Implementation of Geometric Algorithms for visibility problems

Bachelor of Technology Project Report

-under the supervision of Prof. Sudebkumar Prasant Pal

L ASWANTH KUMAR
13CS30019
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

AUGUST 2016

Table of Contents

CHAPTER 1

GEOMETRIC IMPLEMENTATIONS-CGAL SOFTWARE

Abstract

This thesis summarizes the various implementations of geometric Algorithms. The implementation would require the easy way of representing the various geometric notations. In this session we propose a method for easy way of implementing such geometric figures or notation how they are represented and how they are used to solve the various problems in simple and easy manner. The paper briefly discuss about the usage of software CGAL (Computation Geometry Algorithms Library), an open software which was was founded and initially developed by a consortium consisting of $ETHZ$√ž$rich$ (Switzerland), $FreieUniversit$√§t Berlin (Germany), $INRIASophia-Antipolis$ (France), $Martin-Luther-Universit$√§t Halle-Wittenberg (Germany), $Max-PlanckInstitutf$√ž$rInformatik, Saarbr$√ž$cken$ (Germany), $RISCLinz$ (Austria) $Tel-AvivUniversity$ (Israel), and $UtrechtUniversity$ (The Netherlands).

1.1 Introduction to CGAL Software

1.1.1 Goal and Examples

The goal of the CGAL Open Source Project is to provide easy access to efficient and reliable geometric algorithms in the form of a C++ library.The Computational Geometry Algorithms Library offers data structures and algorithms like triangulations, Voronoi diagrams, Polygons, Cell Complexes and Polyhedra, arrangements of curves, mesh generation, geometry processing, convex hull algorithms, to name just a few.

All these data structures and algorithms operate on geometric objects like points and segments, and perform geometric tests on them. These objects and predicates are regrouped in CGAL Kernels.Finally, the Support Library offers geometric object generators and spatial sorting functions, as well as a matrix search framework and a solver for linear and quadratic programs. It further offers interfaces to third party software such as the GUI libraries Qt, *Geomview*, and the Boost Graph Library.

1.1.2 Various Real-world projects using CGAL

List of projects which uses CGAL are as follow:
$Architecture, BuildingsModeling, UrbanModeling$
$Astronomy$
$ComputationalGeometryandGeometricComputing$
$ComputerGraphics$
$ComputationalTopologyandShapeMatching$
$ComputerVision, ImageProcessing, Photogrammetry$
$Games, VirtualWorlds$
$GeographicInformationSystems$
$GeologyandGeophysics$
$GeometryProcessing$
$MedicalModelingandBiophysics$
$MeshGenerationandSurfaceReconstruction$
$2Dand3DModelers$
$MolecularModeling$
$MotionPlanning$
$ParticlePhysics, Materials, Nanostructures, Microstructures, FluidDynamics$
$Peer-to-PeerVirtualEnvironment$
$SensorNetworks$

1.1.3 Benefits of using CGAL

CGAL produces correct results, in spite of intermediate roundoff errors. If three lines meet in one point, they will do so in CGAL as well, and if a fourth line misses this point by 1.0e-380, then it also misses it in CGAL. Situations that are sometimes tagged as "degenerate" (like a 3-D point set actually living in a 2-D plane) are properly handled by CGAL. In fancy terms, this is called the exact computation paradigm, and it ultimately relies on computing with numbers of arbitrary precision. The exact computation paradigm is not an invention of CGAL, but CGAL is probably the place that implements it at a large scale.

Such guaranteed correctness requires that CGAL is properly used (see the FAQ section on using inexact number types), and it comes at a price: compared to algorithms that use fixed-precision numbers only, the performance is lower. In the best case (as in computing Delaunay triangulations in 3-space), the overhead is around 25%. This is possible because CGAL tracks error bounds and resorts to extended precision only when this is really needed.

In CGAL, we write the high-level algorithms in terms of a well-chosen set of basic questions (where is a point with respect to a line?) and basic objects (like a circle through three points). Doing this in the right way is not always easy, but once it is done, we have outsourced all the numerical issues, and we only have to make sure that the part of CGAL concerned with the basics returns correct results. Given this, the algorithms on top of it just work. Not in most cases, but always!

Getting the underlying basics always right must obviously involve something beyond naive floating-point computations, and it indeed does. The details are pretty complex, but what essentially happens is that we increase the numerical precision of the computations, if necessary, by using numbers that in principle allow arbitrary precision. These techniques are constantly being refined, with the goal of increasing the overall performance of the high-level algorithms under the exact computation paradigm.

