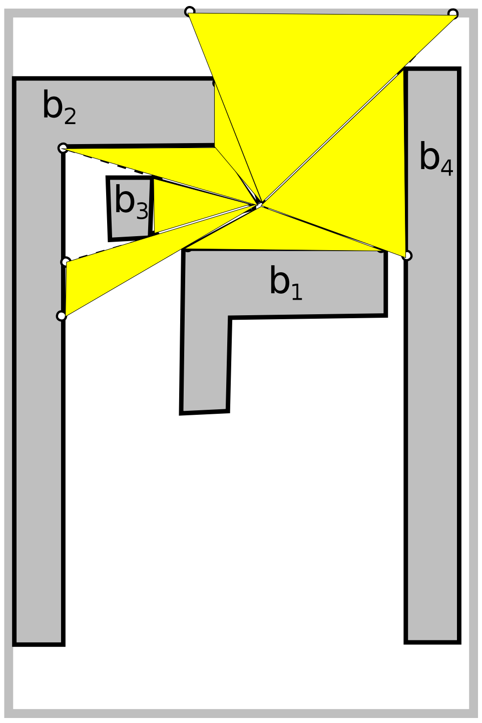


Figure 2.4: Triangles in clockwise order form the Point Isovist

2.3.3 Calculating polygon isovists

To calculate an isovist for a polygon (as in figure **??**), we compute the isovist of each polygon point and union the collection of point isovists. However, there is a complication in this process.

Reflection points

Figure 2.5 shows the isovist of building $b1$ that is computing using visible and shadow points. It is clear that $b1$ can be seen from points in space that are not within the isovist. However the correct result is calculated if the two *reflection points* are added to the polygon. A *reflection point* is the endpoint of a shadow ray. Defining a reflection point is similar to adding a polygon point.

In figure 2.5 two reflection points are added to the polygon of b_1. A `ray`(p_1, p_2) produces the reflection point r_1 and a respective ray from b_3 produces r_2. Reflection points r_1 and r_2 are added to polygon $b1$. Given the addition of reflection points to polygons, the correct isovist of the entire polygon may be computed by unioning all the point isovists (including reflection points) of the polygon.

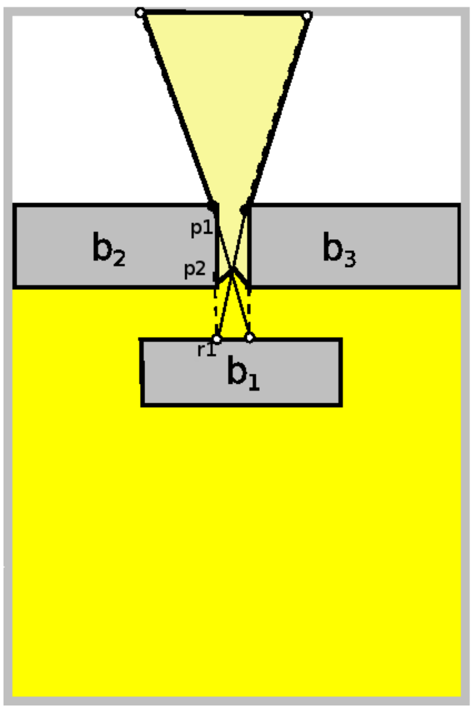
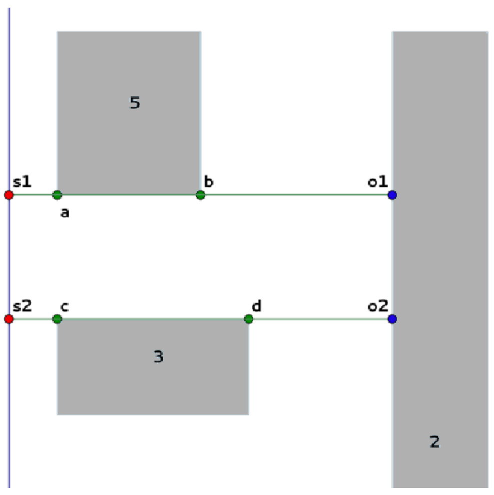


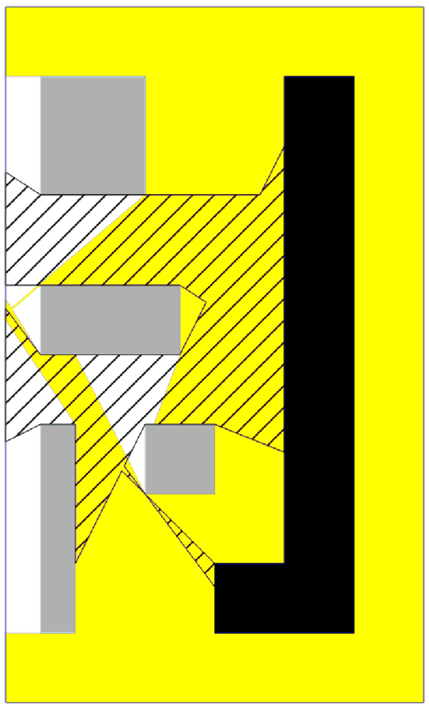
Figure 2.5: Reflection points

Although it seems simple to treat reflection points as polygon points, computing their isovist and unioning it with the polygon's isovist is a very expensive procedure. In a real city environment, there are far too many reflection points per building.

Running algorithms 3, 4, 5, 6 for each generated reflection point would increase the execution time exponentially. An alternative is to combine reflection points that are located on the same polygon edge to create small polygons. In figure 2.6a the reflection rays `ray`(o_1, s_1) and `ray`(o_2, s_2), originate from the same polygon edge and form the `polygon`$(o_1, s_1, s_2, o_2, o_1)$. From any point inside that polygon, building 2 can be seen. This procedure is repeated for any pair of points on the same edge. If a polygon can be constructed and does not intersect with any buildings in the scene, it is added to the regular isovist. Figure 2.6b shows how the isovist computed from visible and shadow points (yellow ink) is augmented by the isovist of the reflection points (diagonal fill).



(a) Reflection rays


(b) Reflection isovist

Algorithm 9: Polygons from reflection points

```
   Input: $R_r$ reflection rays, $S$, Isovist $I$
   Result:
 1 $checked \leftarrow \emptyset$;
 2 foreach ($r_{outer} \in R_r$) do
 3     foreach ($r_{inner} \in R_r$, $r_{inner} \neq r_{outer}$, $r_{inner}, r_{outer} \notin checked$) do
 4         if intersects($r_{inner}, r_{outer}$) then
 5             $P \leftarrow$ create_triangles($r_{inner}, r_{outer}$);
 6         else
 7             $P \leftarrow$ create_polygon($r_{inner}, r_{outer}$);
 8         if $\not$intersects($P, S$) then
 9             $I \leftarrow I \cup P$;
```

Effects of this implementation of reflection points on a isovist of a building are presented in detail in section **??**.

2.3.4 Building the isovist indexes

Creating the isovist index is as simple as storing the isovists in a database table, and creating a GiST[17, 18] index on their geometries. Queries such as *"Which buildings are visible from point x?"* are answered by locating all the buildings where x lies within their corresponding isovist. We elaborate on such queries in our experiments.

Chapter 3

Implementation

This chapter explains the system design, provides information for replicating the input data, presents the generation of a polygon isovist step by step and, finally, analyzes implementation details and decisions made during the development of the algorithms.

3.1 System architecture

The system consists of two components. A spatially enabled relational database and the *index builder* (figure 3.1). The first is an Open Source object-relational database system (PostgreSQL) with the Spatial and Geographic Objects for PostgreSQL (PostGIS) extension enabled. The index builder is a java program and it relies on the computational geometry library Java Topology Suite (JTS)[19]. The components communicate to load geometries of polygons and store the computed polygon isovists.

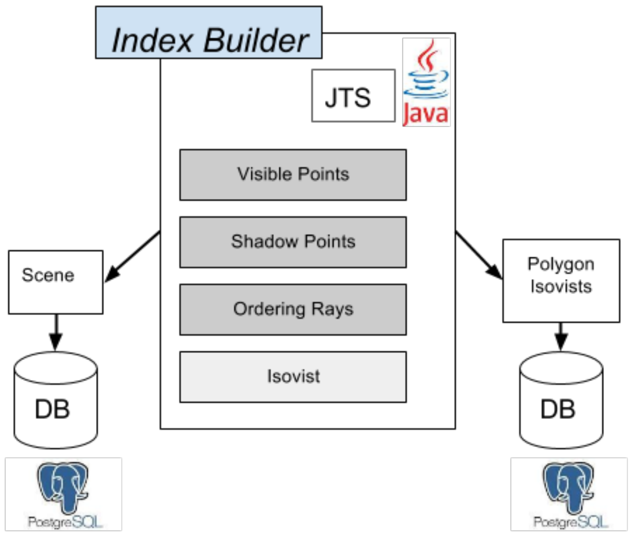


Figure 3.1: System Architecture

In the figure above, a scene of polygons is the input of *index builder* and the polygon isovists are the output that is stored in the database. The index builder consists of the following modules: Visible Points, Shadow Points, Order Rays and the Isovist. In order to accurately compute a polygon isovist with the last module, we need the first three to complete for all polygon points in the scene. This happens because the reflection points of a polygon are derived from all the computed shadows of the scene.

