VisPoly

A Software of Visibility Graph with Multiple Reflection and its Application of Wireless Communication Design

Author: Min Fan

Advisor: Diane Souvaine

April 23, 1998

Abstract:

***VisPoly* is geometric software designed and implemented by Min Fan to calculate the visibility graph of a simple polygon with k reflections. The current version requires that software package LEDA installed. This report will cover the algorithm to compute a k-reflection visibility graph, the applications on wireless communication design, and the author's ideas for further research in this field.**

1 Introduction:

Visibility-related problems have been extensively studied, and there are hundreds of papers devoted to this area. Visibility topics include art gallery problems, visibility graphs, computation and characterization. This project is related to the computing of the visibility polygon and characterization. According to [OR97], designing algorithms to compute aspects of visibility was a major focus of the computational geometry community in the 1980's. And, for most of the basic problems, optimal algorithms were found. For more information, see [OR97].

Remarkably, visibility with reflection has not been extensively investigated. A recent result on the geometric complexity of visibility with multiple reflections is [ADD+95]. In this paper, Aronov et al. concluded that the region lit by a point light source inside a simple n-gon after at most k reflections off the boundary has combinatorial complexity at most $O(n^{2k})$ for any $k \geq 1$. Furthermore, there are instances that achieve a complexity of $\Omega((n/2k)^{2k})$, making that a lower bound on the complexity of the problem itself. For any fixed k, the lower bound matches the upper bound. In their paper they also presented a simple near-optimal algorithm to calculate the boundary of the visibility region, which runs in $O(n^{2k} \log n)$ time and $O(n^{2k})$ space [ADD+95].

VisPoly is inspired by [ADD+95] to calculate the visibility graph of a simple polygon with k reflections. First an algorithm based on [EA81] mentioned in [ADD+95] to calculate the visibility graph without reflection in linear time was implemented. Then visibility regions with at most k reflections are computed. Finally *VisPoly* provides a graphical user interface to answer questions such as whether the query point is visible, what is the shortest path from the light source to the query point, and which areas are covered after 1 reflection, 2 reflections, etc.

VisPoly is written in C++ and using LEDA (Library of Efficient Data types and Algorithms) as its object library. The current version of *VisPoly* is built on the X-window system on a *solaris* platform. It can also be ported to other UNIX system with X-window display (see [LEDA], [Nutshell]).

The remainder of this paper is organized as follows. Section 2 presents the algorithms used in *VisPoly*. Section 3 discusses the implementation issues. Section 4 describes the

possible applications to wireless communication system design using the visibility graph with k reflections. Section 5 covers the possible enhancement of the algorithm to solve more difficult problems. Appendix A and B provide a user's manual and an illustrative example for an algorithm in section 2, respectively. Finally, Appendix C lists the major source code.

2 Visibility graph with k reflections

2.1 Definitions

Here we introduce some basic concepts involved in the problem. We are given a simple polygon *P* with n vertices, and a point inside the polygon *s* as a light source. A point *x* is said visible from point *y* if the line segment *xy* lies entirely inside *P*. The set of points visible from *s* forms the **visibility region** of *P*. The visibility region should be a simple polygon. For a given edge e_i , a portion of it visible from *s* is called a **mirror** m_i . Each mirror generates a mirror image s^i of the light source, which is the point on the line through s perpendicular to m_i from s and equidistant to m_i with *s*. (see, e.g. [OR96], [ADD95],[Physics]).

Figure 1. Visibility region of a given simple polygon and a light source. The visibility region is shown as the shaded area. e_i is an edge, and m_i is the associated mirror.

The above visibility region is known as the **direct visible polygon**, or the **visibility polygon without reflection**. The rays reflected at a mirror form the **reflected visibility region**. When a ray hits a mirror, it will be bounced off in such a way that the angle of incidence equals the angle of reflection (Shell's law) (see [Physics] for details of Shell's law). The region lit up by the rays bouncing from the same mirror is called the reflected visibility region with respect to that mirror. How can we calculate a reflected visibility region? As we already have shown, a mirror image of the light source is generated for each mirror. The rays reflected by that mirror are equivalent to the rays emitted directly from the mirror image s^i . Hence by

calculating the direct visibility region of the mirror image we get a reflected visibility region after 1 reflection. The visibility region for other mirrors after 1 reflection, 2 reflections etc. can be done by using similar methods (see [ADD95]).