1.1.4 CGAL concepts

All CGAL header files are in the subdirectory include/CGAL. All CGAL classes and functions are in the namespace CGAL. Classes start with a capital letter, global function with a lowercase letter, and constants are all uppercase. The dimension of an object is expressed with a suffix.The geometric primitives, like the point type, are defined in a kernel. A predicate has a discrete set of possible results, whereas a construction produces either a number, or another geometric entity.

1.2 Important CGAL Packages

1.2.1 Arthimetic and Algebraic

CGAL is targeting towards exact computation with non-linear objects, in particular objects defined on algebraic curves and surfaces. As a consequence types representing polynomials, algebraic extensions and finite fields play a more important role in related implementations. This package has been introduced to stay abreast of these changes.

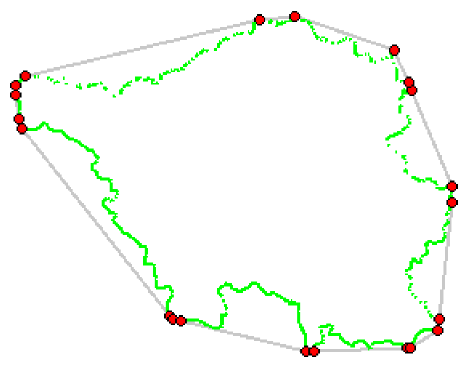
The built-in number types float, double and long double have the required arithmetic and comparison operators. They lack some required routines though which are automatically included by CGAL. All built-in number types of C++ can represent a discrete (bounded) subset of the rational numbers only. We assume that the floating-point arithmetic of your machine follows Ieee floating-point standard. Since the floating-point culture has much more infrastructural support (hardware, language definition and compiler) than exact computation, it is very efficient. An algebraic structure is considered exact, if all operations required by its concept are computed such that a comparison of two algebraic expressions is always correct. An algebraic structure is considered as numerically sensitive, if the performance of the type is sensitive to the condition number of an algorithm. Note that there is really a difference among these two notions, e.g., the fundamental type int is not numerical sensitive but considered inexact due to overflow.

The package introduces a concept $Polynomial\_d$, a concept for multivariate polynomials in d variables. Though the concept is written for an arbitrary number of variables, the number of variables is considered as fixed for a particular model of $Polynomial\_d$. The concept also allows univariate polynomials.

First of all a model of $Polynomial\_d$ is considered as an algebraic structure, that is, the ring operations +„àí„ãÖ are provided due to the fact that $Polynomial\_d$ refines at least the concept IntegralDomainWithoutDivision. However, a model of $Polynomial\_d$ has to be accompanied by a traits class $Polynomial\_traits\_d$<$Polynomial\_d$> being a model of $PolynomialTraits\_d$. This traits class provides all further functionalities on polynomials.

1.2.2 Convex Hull

A subset S of R^2 is convex if for any two points p and q in the set the line segment with endpoints p and q is contained in S. The convex hull of a set S is the smallest convex set containing S. The convex hull of a set of points P is a convex polygon with vertices in P. A point in P is an extreme point (with respect to P) if it is a vertex of the convex hull of P. A set of points is said to be strongly convex if it consists of only extreme points.



Each of the convex hull functions presents the same interface to the user. That is, the user provides a pair of iterators, first and beyond, an output iterator result, and a traits class traits. The points in the range [first, beyond) define the input points whose convex hull is to be computed. The counterclockwise sequence of extreme points is written to the sequence starting at position result, and the past-the-end iterator for the resulting set of points is returned. The traits classes for the functions specify the types of the input points and the geometric primitives that are required by the algorithms. All functions provide an interface in which this class need not be specified and defaults to types and operations defined in the kernel in which the input point type is defined.

The functions is_ccw_strongly_convex_2() and is_cw_strongly_convex_2() check whether a given sequence of 2D points forms a (counter)clockwise strongly convex polygon. These are used in post condition testing of the two-dimensional convex hull functions.