3.2 Generation of a scene

The input of the isovist index builder algorithm is a scene. A scene is a database table that contains the geometries of each building. The data for these buildings are extracted from Open Street Map (OSM)[20] and are a large region of Stockholm. Appendix B shows how to import and filter OSM data in a PostgreSQL database. Initially the imported data contain polygons labeled as buildings, parks, forests, water etc. The filtering process excludes everything but buildings. Seven scenes were extracted to perform experiments and evaluate the algorithms, and they contain 10, 50, 100, 200, 300, 400 and 500 polygons respectively. Figure 3.2 shows one of the larger scenes extracted (500 polygons).

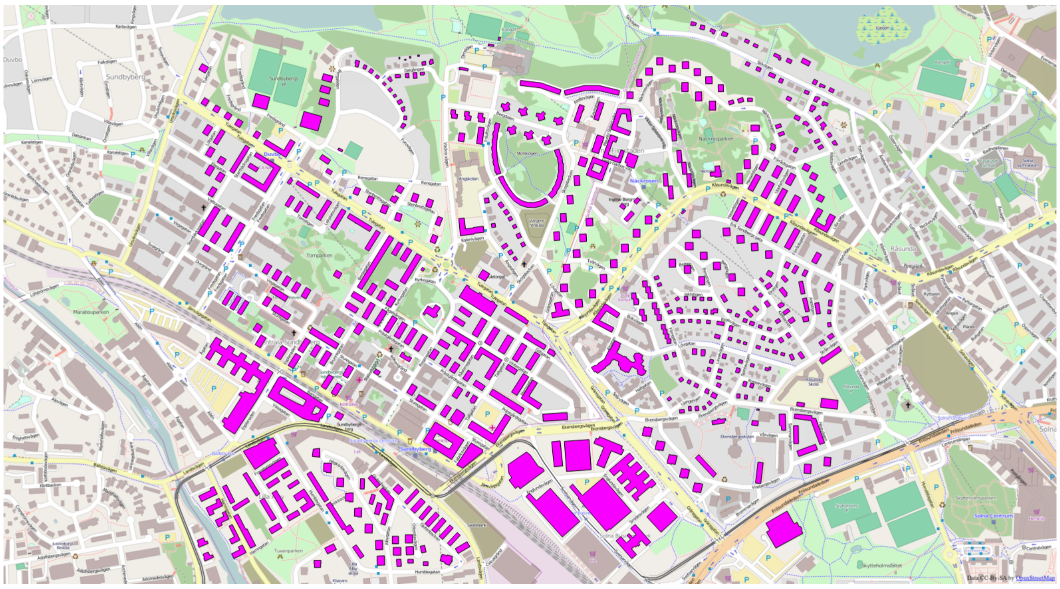


Figure 3.2: Polygon scene from a region of Stockholm.

The buildings that are within the extracted scene in figure 3.2 are highlighted with purple color. The envelope of this scene contains more than 500 buildings. Many polygons were filtered out so that the scene does not contain any buildings that touch with each other. Section 10 elaborates on the filtering process.

3.3 Isovist index builder

Consider the highlighted polygon in figure 3.3. The scene consists of 100 polygons and totally 996 polygon points and is a region around Royal Institute of Technology (KTH). This section shows the calculation of the polygon's isovist in steps (Q building of KTH Main Campus). The scene contains challenging data, such us polygons with holes or complex shapes that require more advanced computations. In chapter 2 the algorithms for calculating a point isovist were presented. This section applies the same steps for every point of the given polygon and finally produces its isovist.

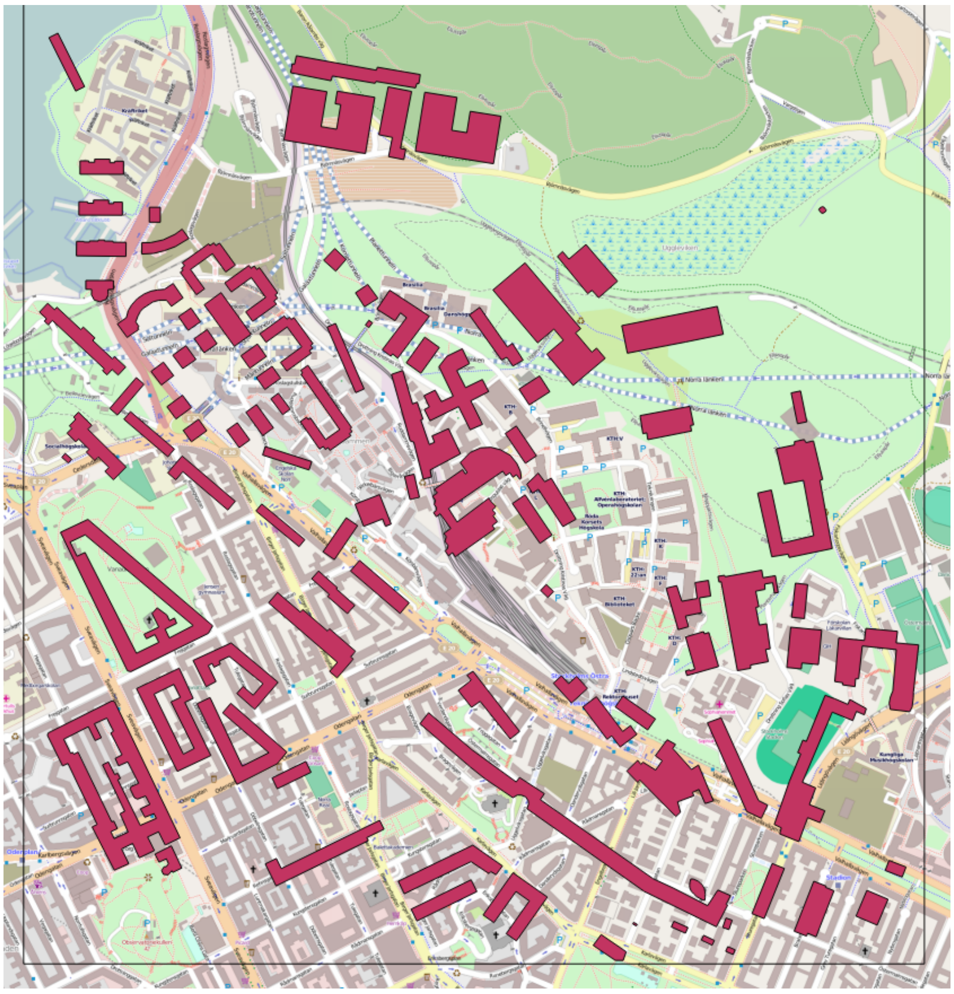


Figure 3.3: Test scene with 100 polygons.

3.3.1 Visible points

The scene is loaded into the *index builder*. For each polygon point in the scene, visible, shadow and reflection points are computed. Figure 3.4 contains visible points (green dots) and corresponding rays of a polygon point located on the upper left corner of the perimeter of the highlighted building.

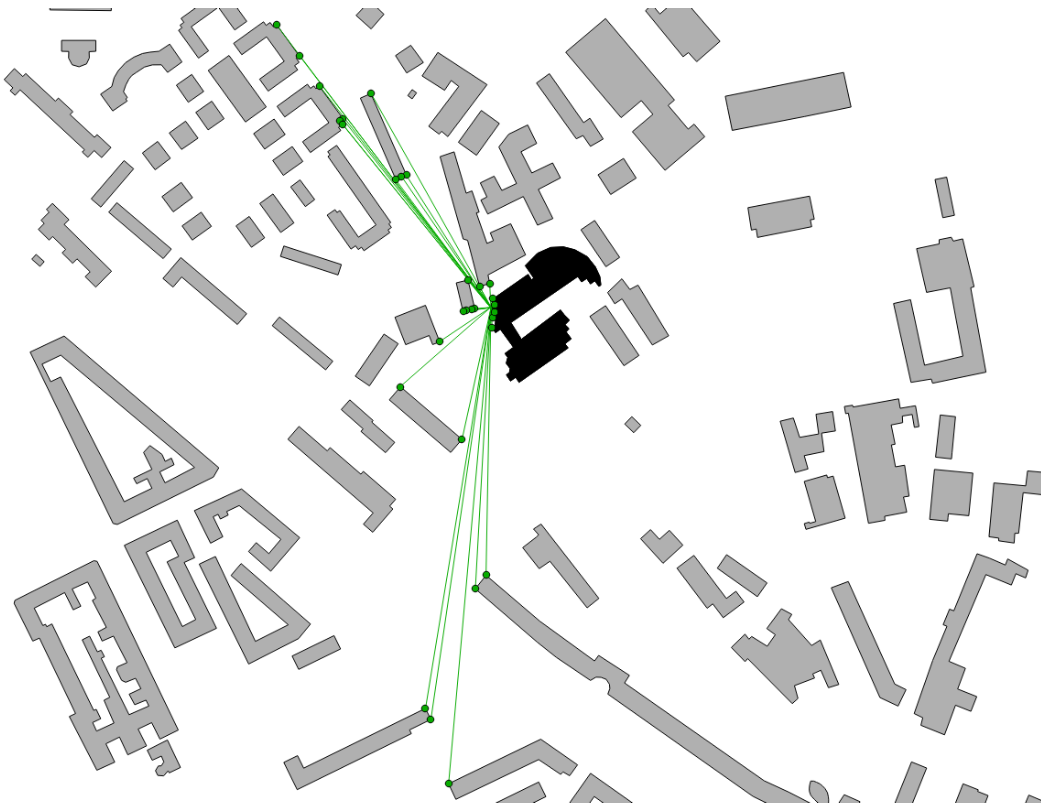


Figure 3.4: Visible points and rays of a polygon point.