Figure 2: Visibility region associated with mirror m_i after one reflection. The region is shown as shaded area.

From the above description of the problem, we can see there are lots of complex data types involved in it as well as some basic operations on those data types that are not provided by C++ standard library (see, e.g. [C++]). For examples, the complex data types include points, segments, lines, polygons, stacks, etc (see, e.g. [Algo]). The operations include calculating the intersection of lines and polygons, etc. Fortunately, there is a software package LEDA (Library of Efficient Data types and Algorithms) distributed by Max-Planck Institut, Saarbruecken that provides the capability we need. In the next section we take a look at some features of LEDA (see [LEDA]).

2.2 Object library LEDA

LEDA is an object library of combinatorial computing. It is implemented in C++. As an object oriented software library, it provides data types as well as algorithms, and encapsulates the data types and the associated algorithms in objects (see [LEDA]).

The objects provided by LEDA are comprehensive. LEDA has simple objects such as string, array, but also has complex objects useful for graph theoretic and geometric problems. The LEDA project started ten years ago, and is still under aggressive development. As the difference of different versions of LEDA is not negligible, software developed on LEDA may work with one version and fail when using another version of LEDA. *VisPoly* works with LEDA 3.6 [LEDA].

The objects such as list, stack, and polygon are fundamental in the implementation of *VisPoly*. *VisPoly* also uses member functions such as point inclusion and polygon intersection (see, e.g. [OR96]). LEDA also provides a display system, which gives a vivid display of the results and hence speeds up the developing process [LEDA].

2.3 Description of algorithms

Here we describe the main flow of processing steps in *VisPoly*.

First we calculate the direct visible region without reflection within a simple polygon (see, e.g. [OR96]). The algorithm presented by [EA81] is a linear time algorithm, and hence it is an optimal algorithm. This is the algorithm used in *VisPoly*. The general description is given in Section 2.4.

The next step is to calculate the visibility regions with one reflection. In order to calculate the visibility regions with one reflection, we need to calculate a mirror image s^i relative to mirror m_i of the light source s first (see, e.g.[Physics]). Then we can calculate the visibility region associated with that mirror image s^i. Let us denote Δ^i as the triangle formed by s^i and two end points of m_i. By gluing the triangle with polygon P, we get a polygon which is the union of P and Δ^i. The reflected visibility region $V_i(s) = P \cap V(s^i, P \cup \Delta^i)$, where $V(s^i, P \cup \Delta^i)$ is the direct visibility region from s^i in polygon $P \cup \Delta^i$, where $P \cup \Delta^i$ denotes here the "**Riemann surface**"(see [RS]) obtained by gluing P to Δ^i along m_i, with the possibility that , in the plane, Δ^i would overlap P. Hence we can first calculate the polygon $P \cup \Delta^i$, and then reuse the standard algorithm in [EA81] to calculate the visibility region from s^i with one reflection in polygon $P \cup \Delta^i$ [ADD95].

The algorithm in [EA81] requires the original polygon to be **simple**, and the source to be **inside** of that polygon. In the case of calculating visibility regions with one reflection, s^i is **not inside** of $P \cup \Delta^i$, but is a **vertex** of it. Moreover, the polygon P and Δ^i may overlap. For example, refer to Figure 3. [EA81] cannot handle the case of overlapping. So we need to do some preprocessing in order to solve these two problems. This preprocessing in calculating the reflected visibility region will be presented in section 2.5.

Figure 3: Example of Δ^i that overlaps P.

Finally, the above strategy can be repeated to handle the problem of calculating visibility regions with k reflections.

2.4 Linear algorithm for visibility region without reflection

Let *P* be the given simple polygon, and *s* be the light source. The idea behind [EA81] is to visit all the vertices of *P* in clockwise order. While walking through the vertices of *P*, we detect of all the vertices that are visible from *s*. We can tell whether a vertex is visible or not by analyzing the position of the nearby vertices. If we cannot conclude whether a vertex is visible, we put it in temporary storage, and search forward or backtrack to get more information. See [EA81] for details.