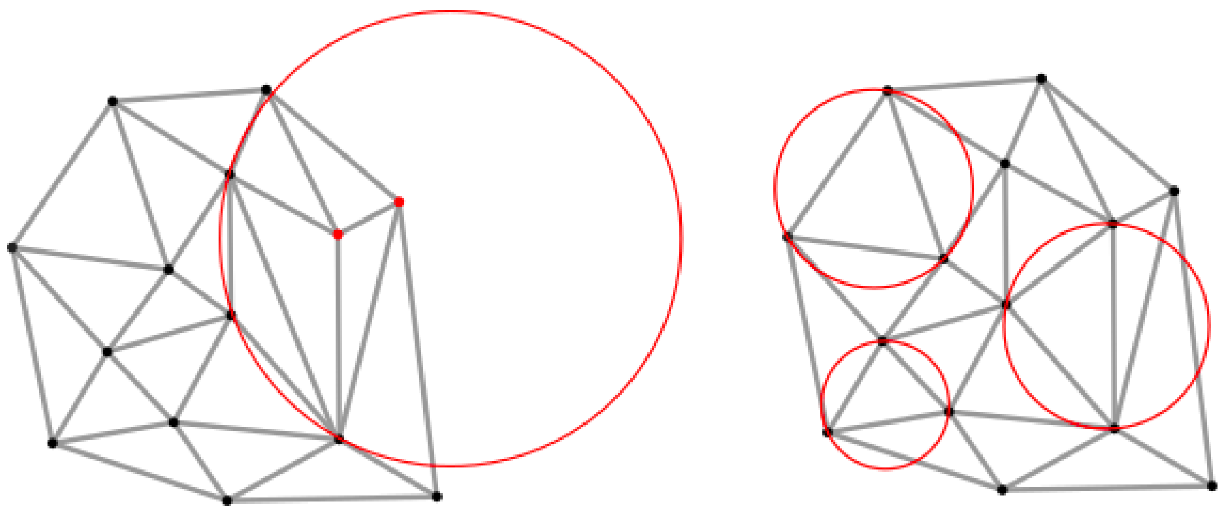
1.2.3 Polygons

A polygon is a closed chain of edges. Several algorithms are available for polygons. For some of those algorithms, it is necessary that the polygon is simple. A polygon is simple if edges don't intersect, except consecutive edges, which intersect in their common vertex.

The following algorithms are available:
1.$findtheleftmost, rightmost, topmostandbottommostvertex.$
2.$computethe(signed)area.$
3.$checkifapolygonissimple.$
4.$checkifapolygonisconvex.$
5.$findtheorientation(clockwiseorcounterclockwise)$
6.$checkifapointliesinsideapolygon.$

The type Polygon_2 can be used to represent polygons. Polygons are dynamic. Vertices can be modified, inserted and erased. They provide the algorithms described above as member functions. Moreover, they provide ways of iterating over the vertices and edges. Currently, the Polygon_2 class is a nice wrapper around a container of points, but little more. Especially, computed values are not cached. That is, when the Polygon_2::is_simple() member function is called twice or more, the result is computed each time anew.

1.2.4 Triangulations



A two dimensional triangulation can be roughly described as a set T of triangular facets such that:
1. two facets either are disjoint or share a lower dimensional face (edge or vertex).
2. the set of facets in T is connected for the adjacency relation.
3. the domain UT which is the union of facets in T has no singularity.

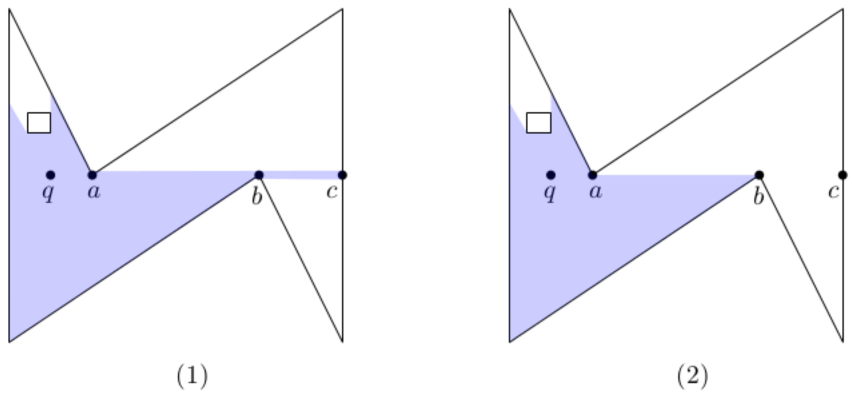
Each facet of a triangulation can be given an orientation which in turn induces an orientation on the edges incident to that facet. The orientation of two adjacent facets are said to be consistent if they induce opposite orientations on their common incident edge. A triangulation is said to be orientable if the orientation of each facet can be chosen in such a way that all pairs of incident facets have consistent orientations.

The data structure underlying CGAL triangulations allows the user to represent the combinatorics of any orientable two dimensional triangulations without boundaries. On top of this data structure, the 2D triangulations classes take care of the geometric embedding of the triangulation and are designed to handle planar triangulations. The plane of the triangulation may be embedded in a higher dimensional space.