For every point b in the scene, a ray is created from starting point a to b and is checked if it `crosses` any edges in the scene, and if not, it names b visible from a. In detail, it is checked whether the ray `intersects` but does not `touch` any polygon. This is simply performed by using the corresponding functions of the JTS Application Programming Interface (API) for a geometries.

In OSM, the coordinates of a point are defined using up to ten decimal digits. On the other hand, JTS uses up to twenty decimal digits. This difference in accuracy is sufficient enough to provide false negative results for functions such as `touches` and `intersects`. For example, a point that is located on a polygon edge may not touch the edge, or a ray may not intersect at a given point with a polygon. The workaround is to use a threshold value for minimum distance. A point that is within the minimum distance from a line segment, is considered to be on that line. Respectively, the `touches` and `intersects` functions operate using the defined threshold value. In this project the chosen threshold is 10 millimeters, which is extremely accurate from a pedestrian's point of view.

3.3.2 Shadow points

For each visible point, shadow points are computed according to algorithm 4. Figure 3.5 shows such points are located on other geometries, on the polygon of the start point and on the bounding box of the scene.

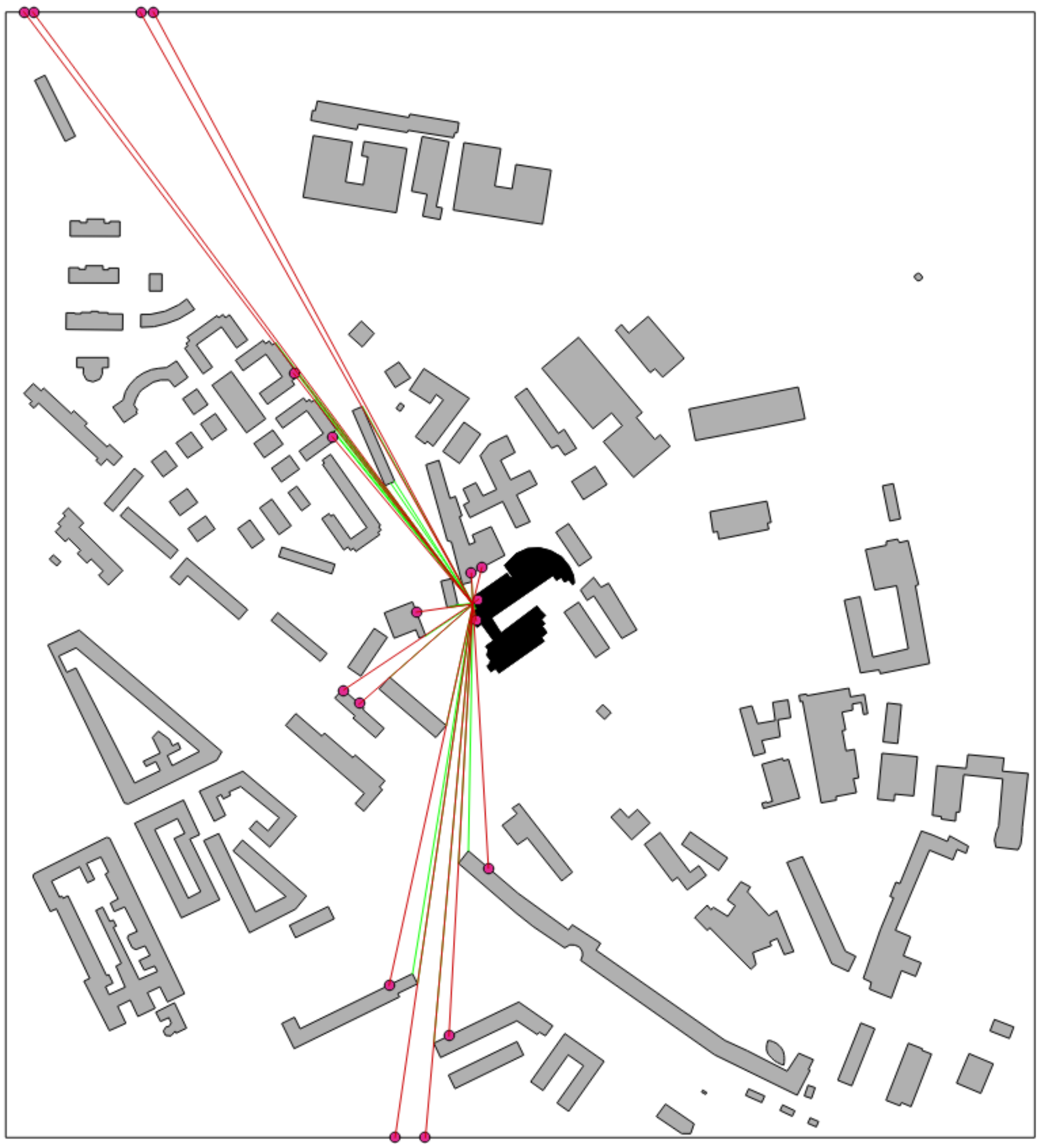


Figure 3.5: Shadow rays on top of visible rays of a polygon point.

The bounding box of the scene consists of four line segments, the edges of the box. A visible ray that is extended towards both directions (ray with infinite length), intersects at least two of these segments. The extended segment with same slope as the visible ray is the shadow ray. The intersection point of that extended segment and the bounding box is the shadow point, except if the ray intersects with any polygons in the scene. Then, as shadow point is defined as explained in algorithm 4, the closest intersection point. Finally, if the shadow point is located on a polygon, it is added as a reflection point.

3.3.3 Polygon isovist

For each polygon point of the highlighted building of figure 3.5 the rays are sorted and all possible triangles are formed according to algorithm 6 and unioned to form a point isovist. Then all the previously computed point isovists (figure 3.6) are unioned to form the isovist of that polygon. Finally, if any reflection points exists, their isovist is computed and merged, with the isovist result according to 9. Figure 3.7 presents the isovist of the highlighted with black ink building.

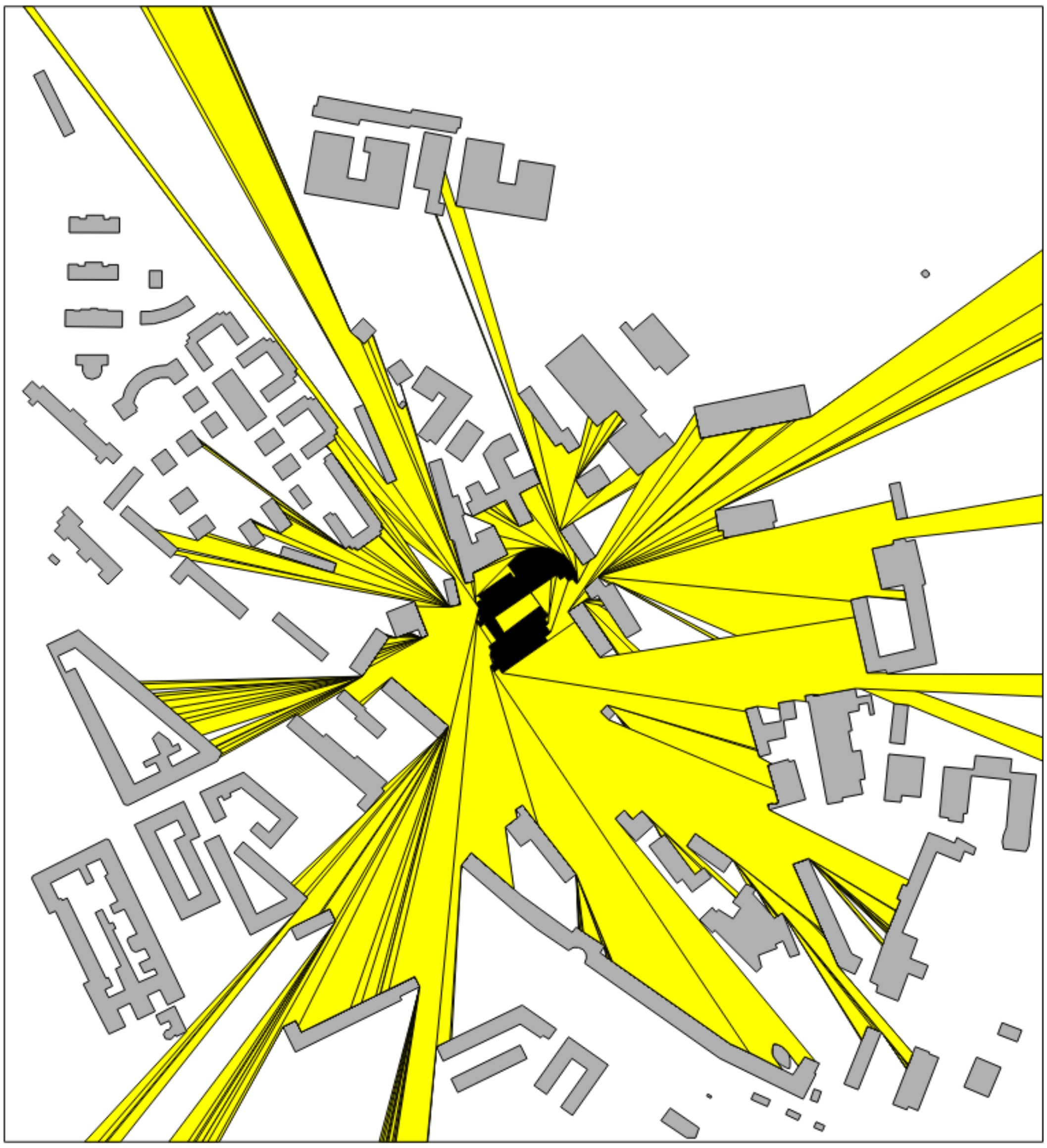


Figure 3.6: All point isovists of the highlighted polygon.