2.5 Algorithm for preprocessing in calculating visibility region after reflection

After calculating the direct visible polygon, we know which edge e_i is visible from the source *s* and which portion of e_i is actually visible from *s* (known as a **mirror**). The mirror is denoted by m_i [ADD95].

As we have known, when gluing Δ^i to *P*, the $P \cup \Delta^i$ may not be a simple polygon. We will describe two methods that can be used to preprocess $P \cup \Delta^i$ in order to use [EA81] iteratively to calculate the visibility polygon after reflections. Consider the example shown in figure 4. *P* is a simple polygon, but *P* and Δ^i may overlap. In order to use [EA81], we need to discard the overlapping portion first. Unfortunately, there is no simple rule that can tell which part of *P* causes the overlapping. We found out that the reflected visibility region can be considered as the part of P lit up by the rays emitting from s^i which are limited by the two side edges of Δ^i . s^i and two side edges of Δ^i form a **cone (**see, e.g.[OR96]). Any part of *P* that lies outside of the cone will not be lit up. Therefore, if we discard the part outside of the cone, the reflected visibility region will not be changed.

If we just keep the portion of *P* that is inside of cone, we may come up with several disjoint polygons, only one of which is the one we need. In the example in Figure 4, we have three polygons: *P1*, *P2*, and *P3*. *P1* needs to be discarded, as it is the part causing the overlapping. *P3* is not connected with Δ^i and needs to be discarded too. Only *P2* should be kept. Then by calculating the visibility region in $P2 \cup \Delta^i$, we get the correct reflected visibility region.

Figure 4: An example of discarding the overlapping portion of P and Δ^i.

Two approaches to eliminate the overlapping area are discussed in the following sections. Approach 1 can be easily implemented using LEDA, but it is not an optimal approach, as its time complexity is O(*n log n*). The time complexity for approach 2 is O(*n*), which is optimal. As approach 2 requires complex book-keeping, it is not used in *VisPoly*. As approach 2 is designed by the author and may be implemented in later on, it is kept in this paper as a comparison to approach 1. Both of the two algorithms (approach 1 and approach 2) solve the same problem: discarding the overlapping part of P and Δ^i.

2.5.1 Approach 1: Discard the polygon outside the range of cone by using polygon intersection.

The object type polygon is provided in LEDA. It has the member function "polygon intersection", which can calculate the intersection of two polygons. The prototype is the following.

list<polygon> *P*.intersection(polygon *Q*).

Polygon *P* and polygon *Q* in the member function above are required to be simple polygons. What we have now is an unbounded polygon *P* and a cone. Let us first bound the cone with an edge which is far enough away so as not to interfere the result of intersection (see Figure 5 for examples).

Figure 5. Bounding a cone by adding an edge in order to use polygon intersection function.

The algorithm follows.

1. Bound the cone by adding an edge to the cone. Denote the result triangle as Q.
2. Call function P.intersection(Q). The result is a list of polygons that contains the one we needed.
3. Pick the polygon that shares m_i with Δ^i in the list. Denote it as R.
4. Calculate $R \cup \Delta^i$. We are then ready to use [EA81] except for the light source is a vertex of $R \cup \Delta^i$.

This algorithm is based on polygon intersection, which uses the sweep-line algorithm (see [BO79] for details). The time complexity is $O(n \log n)$. We suffered a $\log n$ penalty.

2.5.2 Approach 2: Discard the portion outside of the cone by traversing the vertices.

The time complexity for approach 1 is O($n \log n$), which is not optimal. We should be able to find the right polygon within the cone by traversing the vertices of polygon P. The following is an algorithm designed by the author with linear time complexity.

In this approach, we examine each vertex in turn, and determine whether a vertex belongs to the polygon that shares the edge with Δ^i. Hence in this approach the discarding the portion outside of the cone and picking the polygon which shares the edge with Δ^i are done simultaneously.