Because a triangulation is a set of triangular faces with constant-size complexity, triangulations are not implemented as a layer on top of a planar map. CGAL uses a proper internal representation of triangulations based on faces and vertices rather than on edges. Such a representation saves storage space and results in faster algorithms [3]. The basic elements of the representation are vertices and faces. Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces.

1.2.5 2D Visibility

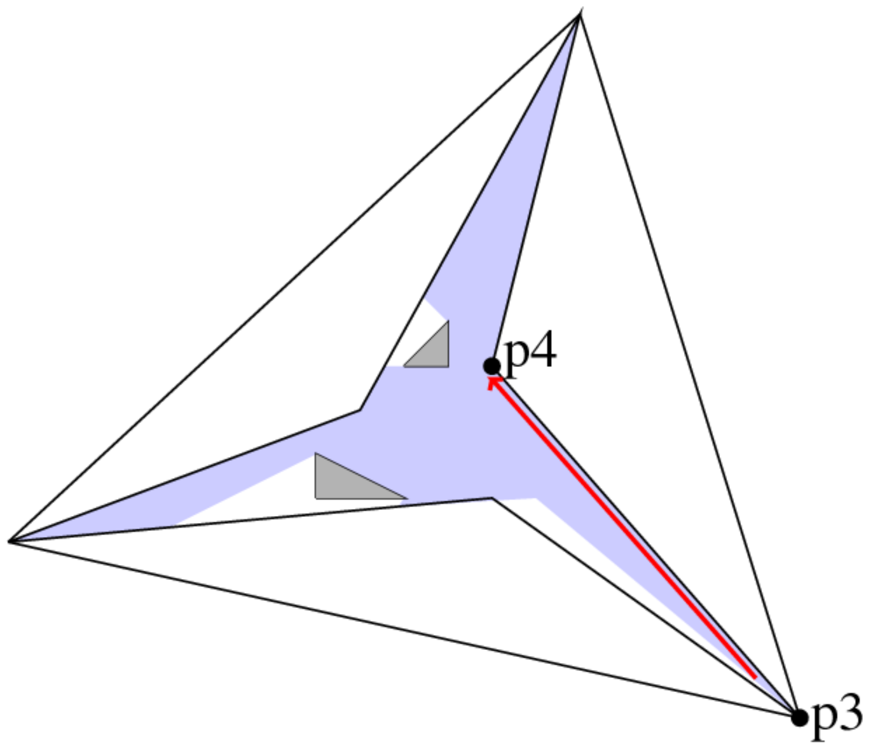
This package provides functionality to compute the visibility region within polygons in two dimensions. The package is based on the package 2D Arrangements and uses CGAL::Arrangement_2 as the fundamental class to specify the input as well as the output. Hence, a polygon P is represented by a bounded arrangement face f that does not have any isolated vertices and any edge that is adjacent to f separates f from another face. Note that f may contain holes. Similarly, a simple polygon is represented by a face without holes.



As illustrated in Figure the visibility region V_q of a query point q may not be a polygon. In the figure, all labeled points are collinear, which implies that the point c is visible to q, that is, the segment bc is part of V_q. We call such low dimensional features that are caused by degeneracies needles. However, for many applications these needles are actually irrelevant. Moreover, for some algorithms it is even more efficient to ignore needles in the first place. Therefore, this package offers also functionality to compute the regularized visibility area.

Answering visibility queries is, in many ways, similar to answering point-location queries. Thus, we use the same design used to implement 2D Arrangements point location. Each of the various visibility class templates employs a different algorithm or strategy for answering queries. Similar to the point-location case, some of the strategies require preprocessing. Thus, before a visibility object is used to answer visibility queries, it must be attached to an arrangement object. Afterwards, the visibility object observes changes to the attached arrangement. Hence, it is possible to modify the arrangement after attaching the visibility object. However, this feature should be used with caution as each change to the arrangement also requires an update of the auxiliary data structures in the attached object.

The following example shows how to obtain the regularized visibility region using the model Triangular_expansion_visibility_2, see Figure The arrangement has six bounded faces and an unbounded face. The query point q is on a vertex. The red arrow denotes the halfedge pq, which also identifies the face in which the visibility region is computed.



Shown in the figure visibility region of p4 to the polygon.

This can be implemented which gives the resultant visibility region as the polygon with holes. Given the art gallery problem such that polygon with holes was given as input and some location of random guards was also given, then we can check the visibility regions for each guard which would result in number guard numbered polygons with holes.

$bool join(const Type1 \& p1, const Type2 \& p2, General\_polygon\_with\_holes\_2 \& res);$

We need to find the union of all those polygons which can also be implemented with function signature as above. Further, we can make improvements to the heuristics in order to find the minimum guard set in case of point guards.