Figure 3.6 shows the point isovist for each polygon point of the highlighted building. Following a clockwise order while merging the isovists for each point ensures the success of the union operation. The isovist polygons of two neighboring polygon points always touch in one or more points. Figure 3.7 features the resulting union.

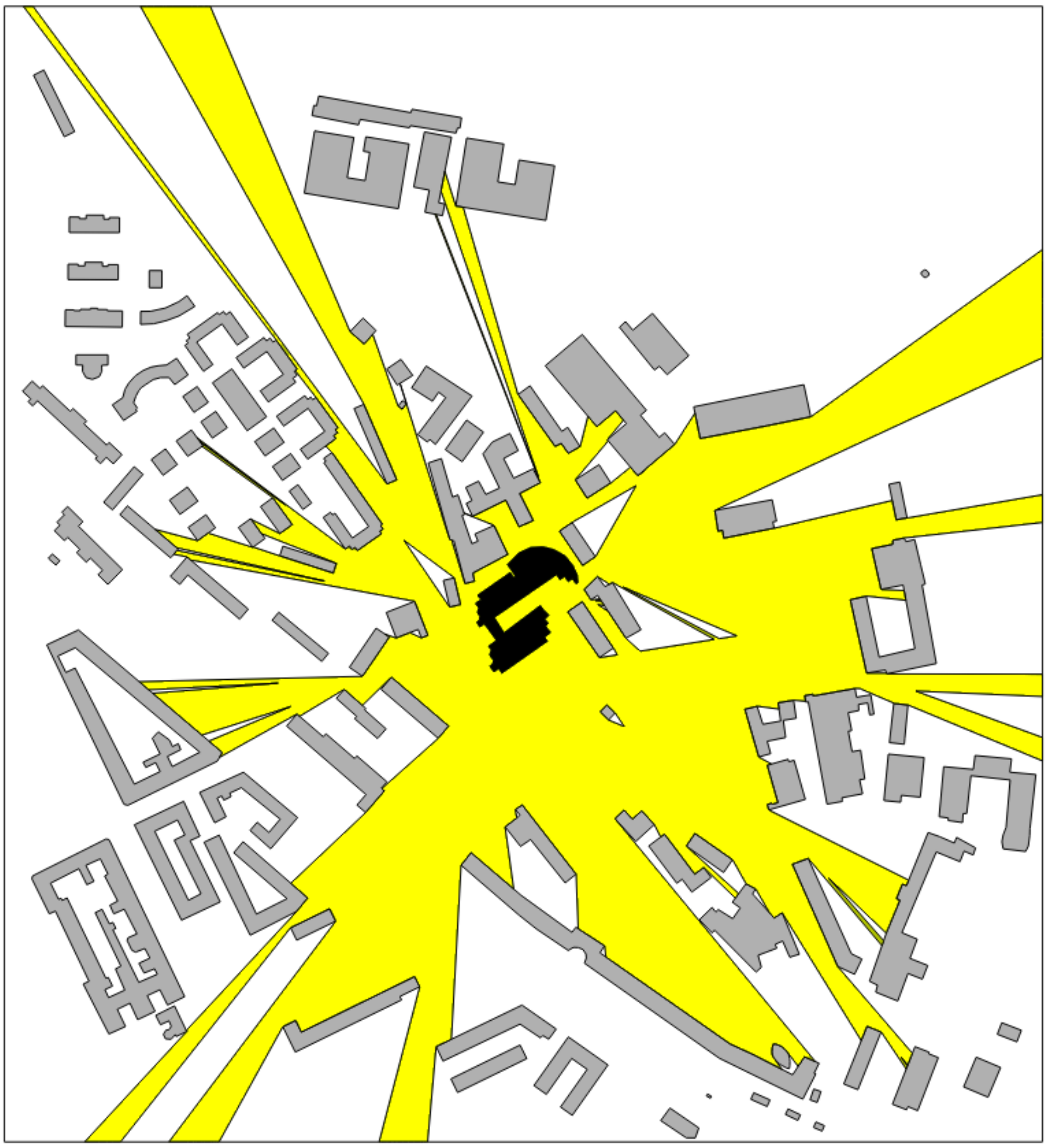


Figure 3.7: Polygon isovist of the highlighted building

Isovist of reflection points.

Section 2.3.3 presented the algorithm for augmenting an isovist result using reflection points. Two approaches were mentioned. The first is to treat reflection points as regular polygon points and for each reflection point run again the visible, shadow and order rays algorithms. The second, is the one used in the current implementation, and was described in detail in 9 and tries to combine reflection rays that originate from the same polygon edge, to form a polygon or a triangle from which that polygon edge is visible.

Since these two are totally different approaches, they are expected to produce unequal results. Reflection as regular polygon points define a polygon with more detail. The first approach gathers more information regarding visible and shadow points thus the computed isovist encloses more space than in the second approach. Figure 3.8 shows two isovists for the same building computed by the 2 different implementations. The scene is the smallest extracted scene (10 polygons). With yellow ink is denoted the isovist produced using the second approach, and with blue ink is the extra space computed by the first approach.

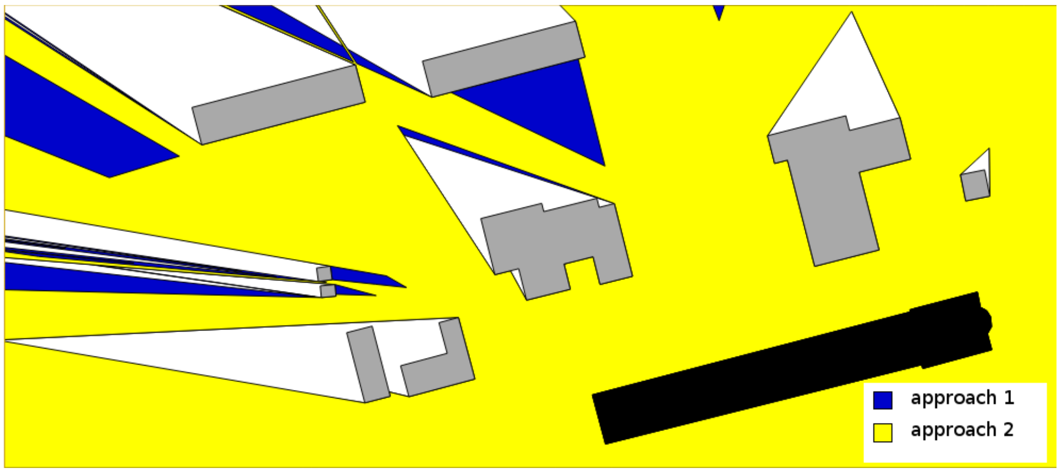


Figure 3.8: Polygon isovist using the two different approaches for reflection points.

A scene with many polygons produces more visible, shadow and reflection points. This derives two things. First, it takes more time to compute the regular isovist, and second, the isovist itself is defined in more details than in a smaller scene. The $IsoVist$ index builder has a complexity of $O(N^2)$ where N are the polygon points in the scene. If the algorithm is implemented using the first approach, the complexity becomes $O(N + R)^2$ where R are the reflection points and $R >> N$. Using the first approach in a large scene will produce optimal results, but clearly, the computation time dependent on $O(R^2)$ operations increases so much, that for the scope of this project, it is not feasible to use. Instead, the second one was chosen for the experiments in chapter 4.

3.4 Other implementation details

Geometry operations

Operations such as `intersects` and `touches` were implemented using both the PostGIS extension or the JTS library. For computing the isovist the most common operation is to check if a line intersects the buildings in the scene. PostGIS provides the function `ST_Intersects(geometry` g_A `, geometry` g_B`)`.

The function creates a bounding box around each parameter and makes use of any available indexes on the geometries. When g_A is a line connecting two polygon points located near opposite corners of the scene and g_B is any of the available scene polygons, the bounding box around g_A is so big that it probably contains g_B and no index can be used. Instead a sequential scan is performed and it is checked against each polygon.

Since the sequential scan in most cases is inevitable, the same procedure was implemented in Java using the JTS Library. For two geometries g_A and g_B it provides the function g_A`.intersects(`g_B`)`. This implementation, always performs a sequential scan but returns when an intersection of a line with a building is found (worst case complexity is O(N)). Additionally, JTS allowed for an easier manipulation of threshold values such as the minimum distance mentioned in section 3.3.1. For example, using PostGIS `intersects` and `touches` produces less accurate isovists than when using JTS. In a scene of 96 polygons, the isovists were computed with both implementations and 83 were the same, but 13 were worse with PostGIS, meaning they captured less visible space around the polygons.