Recall the problem in Figure 4. First, order the vertices of P in clockwise order. The mirror m_i has two end points. The point first visited in clockwise traversal is call the **left end point,** denoted by x, and the other endpoint is called the **right end point of** m_i,

denoted by *y*. Next assume the mirror image s^i of the source *s* is sitting above the mirror m_i. The ray from s^i passing through *x* is called the left ray *l*, and the ray starting from s^i passing through *y* is called the right ray *r*.

The algorithm to cut off the portion outside of the cone is done in two steps. Assuming we have scissors in hand, we cut the polygon starting from the right end point *y* of mirror m_i along the right ray *r*. The part of polygon not attached to mirror m_i is generally to the right of the right ray *r*, and is labeled as such. As we only need the part of polygon *P* with m_i as an edge, the part of polygon "to the right of the right ray *r*" is discarded. We do the same operation to the left ray, i.e. cut from *x* along the direction of *l*, and discard the part of polygon *P* "to the left of left ray *l*".

The two steps are then:

1. Cut off the portion of polygon *P* that is to the right of the right ray of the cone. Denote the resulting polygon as *Q*.
2. Cut off the portion of *Q* that is to the left of the left ray of the cone. Denote the resulting polygon as *R*. Polygon *R* is the part of *P* without the problem of overlapping.

The algorithm to cut off a portion of polygon "to the left of a ray", and "to the right of a ray" are similar. So only the algorithm to cut off the portion to the right of a ray is presented below. Although the idea of cutting along the ray is simple, the algorithm to realize the idea is not trivial, as bookkeeping of global information is needed. As there is not a simple example that can show the whole idea of this algorithm, a **step by step illustrative example is shown in Appendix B.**

We start from the vertex z_1. z_1 is the first vertex on the list of vertices of polygon *P* right after *y* (the right end point of mirror m_i) in clockwise order. We make z_1 the current visiting point. z_1 is tested again ray *r* to find out whether it is on the same side of ray *r* with mirror m_i. If the current visiting point is on the same side, it is in the range of the cone, otherwise it is to the right of ray *r*. Therefore z_1 should be discarded. We keep the status of the current visiting point in a global variable **S** to show whether current visiting point is inside or outside of the range of cone. As we walk through the vertices of polygon *P*, we cross the ray *r* many times. Each time we cross ray *r*, the status of the current visiting point switches between "in" and "out".

The temporary stack **ST** keeps a chain (see [EA81] for definition of chain) of vertices which could possibly bound the final polygon. We will not have complete information for polygon *Q* until every vertex of *P* is visited. Hence, **ST** may contain some points that will need to be removed when more information is known. We solve the problem by backtracking: when we detect a vertex that should not be kept in **ST**, we need to pop it out of the stack. The backtracking is done in the last portion of Case 2 in the procedure Poly-Cut. The backtracking is not easy to understand at first, but a real example makes it

clear. Please refer to illustrative example in Appendix B to understand backtracking in details.

The procedure Poly-Cut is given in the following paragraphs. Explanations will be given in brackets.

Input: A mirror m :x→y . x is the left endpoint, and y is the right end point.
Origin of the ray: o. {Note: o is the mirror image of *s* with respect to m}
A circular linked list z_1 , succ(z_1), succ(succ(z_1)), … , z_n .

Method: The algorithm examines the vertices of P in clockwise order, updating a stack and other global variables.

Stack **ST** contains possible vertices to the left of the cutting ray that belong to the polygon containing the mirror.

Status **S** has a value either "in" or "out", which identifies whether the examined was inside or outside the area of cutting ray.

Point t is the farthest point reached along the ray o→ y.

Vector d_0 represents the direction of the ray o→ y; d_1 represents the opposite direction, and so its direction is y→ o.

When each vertex of polygon P is examined, the points in ST are the vertices of polygon left by cutting off the portion of P on the right of the ray with the mirror as an edge.

Procedure Poly-Cut

Initialization step:

If z_1 is to the right of the ray, S ← out;
else S ← in.
Push(ST) ← z_1
Last point : u ← z_1
t ← z_1

General step:

Case 0. succ(u) = z_1.

STOP.