Chapter 2

Art Gallery Problem

2.1 Art Gallery Implementation with high precision

2.1.1 Introduction

The art gallery problem or museum problem is a well-studied visibility problem in computational geometry. It originates from a real-world problem of guarding an art gallery with the minimum number of guards who together can observe the whole gallery. In the computational geometry version of the problem the layout of the art gallery is represented by a simple polygon and each guard is represented by a point in the polygon. A set S of points is said to guard a polygon if, for every point p in the polygon, there is some $q \in S$ such that the line segment between p and q does not leave the polygon.

There are numerous variations of the original problem that are also referred to as the art gallery problem. In some versions guards are restricted to the perimeter, or even to the vertices of the polygon. Some versions require only the perimeter or a subset of the perimeter to be guarded. Solving the version in which guards must be placed on vertices and only vertices need to be guarded is equivalent to solving the dominating set problem on the visibility graph of the polygon.The question about how many vertices or watchmen or guards were needed was posed to $Chvatal$ by Victor Klee in 1973[4]. $Chvatal$ proved it shortly thereafter.[1] $Chvatal's$ proof was later simplified by Steve Fisk, via a 3-coloring argument.[2]

2.1.2 Objective of our Implementation

Given a polygon (with or with out holes in it) and also set of vertices which are included in the polygon (on the outer boundary or with in any of the holes) as input , we need to find whether the given set of vertices satisfy the guard set or not. Solving this problem is same as the solving the art gallery problem where the dimensions of the art gallery can be assumed to be polygons with holes and the cameras or guards placed are assumed to be the set of vertices given as input and we need to find whether or not the given guard set cover the entire polygon or not.

2.1.3 Methods of Implementation and Program

Considering the above objective as the final goal for our implementation, we follow certain methods to program it using $CGAL$. The polygon with holes that we are giving as input was stored in the data structure $polygon\_with\_holes\_2$ in cgal library. Now, given a vertex in this polygon we can able to find the visibility region from the vertex using $compute\_visibility$ function which would again give us the visibility region as polygon with holes. Likewise, we compute the visibility region for every vertex and collect the set of polygon with holes. The visibility region covered by all the vertices is represented as $UnionR$ which is union of all the above visibility regions which is also polygon with hole. Now, we compute the symmetric difference of $UnionR$ polygon and the original polygon which if null represents guard set satisfies, doesn't satisfy otherwise.