Performance optimizations

The calculation of visible and shadow points of a given point, along with sorting the corresponding rays does not depend on any similar operation for another polygon point. This allows to run in parallel the same operation for all points in a scene. The index builder groups together the above mentioned algorithms and runs them in a separate thread (VSO thread) for every point. On the other hand, the computation of a polygon's isovist depends on reflections points, which are computed when all threads complete. Thus, when all operations for visible and shadow points, order rays complete, a thread for every polygon (ISV thread) computes the polygon isovist. All experiments presented in chapter 4 are based on this multi-threaded implementation of the isovist index builder algorithm.

Chapter 4

Results

A set of experiments were performed to evaluate the performance and the results of the implemented algorithms. The experiments measure the execution time of the $IsoVist$ index builder for various scenes, execution time of visibility queries using the index and the traditional ray shooting algorithm.

All experiments were written in Java including the SQL queries for determining visible buildings around random points. The connection with the database was implemented using Java Database Connectivity (JDBC). The monitoring tool used to measure execution time was Java Simon [21]. All experiments were performed on an isolated laptop under Ubuntu. The hardware specifications of the machine and the software used for development and execution of the experiments are presented in appendix A.

4.1 Generation of the $IsoVist$ index

The first experiment measures the execution time for constructing the isovist index over various sized scenes. The input of the experiment is the seven scenes of polygons that were extracted and the output is the $IsoVist$ index of each scene with the corresponding execution time. As expected, figure 4.1 is showing polynomial growth proportional to the size of the scene in total execution time. This is a result of the N^2 computations for defining the visible points of the scene where N is the total number of polygon points.

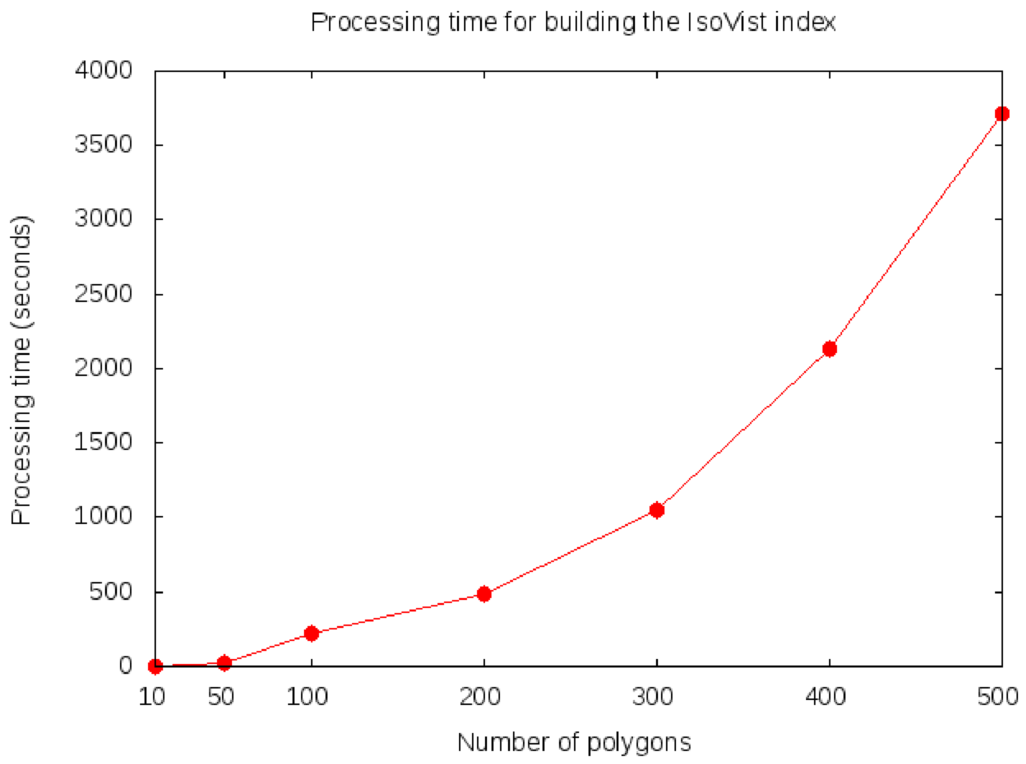


Figure 4.1: Cost for building isovist indexes

Table 4.1 reports the exact numbers for total execution time per scene. The first column refers to the number of polygons per scene and the second column is the total number of polygon points in the scene. The last column is the total execution time as reported by Java Simon for every input scene.

Polygons	Points	Total time(sec)
10	68	5.52
50	256	20.2
100	966	221
200	1354	489
300	1544	1046
400	2163	2129
500	2758	3713

Table 4.1: Execution time of isovist index builder over various sized scenes

The calculation of visible and shadow points and ordering the rays is grouped in one thread (VSO thread) per polygon point. The experiments measures the execution time of each thread and adds it to the total time. A separate thread (ISV thread) computes the isovist of each polygon, after all VSO threads finish. Table 4.2 presents the mean time, standard deviation and total execution time for VSO and ISV threads.

All reported execution times are in seconds

Polygons	Mean VSO	Sd VSO	Total VSO	Mean ISV	Sd ISV	Total ISV
10	0.0516	0.0547	3.51	0.308	0.307	3.08
50	0.0646	0.0871	16.5	0.66	1.84	33
100	0.187	0.117	186	0.349	0.694	34.9
200	0.320	0.191	434	0.275	0.315	55
300	0.582	0.250	898	0.491	2.53	147
400	0.895	0.350	1936	0.541	1.12	216
500	1.19	0.474	3277	0.720	2.86	360

Table 4.2: Mean time, standard deviation and total time for the Visible-Shadow-Order and the Isovist threads.

The mean time of the VSO threads grows proportionally to the number of points in the scene, while the mean time of the ISV threads proportionally to the number of polygons in the scene. The irregular standard deviation (Sd) of the ISV threads proves that the computation of a polygon isovist depends on the complexity of the polygon's geometry, meaning, the more points and edges it has, the more time it takes to compute. The scenes were extracted randomly, and do not have the same level of complexity for geometries.

4.2 Isovist of random points

The second experiment measures the performance of the $IsoVist$ index. The index is a database table that holds information such as the polygon id of each polygon in the scene and the geometry of it's corresponding isovist. A GiST index is created on the geometries of the isovists. An SQL query was written to retrieve all the ids of the polygons, that a random point is within their computed isovist. Batches of random points were created per scene where the point was explicitly located outside the scene's buildings.

The SQL query checks which buildings are visible from each random point.

```
SELECT   r.id, i.polygon_id
FROM   isovist_table i, random_points r
WHERE    ST_Within(r.geometry, i.isovist)
```

It was chosen to check the batch of random points instead of each point individually to show the advantages of using the index on simultaneous requests. Figure 4.2 shows the mean execution time for a set of 100 random points in various sized scenes. The index is used in every query and the query is executed 100 times, one for every point.

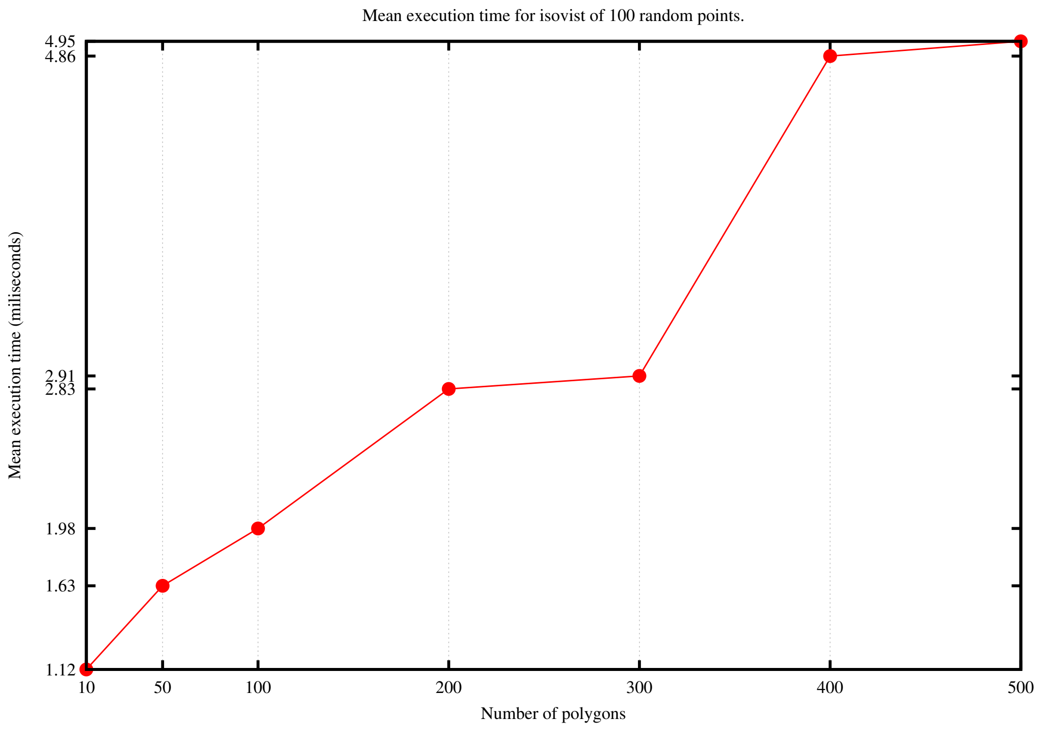


Figure 4.2: Mean time for random point isovist executed for 100 random points

Comparing the execution times of figure 4.2 and the ones in figure 4.3, table 4.3 shows that the index performs better when computing the isovist of many points simultaneously that for single points per query. This is a result of the caching the PostgreSQL database.