{Note: All vertices of P are visited, halt}

Case 1. S = out

Search from u for a segment that intersects the ray starting at t and with the direction d_0 at a point k. Let the segment be wv, where w is visited first in clockwise order.

Push(ST) ← k, v

S ← in

t ← k

Case 2. S = in

v ← succ(u)

while (segment uv does not intersect the ray from y with direction d_0)

{

Push(ST) ← v

u ← v

v ← succ(u)

}

If (v = z_1)

STOP

If edge uv intersects ray from t with direction d_0 at a point k

{

Push(ST) ← k

S ← out

u ← v

t ← k

}

If segment uv intersect segment yt at a point k,

Search for the next segment qp that intersects the ray from k with direction d_1 at a point w.

{

Pop(ST) all points until t

t ← w

Push(ST) ← w, p

u ← p

S ← in

}

end

As each vertex is visited once: it is pushed on the stack at most once, and it is popped off the stack at most once, this algorithm is a linear-time algorithm. As the above algorithm involves quite complex bookkeeping, it is not easy to prove its correctness, and it is possible that not all polygonal shapes are handled correctly. Therefore, this algorithm is not adopted in *VisPoly*.

VisPoly uses a safer approach, i.e. approach 1 using polygon intersection. This implementation changes the complexity from linear time to $O(n \log n)$. On the other hand, if we look at the total complexity of the algorithm for k reflections, this penalty may not hurt the performance a lot. The reason is the following. Our goal is to calculate all visibility polygons within at most k reflections. We started with one polygon. We first calculate the direct visibility polygon, and then calculate the visibility polygon with one reflection. The direct visibility polygon has up to n mirrors, and each mirror can generate a visibility polygon. Therefore after one reflection, we can have O(n) polygons. Similarly, after another reflection, a total of $O(n^2)$ polygons may appear. Hence the number of polygons at reflection level k is $O(n^k)$. The summation of the total number of polygon at all the levels is $O(n^{k+1})$. The time complexity to calculate one visibility polygon using polygon intersection as preprocessing (see Section 2.5.1) is $O(n \log n)$. The complexity of calculating $O(n^{k+1})$ visibility polygons using polygon intersection is $O(n^{k+2} \log n)$. The polynomial in n is more significant than the $\log n$. Therefore we did not suffered too great a penalty by not using the optimal algorithm. Also the fundamental algorithm in approach 1 is the polygon intersection algorithm, which is provided by LEDA. LEDA is very reliable software. We generally have fewer problems if we reuse good code.

2.6 Algorithm to preprocess step 2: moving the light source from a vertex to an internal point.

We can use either approach presented in Section 2.5 to discard the overlapping part by discarding all parts of the polygon outside of the cone. Once we discard the part of the polygon outside of the cone, we then need to fix the polygon so that the source will be an internal point rather than a vertex, as required by [EA81]. *VisPoly* expands the triangle Δ^i to make the source point an internal point while keeping the reflected visibility polygon unchanged. In fact if we move the vertex s^i further away from the mirror edge m_i, the reflected visibility polygon is not affected. This observation is illustrated in the figure 6.

Figure 6: Expand the triangle to move the source internal

After the two preprocessing steps described above, we are ready to apply the algorithm of direct visibility [EA81] to calculate the reflected visibility polygons. When we get the reflected visibility polygon, the glued triangle is definitely visible from the reflected source light. The next step is to remove the attached triangle in order to form the reflected polygon with respect to one mirror (see Section 2.1 for definitions). This step is trivial, the detail is omitted here (see Section 4.1 for details).