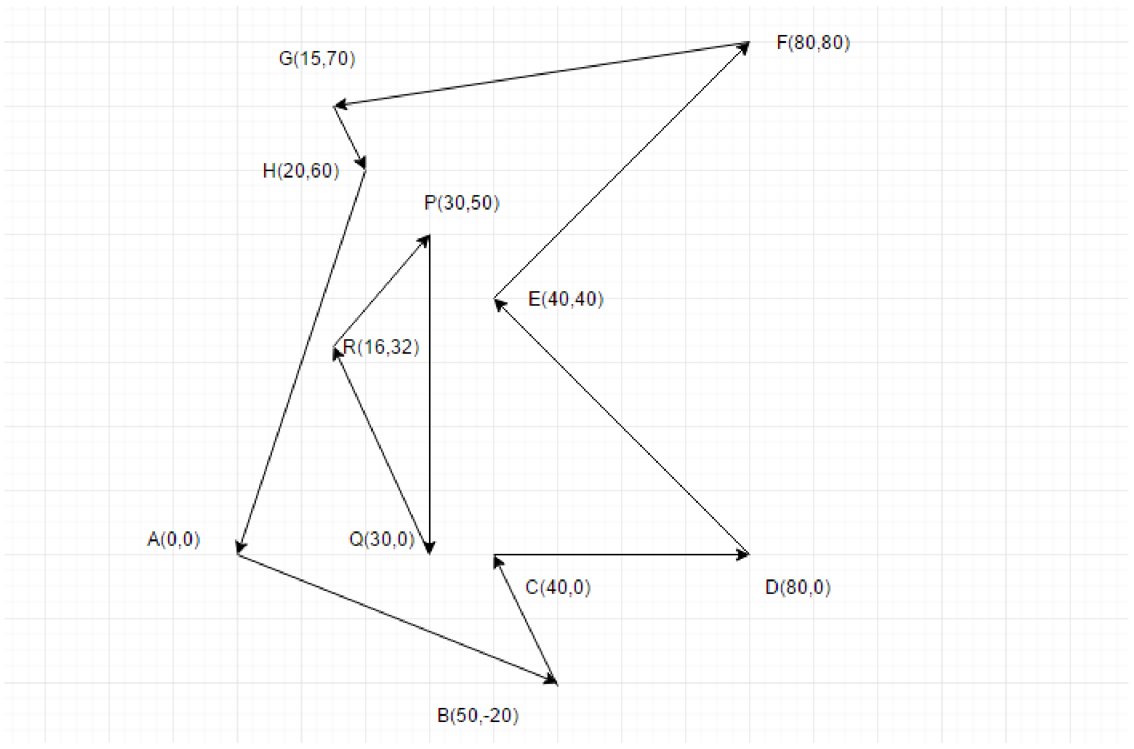
The above method is implemented in following way:

```
#include <CGAL/Exact_predicates_exact_constructions_kernel.h>
#include <CGAL/Triangular_expansion_visibility_2.h>
#include <CGAL/Arr_segment_traits_2.h>
#include <CGAL/Arrangement_2.h>
#include <CGAL/Boolean_set_operations_2.h>
#include <CGAL/Polygon_2.h>
#include <CGAL/Polygon_with_holes_2.h>
#include <iostream>
#include <vector>
// Define the used kernel and arrangement
typedef CGAL::Exact_predicates_exact_constructions_kernel Kernel;
typedef Kernel::Point_2                                   Point_2;
typedef CGAL::Polygon_2<Kernel>                           Polygon_2;
typedef Kernel::Segment_2                                 Segment_2;
typedef CGAL::Arr_segment_traits_2<Kernel>                Traits_2;
typedef CGAL::Arrangement_2<Traits_2>                     Arrangement_2;
typedef CGAL::Polygon_with_holes_2<Kernel>                Polygon_with_holes_2;
typedef Arrangement_2::Halfedge_const_handle              Halfedge_const_handle;
typedef Arrangement_2::Face_handle                    Face_handle;
typedef std::list<Polygon_with_holes_2>               Pwh_list_2;

#include "print_utils.h"
// Define the used visibility class
typedef CGAL::Triangular_expansion_visibility_2<Arrangement_2> TEV;

int main() {
  // Defining the input geometry
  std::vector<Segment_2> segments;
  int j,a,b,c,d,i,n,qn,qj,holes,points;
  Point_2 x,y;
  Polygon_2 outer;
  std::cout<<"give the outer polygon size: ";
  std::cin>>n;
  std::cout<<"enter then number of holes: ";
  std::cin>>holes;
  std::cout<<"Enter the each segment of outer polygon:"<<std::endl;
  Point_2 p[1000];
  std::vector<Polygon_2> visiPolygons;
  for(i=0;i<n;++i){
    std::cin>>x;
    outer.push_back (x);
    std::cin>>y;
    Point_2 p1=x,p2=y;
    p[i] = p1;
    p[(i+1)%n] = p2;
    segments.push_back(Segment_2(p[i],p[(i+1)%n]));
  }

  std::vector<Polygon_2> holes_array;
  i = 0;j = n;
  // Orientation of the holes is designed in the program
  // in a way that holes which have higher y co-rdinate wil be given first
  // starting with highest y in a clockwise manner
  // if y-cordinates are same then consider greatest x first
  for(i=1;i<=holes;++i){
        Polygon_2 add_hole;
        std::cout << "\nEnter the number of points on boundary of hole #" << i << ":
            ";
        std::cin>>points;
        std::cout << "\nEnter the points in outer bounday in cw starting with largest
            y : ";
```

```
      std::cin>>x;
      add_hole.push_back(x);
      Point_2 temp1=x;
      p[j] = temp1;
      ++j;--points;
      Point_2 temp2,start = temp1;
      while(points--){
      std::cin>>temp2;
      add_hole.push_back(temp2);
      p[j] = temp2;
      ++j;
      segments.push_back(Segment_2(temp1, temp2));
      temp1 = temp2;
      }
      segments.push_back(Segment_2(temp2, start));
      holes_array.push_back(add_hole);
}
Arrangement_2 env;
CGAL::insert_non_intersecting_curves(env,segments.begin(),segments.end());

Polygon_with_holes_2 original(outer,holes_array.begin(),holes_array.end());
//Polygon_with_holes_2 original = get_polygon(env, holes);

std::cout<<"Enter the number of query points: ";
std::cin>>qj;
qn = qj;
while(qn--){
Polygon_2 polygon;
std::cout<<"Enter the query point number: ";
std::cin>>j;
Point_2 query_point = p[j];
Halfedge_const_handle he = env.halfedges_begin();
while (he->target()->point() != query_point || he->face()->is_unbounded() )
  he++;
Arrangement_2 output_arr;
TEV tev(env);
Face_handle fh = tev.compute_visibility(query_point, he, output_arr);

//print out the visibility region.
std::cout << "Regularized visibility region of q has "
         << output_arr.number_of_edges()
         << " edges." << std::endl;
  std::cout << "Boundary edges of the visibility region:" << std::endl;
  Arrangement_2::Ccb_halfedge_circulator curr = fh->outer_ccb();
  polygon.push_back(Point_2(curr->source()->point()));
  std::cout << "[" << curr->source()->point() << " -> " << curr->target()->point()
      << "]" << std::endl;
  while (++curr != fh->outer_ccb()){
    polygon.push_back(Point_2(curr->source()->point()));
    std::cout << "[" << curr->source()->point() << " -> " << curr->target()->point()
        << "]"<< std::endl;
  }
  visiPolygons.push_back(polygon);
  }
  Polygon_with_holes_2 unionR;
  if (CGAL::join (visiPolygons[0], visiPolygons[0], unionR)) {
    std::cout << "The union: ";
    print_polygon_with_holes (unionR);
  }
  for(i=1;i<qj;++i){
   if (CGAL::join (unionR, visiPolygons[i], unionR)) {
    std::cout << "The union: ";
    print_polygon_with_holes (unionR);
  }
  }
  std::cout<<"Final union of all polygons:"<<std::endl;
  print_polygon_with_holes (unionR);
  std::cout<<"Original polygon:"<<std::endl;
  print_polygon_with_holes (original);
  Pwh_list_2 symmR;
  Pwh_list_2::const_iterator it;
  CGAL::symmetric_difference (original, unionR, std::back_inserter(symmR));
  if(symmR.size() == 0){
    std::cout<<"Vertex Guard set holds"<<std::endl;
  }
  else{
    std::cout << "\n\nThe symmetric difference:" << std::endl;
    for (it = symmR.begin(); it != symmR.end(); ++it) {
      std::cout << "--> ";
      print_polygon_with_holes (*it);
    }
    std::cout<<"Vertex Guard set is invalid"<<std::endl;
  }
  return 0;
}
```