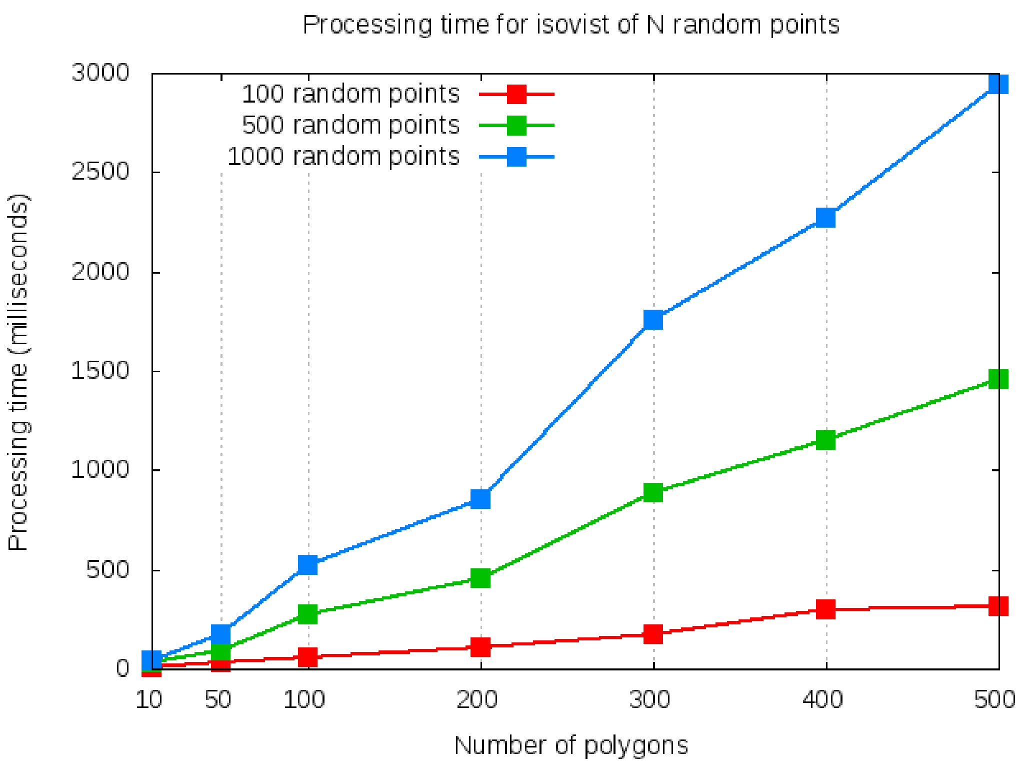


Figure 4.3: Calculation of buildings visible from sets of random (outside) points

All reported execution times are in milliseconds.

Polygons	100 points	500 points	1000 points
10	9.52	36.4	44.1
50	32.8	89.0	174
100	57.1	277	524
200	106	458	854
300	174	883	1760
400	296	1150	2270
500	311	1460	2940

Table 4.3: Execution time for identifying visible buildings from batches of random points.

4.3 Ray shooting vs IsoVist index

The execution time of constructing the *IsoVist* index was presented in the first experiment, while the performance of the index in the second. The last and most challenging experiment attempts to measure and compare the execution time of the index and ray casting. For a given set of random points in a scene, it measures the execution time of both methods and compares the computed visible buildings per random point. Section 2.1 presented an implementation of ray shooting that is used for the purposes of this experiment.

The execution time of the *IsoVist* is established, but the accuracy is not. The means for measuring accuracy is comparing the results with ray shooting. The performance of the latter is affected by the buffer radius and the angle on which the rays are shot. It's accuracy is expected to increase when the angle decreases and the buffer is large enough to include the whole scene.

A comparison method was implemented that gathers the results of both approaches in sets and measures their differences and intersections. In order to compare the execution time of these approaches, a "ground truth" must be established, meaning a baseline on which both methods under several parameters have the same isovist output. Since the index always computes same isovists, the ground truth is extracted by finding the correct ray shooting parameters that produce equal results. Achieving similar results is very tricky. The reason is that the implementation of the index is based on threshold values and also the chosen reflection points module as mentioned in section 3.3.3 does not produce optimal isovists. This concludes that the index cannot be used as ground truth. A different approach is to find the experimental setup under which both methods appear to have equal false negative results. For the same reason mentioned above, such comparison is inefficient and is an approximation of the truth. Finally, the last option for determining the baseline is to identify the ray shooting parameters under which all index results are included in the ray shooting ones, meaning that the latter does not have false negatives comparing to the index. Then it is easier to conclude which method has higher accuracy with trade-off of the execution time.

Table 4.4 shows the parameters of ray shooting that were used as baseline per scene. The radius of the buffer was selected to be the diagonal of the scene, so that all buildings are included. This resulted to choosing a very low angle degree for the ray shooting experiments.

Polygons	Angle (degrees)	Radius (meters)	Execution time Index (ms)	Execution time Ray Shooting (sec)
10	0.25	584.95	59.9	44
50	0.0078125	1792.26	47.7	5120
100	0.0078125	2790.20	212	14958

Table 4.4: Ground truth setup for the 3 scenes and 100 random points

The next three tables (4.5,4.8,,4.8) show the selected angles at which the two algorithms were compared for the scenes that contain 10, 50 and 100 polygons. Each execution of the algorithm is for a set of 100 random points. The first row of each table corresponds to the ground truth. Column Equal denotes the amount random points for which both algorithms had the same output and column different for when the output was not the same. Columns *Not in Ray* and *Not in Index* denote the false negatives per algorithm. Not in ray is the unique number of buildings missed by ray shooting but computed by the index. Not in index is the exact opposite. Finally the last two columns denote the maximum number of buildings that were missed for a random point by each algorithm.

Angle	Equal	Diff	Not in Ray	Not in Index	Max not in Ray	Max not in Index
0.25	93	7	0	5	0	1
0.5	86	14	3	5	1	1
1.0	20	20	5	5	1	1
2.0	76	24	7	4	2	1
4.0	61	39	10	2	3	1
8.0	43	57	10	2	5	1

Table 4.5: False negative results for the scene of 10 polygons

Angle	Equal	Diff	Not in Ray	Not in Index	Max not in Ray	Max not in Index
0.0078125	62	38	0	28	0	4
0.015625	62	38	1	28	1	4
0.03125	63	37	2	27	1	4
0.0625	62	38	4	27	2	4
0.125	60	40	8	27	2	4
0.25	58	42	13	25	2	4
0.5	54	46	17	25	3	4
1.0	48	52	28	24	5	4
2.0	42	58	39	21	7	3
4.0	31	69	47	19	11	3
8.0	20	80	48	15	13	3

Table 4.6: False negative results for the scene of 50 polygons

The three tables have an ascending order on the angles. For the last scene of 100 polygons as ground truth was selected an angle that still produced one false negative result. This was done because of the execution time that increased while selecting smaller angles. When the angle increases, as expected, ray shooting starts missing buildings that are in the Index and the index has a better output than ray shooting. The maximum number of buildings missed by ray shooting increases while the maximum missed by Index decreases.

Both methods are error prone as seen in the results above, but the Index performs clearly much better than ray shooting. Figures 4.4, 4.5 and 4.6 show the

Angle	Equal	Diff	Not in Ray	Not in Index	Max not in Ray	Max not in Index
0.0078125	28	72	1	63	1	5
0.015625	28	72	1	63	1	5
0.03125	28	72	1	63	1	5
0.0625	28	72	3	62	1	5
0.125	28	72	6	62	1	5
0.25	30	70	16	60	2	5
0.5	31	69	36	59	4	5
1.0	29	71	54	50	5	5
2.0	18	82	78	43	7	4
4.0	14	86	91	28	11	3
8.0	11	89	94	25	13	3

Table 4.7: False negative results for the scene of 100 polygons

execution time in seconds of the ray shooting at increasing angle. Previously in figure 4.3 the execution time of the $IsoVist$ index was presented.