2.1.4 Correctness and Precision

The correctness and preciseness of above method can be understood by using the following example:



Consider the above Polygon with 1 hole as the input to the polygon.we can see from the figure that points A,C,D and D,E,H are collinear. so, now when we take the guard set as the vertices AA,D,H the guard set must satisfy. Now, the precision of the program can be checked by varying the input polygon very slightly around 10$\hat{(}$-40) shift in point A which will make A,C,D non-collinear. The slight increase in y-cordinate of the point A will result in unsatisfiability of the guard set. This was shown in below output.

with vertex A as (0,0)

```
Boundary edges of the visibility region:
[40 0 -> 80 0]
[80 0 -> 40 40]
[40 40 -> 30 50]
[30 50 -> 30 0]
[30 0 -> 40 0]
Enter the query point number: Regularized visibility region of q has 8 edges.
Boundary edges of the visibility region:
```

```
[20 60 -> 0 0]
[0 0 -> 50 -30]
[50 -30 -> 40 0]
[40 0 -> 30 0]
[30 0 -> 16 32]
[16 32 -> 36.6667 73.3333]
[36.6667 73.3333 -> 23.7838 71.3514]
[23.7838 71.3514 -> 20 60]
Enter the query point number: Regularized visibility region of q has 8 edges.
Boundary edges of the visibility region:
[15 70 -> 20 60]
[20 60 -> 0 0]
[0 0 -> 10.5263 -6.31579]
[10.5263 -6.31579 -> 16 32]
[16 32 -> 30 50]
[30 50 -> 40 40]
[40 40 -> 80 80]
[80 80 -> 15 70]
The union: { Outer boundary = [ 5 vertices: (30 0) (40 0) (80 0) (40 40) (30 50) ]
  0 holes:
 }
The union: { Outer boundary = [ 11 vertices: (16 32) (36.6667 73.3333) (23.7838
    71.3514) (20 60) (0 0) (50 -30) (40 0) (80 0) (40 40) (30 50) (30 0) ]
  0 holes:
 }
The union: { Outer boundary = [ 11 vertices: (23.7838 71.3514) (15 70) (20 60) (0 0)
    (10.5263 -6.31579) (50 -30) (40 0) (80 0) (40 40) (80 80) (36.6667 73.3333) ]
  1 holes:
    Hole #1 = [ 3 vertices: (30 50) (30 0) (16 32) ]
 }
Final union of all polygons:
{ Outer boundary = [ 11 vertices: (23.7838 71.3514) (15 70) (20 60) (0 0) (10.5263
    -6.31579) (50 -30) (40 0) (80 0) (40 40) (80 80) (36.6667 73.3333) ]
  1 holes:
    Hole #1 = [ 3 vertices: (30 50) (30 0) (16 32) ]
 }
Original polygon:
{ Outer boundary = [ 8 vertices: (0 0) (50 -30) (40 0) (80 0) (40 40) (80 80) (15
    70) (20 60) ]
  1 holes:
    Hole #1 = [ 3 vertices: (30 50) (30 0) (16 32) ]
 }
Vertex Guard set holds
```

With vertex A slightly shifted up

```
give the outer polygon size: enter then number of holes: Enter the each segment of
    outer polygon:

Enter the number of points on boundary of hole #1:
Enter the points in outer bounday in cw starting with largest y : Enter the number
    of query points: Enter the query point number: Regularized visibility region of
    q has 5 edges.
Boundary edges of the visibility region:
[40 0 -> 80 0]
[80 0 -> 40 40]
[40 40 -> 30 50]
[30 50 -> 30 0]
[30 0 -> 40 0]
Enter the query point number: Regularized visibility region of q has 8 edges.
Boundary edges of the visibility region:
[20 60 -> 0 1e-41]
[0 1e-41 -> 50 -30]
[50 -30 -> 40 -3.33333e-42]
[40 -3.33333e-42 -> 30 0]
[30 0 -> 16 32]
[16 32 -> 36.6667 73.3333]
[36.6667 73.3333 -> 23.7838 71.3514]
[23.7838 71.3514 -> 20 60]
Enter the query point number: Regularized visibility region of q has 8 edges.
Boundary edges of the visibility region:
[15 70 -> 20 60]
[20 60 -> 0 1e-41]
[0 1e-41 -> 10.5263 -6.31579]
[10.5263 -6.31579 -> 16 32]
[16 32 -> 30 50]
[30 50 -> 40 40]
[40 40 -> 80 80]
[80 80 -> 15 70]
The union: { Outer boundary = [ 5 vertices: (30 0) (40 0) (80 0) (40 40) (30 50) ]
  0 holes:
 }
The union: { Outer boundary = [ 13 vertices: (16 32) (36.6667 73.3333) (23.7838
    71.3514) (20 60) (0 1e-41) (50 -30) (40 -3.33333e-42) (30 0) (40 0) (80 0) (40
    40) (30 50) (30 0) ]
  0 holes:
 }
The union: { Outer boundary = [ 13 vertices: (23.7838 71.3514) (15 70) (20 60) (0
    1e-41) (10.5263 -6.31579) (50 -30) (40 -3.33333e-42) (30 0) (40 0) (80 0) (40
    40) (80 80) (36.6667 73.3333) ]
  1 holes:
    Hole #1 = [ 3 vertices: (30 50) (30 0) (16 32) ]
 }
Final union of all polygons:
{ Outer boundary = [ 13 vertices: (23.7838 71.3514) (15 70) (20 60) (0 1e-41)
    (10.5263 -6.31579) (50 -30) (40 -3.33333e-42) (30 0) (40 0) (80 0) (40 40) (80
    80) (36.6667 73.3333) ]
  1 holes:
    Hole #1 = [ 3 vertices: (30 50) (30 0) (16 32) ]
 }
Original polygon:
{ Outer boundary = [ 8 vertices: (0 1e-41) (50 -30) (40 0) (80 0) (40 40) (80 80)
    (15 70) (20 60) ]
  1 holes:
    Hole #1 = [ 3 vertices: (30 50) (30 0) (16 32) ]
 }


The symmetric difference:
--> { Outer boundary = [ 3 vertices: (30 0) (40 -3.33333e-42) (40 0) ]
  0 holes:
 }
Vertex Guard set is invalid
```

The above output clearly depicts the precision of the solvability of the implementation.

BIBLIOGRAPHY

[1] V. CHVATAL, *A combinatorial theorem in plane geometry", journal of combinatorial theory*, 18 (1975), pp. 39–41.

[2] S. FISK, *A short proof of chvatal's watchman theorem", journal of combinatorial theory*, 24 (1978), pp. 374–375.

[3] M. T. JEAN-DANIEL BOISSONNAT, OLIVIER DEVILLERS AND M. YVINEC., *Triangulations in cgal*, 16 (2000), pp. 11–18.

[4] J. O'ROURKE, *Art gallery theorems and algorithms, oxford university press*, (1987).