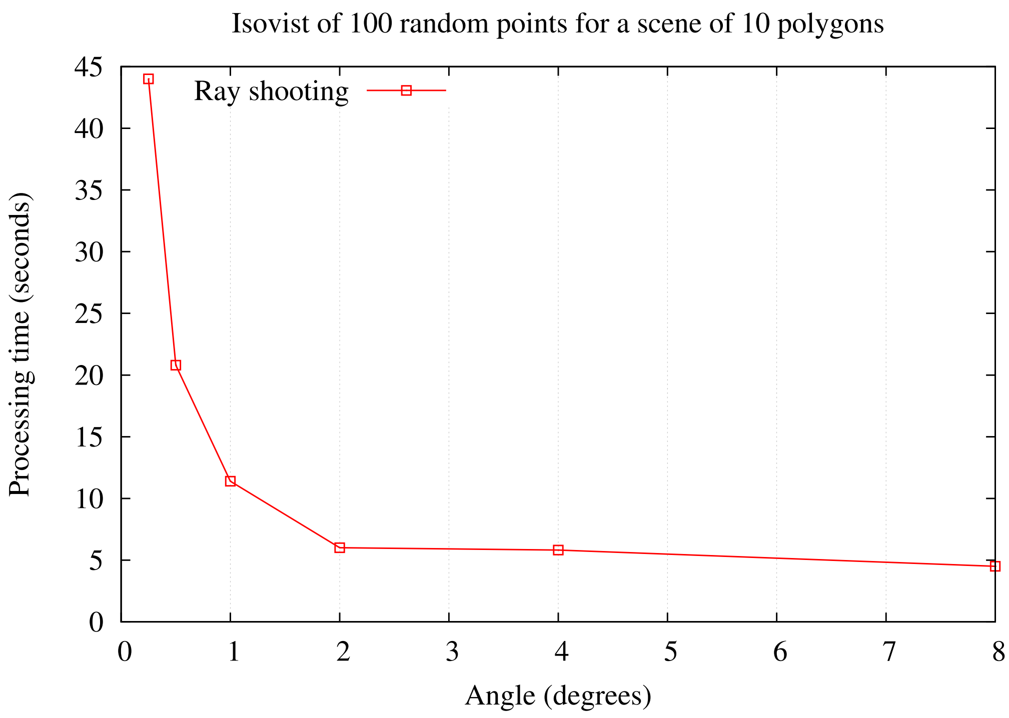


Figure 4.4: Execution time of ray shooting at decreasing for 100 random points in a scene of 10 polygons

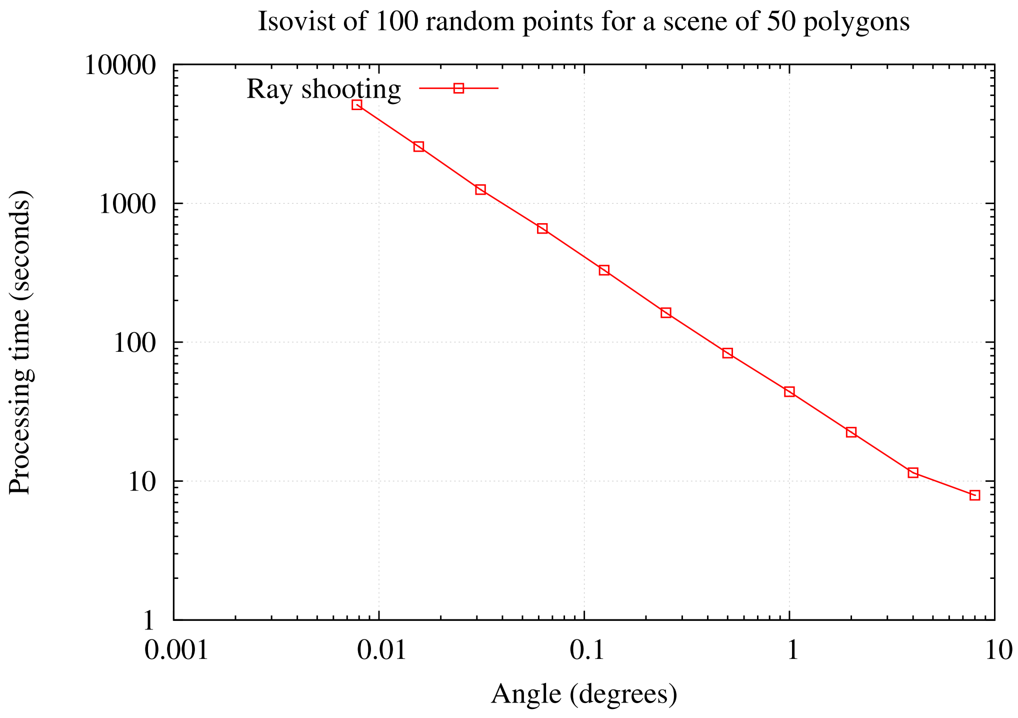


Figure 4.5: Execution time of ray shooting at decreasing for 100 random points in a scene of 50 polygons

All reported execution times are in seconds

Angle	10 Polygons	50 Polygons	100 Polygons
0.0078125		5120	14958
0.015625		2561	7481
0.03125		1254	3747
0.0625		658	1873
0.125		330	939
0.25	44	163	469
0.5	20.8	83.4	241
1.0	11.4	44	126
2.0	6.01	22.5	63.5
4.0	5.82	11.5	32.2
8.0	4.50	7.90	21.8

Table 4.8: Ray shooting: execution time for 100 random points in various sized scenes

The execution times for ray shooting are very high comparing to the ones presented in table 4.3. While the index takes milliseconds to extract visibility relations, ray shooting requires tens of seconds to achieve similar results in terms of accuracy.

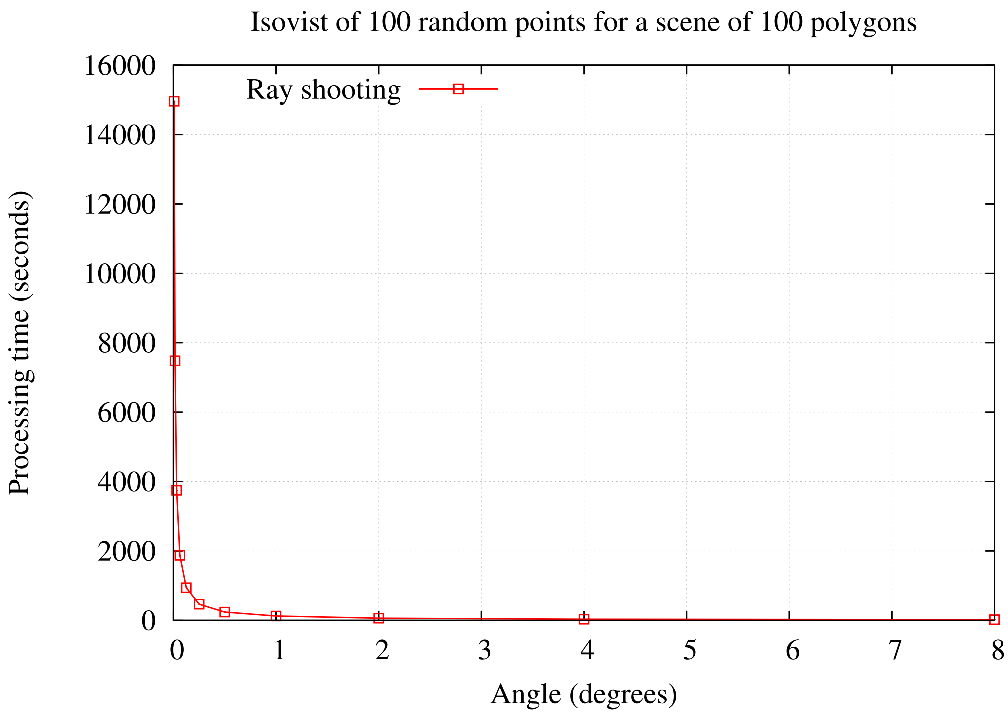


Figure 4.6: Execution time of ray shooting at decreasing for 100 random points in a scene of 100 polygons.

4.4 Discussion

The spatial indexing algorithm has a polynomial complexity proportional to the number of polygon points in the scene. For each of the N points of the scene it iterates through every polygon point to determine which are its visible, shadow and reflection points. Finally for every polygon it iterates through it's points to construct the isovist. The first procedure has complexity $O(N^2)$ while the second $O(N)$.

The construction of the index is a time consuming operation, thus, all experiments were performed using limited sized scenes. Section 4.1 captures the polynomial execution time of the $IsoVist$ index builder algorithm on such scenes. Although the input data were randomly extracted, they provided a diversity of special cases that guided the implementation of the isovist algorithms. The results presented in sections 4.2 and 4.3 show the advantages of using a precomputed spatial index over traditional ray casting to define visibility relations in two dimensional space.

The input data of OSM and their transformation to JTS readable data introduced the first limitation in geometric operations like `touches` and `intersects` and it was discussed in section 3.3.1. The implementation of the algorithm's modules was based on user defined threshold values. Obviously, the accuracy of the computed isovist that mainly relies in such geometric operations, was affected. Section 3.3.3 analyzed another limitation in the implementation which greatly affects the isovist calculation. It was impossible to compute the isovist of reflection points with high accuracy (first approach of reflection point module) without compromising the execution time. This was the reason why only the smallest scene was computed and presented using both approaches.

The above mentioned limitations define the outcome of the experiment 4.3. An isovist computed using the index does not capture all visible space around a polygon. As a consequence, such isovist cannot be used as ground truth for successfully comparing the results with the ones from ray shooting. Instead, an approximation of the ground truth was used, to allow for a comparison of the execution time of the two methods. The purpose of that experiment was to show the advantages of using an index to compute the isovist of a point in real time.

Chapter 5

Conclusions and Future Work

An isovist is the set of points on the plane that are visible from a given point. Defining an isovist using ray casting is a time and resource consuming operation. The $IsoVist$ pre-computed spatial index answers visibility queries efficiently at low runtime.

The idea is to construct an index which describes the visible space around each polygon on the plane and store it in a spatially enabled database system. Then answering a visibility query for a point x is similar to locating all the buildings where x lies within their corresponding isovist. Computing the isovist of a building requires the isovist calculation of every of its polygon points. This is achieved by extracting visibility relations from neighboring visible, shadow and reflection points.

Although the construction of the index has a polynomial execution time, extracting visibility relations for hundreds of points on the plane simultaneously requires less than ten milliseconds. The experiments showed that the index performs better comparing to ray casting. It is practically impossible to achieve high accuracy with ray shooting at similar runtime with the index. Despite the fact that the $IsoVist$ index is error prone due to the selection of threshold values for geometrical operations and the implementation of reflection points, it is preferable than ray shooting for computing visibility relations in a modern environment such as a server setup.

Building the isovist index proved to be a time consuming operation for large scenes. An alternative is to use the principles of external memory viewshed computation [22] and segment the scene into tiles. Then building the index for a scene is reduced to computing visibility relations per tile and combine the results to construct the final index.

A different implementation of the Isovist Builder based on the *visibility polygon* would greatly improve the execution time. The set of algorithms proposed in [15] compute the visible space around a given point and its surrounding obstacles in optimal $\Theta(nlogn)$ time. Since the main goal of this thesis is to measure and evaluate the run time performance of a precomputed index, the current implementation was considered sufficient for this purpose. As part of future work, the Index Builder could be re-designed based on these optimal algorithms, allowing for larger input scenes to be tested since the construction of the index will be significantly faster.

The current implementation of the $IsoVist$ aims to identify which buildings are visible around a stationary object. It is often useful to define the exact visible space at a given angle from a static point. Section 3.3.3 shows that a polygon isovist consists of all its point isovists and each one of them is a triangle that connects the point with an edge of a neighboring building. An observer within that triangle can also see this edge. Using this property, a set of edges around the observer can be extracted and following a clockwise order can form a viewshed. The $IsoVist$ index can be extended to contain information for each point isovist rather than only for polygons, allowing for quick extraction of such edges using corresponding queries.

Finally, the most promising direction for this research is to extend the system to provide visibility information for a moving object or a pedestrian. The index allows for multiple queries to be answered in real time, thus location based applications can benefit from such feature. For example a navigation or a guiding system can provide information for the visible buildings while the user is moving in space.

References

[1] Jiri Bittner and Peter Wonka. Visibility in computer graphics. *Journal of Enviromental Planning*, 30:729–756, 2003.

[2] Michael Minock, Johan Mollevik, and Mattias Åsander. Does tts-based pedestrian navigation work? Technical Report UMINF-07.14, Umeå University, 2014.

[3] M L Benedikt. To take hold of space: isovists and isovist fields, 1979.

[4] D. T. Lee. Visibility of a simple polygon. 22(2):207–221, May 1983. See corrections [**?**].

[5] Subir Ghosh. *Visibility Algorithms in the Plane.* Cambridge University Press, New York, NY, USA, 2007.

[6] Luis Barba, Matias Korman, Stefan Langerman, and Rodrigo I. Silveira. Computing a visibility polygon using few variables. *CoRR*, abs/1111.3584, 2011.

[7] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book.* Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[8] Wassim Suleiman, Thierry Joliveau, and Eric Favier. A new algorithm for 3d isovists. In Sabine Timpf and Patrick Laube, editors, *Advances in Spatial Data Handling*, Advances in Geographic Information Science, pages 157–173. Springer Berlin Heidelberg, 2013.

[9] Phil Bartie, William Mackaness, Morgan Fredriksson, and Jürgen Königsmann. Final viewshed component. In *SpaceBook Spatial & Personal Adaptive Communication Environment: Behaviours & Objects & Operations & Knowledge.* 2013.

[10] Junjun Yin and James D. Carswell. Spatial search techniques for mobile 3d queries in sensor web environments. *ISPRS International Journal of Geo-Information*, 2(1):135–154, 2013.

[11] Michael Minock, Johan Boye, Stephen Clark, Morgan Fredriksson, Hector Geffner, Oliver Lemon, William Mackaness, and Bonnie Webber. Summary of spacebook project results. Technical report, The SpaceBook Project Project (FP7/2011-2014 grant agreement no. 270019)., 2014.

[12] Borko Furht. *Handbook of Multimedia for Digital Entertainment and Arts*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[13] 2d visibility using ray casting and wall tracking. `http://www.redblobgames.com/articles/visibility/`. [Online; accessed 2014-08-27].

[14]

[15] S Suri and J O'Rourke. Worst-case optimal algorithms for constructing visibility polygons with holes. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 14–23, New York, NY, USA, 1986. ACM.

[16] Francisc Bungiu, Michael Hemmer, John Hershberger, Kan Huang, and Alexander Kröller. Efficient computation of visibility polygons. *CoRR*, abs/1403.3905, 2014.

[17] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995.

[18] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[19] Java topology suite. `http://www.vividsolutions.com/jts/JTSHome.htm`. [Online; accessed 2014-08-27].

[20] Open street map. `http://www.openstreetmap.org/about`. [Online; accessed 2014-08-27].

[21] Java simon - simple monitoring api. `https://code.google.com/p/javasimon/`. [Online; accessed 2014-08-27].

[22] Chaulio R. Ferreira, Salles V. G. Magalhães, Marcus V. A. Andrade, W. Randolph Franklin, and André M. Pompermayer. More efficient terrain viewshed computation on massive datasets using external memory. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 494–497, New York, NY, USA, 2012. ACM.

[23] Eclipse java ee ide for web developers. `https://eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/keplersr2`. [Online; accessed 2014-08-27].

[24] Osm2pgsql. `http://wiki.openstreetmap.org/wiki/Osm2pgsql`. [Online; accessed 2014-08-27].

Appendix A

System and hardware specifications

The development process and the experiments of this project have been performed under the same linux machine. Table A.1 shows the hardware characteristics of the system.

Hardware	
Processor	Intel Core i7-2670QM CPU @2.20$GHz \times 8$
Memory	5,7 GiB
Disk	750GiB, 7200rpm

Table A.1: Hardware specifications of the machine.

The machine runs under Ubuntu linux. The development of the algorithms and the execution of experiments was performed using Eclipse Integrated Development Environment (IDE) [23]. All software used in this project together with the development libraries and their respective version details are in table A.2.

Software & Development Libraries	
Operating System	Ubuntu Gnome 14.04 LTS 64-bit
Database Management System	PostgreSQL 9.3.4
Spatial and Geographic objects for PostgreSQL	PostGIS 2.1.3
OSM to PostgreSQL	osm2pgsql SVN version 0.83.0
Quantum GIS	QGIS - 2.0.1
Java	OpenJDK - Java 1.7.0
Java Topology Suite	JTS 1.13
Java Monitoring Tool	Java Simon version 3.5.0
Eclipse IDE	Eclipse Java EE IDE, version Kepler

Table A.2: Software specifications of the machine.

Appendix B

Open Street Map data input

The input of the *IsoVist* index builder algorithm is a set of polygons extracted from OSM data. A user downloads such data via the the OSM web application, and they are formated in an Extensible Markup Language (XML) file. In order to import these data into a spatially enabled database, osm2pgsql[24] is used. This program converts OSM data to postGIS-enabled PostgreSQL databases and it runs via the following command.

```
osm2pgsql -s -U user -d database map.osm
```

This import creates various tables in the database representing the road network, the buildings and the landscape. The information for the buildings are stored in a table named *planet_osm_polygon* and are extracted via the following Structured Query Language (SQL) query:

```
CREATE TABLE buildings AS
SELECT * FROM planet_osm_polygon p
WHERE p.building IS NOT NULL;
```

Finally the extraction of the scenes was performed manually. First, two points in space were selected and an envelope was created where its diagonal was defined by the two points. Then all buildings within that envelope were extracted. Finally, polygons that touched with each other were excluded from each scene. The following queries show how such scenes were extracted. First the bounding box of the scene is extracted, and then the buildings of the scene.

```
CREATE TABLE boundary_10 AS
SELECT ST_boundary(ST_envelope(ST_makeline(
 ST_GeomFromText('POINT(2010666.59␣8248485.49)', 900913),
 ST_GeomFromText('POINT(2010131.27␣8248721.29)', 900913)
))) AS way;
```

```
CREATE TABLE scene_10 AS
SELECT b.osm_id, b.way
```

```
FROM buildings b, indexing_boundary_10 i
WHERE st_within(b.way, st_envelope(i.way));
```

Appendix C

Ray shooting code

The following code is a PostgreSQL function that extracts the endpoints of rays shot from a point p at a given angle a around a buffer of radius R. In detail, the buffer is segmented into tiny lines of a given length l using corresponding PostGIS functions. The starting and ending points of the segments are the endpoints of rays shot from p. The output of this function is all the endpoints of the rays, which are used in the ray shooting module to create the rays.

```
CREATE OR REPLACE FUNCTION _isv_index_segmentize_buffer(
IN geometry, IN double precision, IN double precision)
RETURNS TABLE(id integer, geom text) AS
$BODY$
  DECLARE
    buffer ALIAS for $1;
    angle ALIAS for $2;
    radius ALIAS for $3;
  BEGIN
  return query
  select (dp).path[1] as id, ST_AsText((dp).geom) as geom
  from(
    select st_DumpPoints(
      ST_Segmentize(st_ExteriorRing(buffer),
      2*sin(angle)*radius)) as dp
  ) as sub;
  END;
$BODY$
LANGUAGE plpgsql VOLATILE;
```

Radius of buffer: $radius = R$, Angle: $a = 2 * angle$, Length l: $l = 2 * sin(angle) * R$
The function returns a table with containing the ids of the segment points and their corresponding geometries.