2.7 Data structure used to keep the information of the reflected visibility polygons.

After k reflections we could have $O(n^{k+1})$ visibility polygons . There is one directly visible simple polygon; there are up to n visibility polygons at one reflection (up to n mirrors may be generated by the directly visible simple polygon), and so there are $O(n^k)$ visibility polygons at k reflections. The total number of polygons after k reflections is the summation of number of polygons at each reflection level, which is $O(n^{k+1})$. The time complexity of the algorithm used in *VisPoly* is $O(n^{k+2} \log n)$. We have a huge number of polygons to handle even for moderate number of reflections. Therefore the problem of bookkeeping for the reflected visibility polygons is non-trivial. If we look more at the detail required, we find the situation is even worse than expected. For each reflected visibility polygon, there is a sequence of mirrors that leads to the final reflected visibility region. If a reflected visibility region is formed by *l* reflections, there are *l* mirrors associated with that region. Therefore a well-organized data structure is needed. In *VisPoly*, a depth-first search tree (see, e.g. [Algo]) is used for this purpose. Each node of the tree could have up to n children to keep the O(n) polygons after one more reflection. As illustrated in the following figure, the tree is constructed according to the level of reflections. Each node contains the visibility polygon, the mirror that generated the visibility polygon and the mirror image of light source corresponding to the mirror.

Figure 7: Organize polygons in tree structure

By using the above bookkeeping method, we can readily perform a variety of queries. We achieve this flexibility by allowing the space complexity to grow to $O(n^{k+2})$: there are altogether $O(n^{k+1})$ polygons, and for each polygon, we keep track of all its vertices.

2.8 Queries

It is expensive to calculate all of the reflected visibility polygons. Wireless communication system design provides the motivation. Given a set of points as base stations (see Section 3 for details), we need to calculate the region covered by these base stations after k reflections of the radio signals. A main question is whether the whole area is covered, that is whether there is any point that cannot receive radio transmissions from any base station (see, e.g. [WB1][WB2][Lee 93]). If a point can be reached by several base stations, another question is which base station is the closest and what is the length of the shortest path (see, e.g. [VB]). *VisPoly* solves simplified versions of these problems where only one base station is presented.

2.8.1 Point location, reflection paths, and shortest path

A first question we want to answer is whether a point is visible under the current reflection. The solution used in *VisPoly* is to search through the reflection polygon tree, and perform a point location query on each visibility polygon. If the point is not inside of any of the polygons, the point is in a **blind spot** (see, e.g. [ADD95] for definition of blind spot).

If a point is inside a certain reflected polygon, *VisPoly* further calculates the **reflection paths**, i.e. the paths that a ray can follow from the original source, being reflected at mirrors, and finally reaching the point. Meanwhile the total distance of the reflection path is computed. The information of the reflection paths and their distance is displayed on the screen. In order to give user a clear view, the shortest path will be highlighted with a catchy red path (see User Manual of *VisPoly* for details).

As we search all through the reflection polygon tree, the query for all of above information can be answered in $O(n^{k+2})$. The cost of searching through the polygon tree is $O(n^{k+1})$. Furthermore, we spend O(n) for the query of whether a point is inside a polygon (see, e.g. [OR96] for point simple location in a simple polygon). With additional space and preprocessing, and with substantially more implementation time, we could have achieved an O(log n) point location query time (see, e.g. [DS]).

2.8.2 Visibility region

In order to provide a general view of the region covered with a certain number of reflections. *VisPoly* displays the visible region within that required reflection level on the screen. That is, *VisPoly* displays the simple polygon with direct visibility, the polygons within one reflection (including the direct visibility polygon, and the visibility polygons with one reflection), the polygons within two reflections (including the direct visibility polygon, the visibility polygons with one reflection, and the visibility polygons at the second reflection), etc. The required reflection level cannot exceed a user-given maximum number of reflections, as only visibility polygons up to that maximum number of reflections are calculated. *VisPoly* shows the solution by displaying those polygons on the screen with different colors representing the different levels of reflection (see User Manual of *VisPoly* in Appendix A for details).

3 Application to wireless network design

Visibility graphs with reflections can be used in the study of wireless communication. Wireless communications systems are formed by a collection of transmitters/receivers, which are called base stations. For each base stations, an area, called its *cell* is associated with it. A base station enables communication for the mobiles within its cell. Given a “universe” in which mobiles need to be able to roam freely while maintaining

communication, the locations of the base stations in wireless communication design need to be assigned to achieve this goal. For each base station one must calculate the shape and dimensions of the cell. Cells of adjacent base stations will overlap. A mobile located in the intersection of two or more cells needs a protocol for when it switches its communications from one transmitter to another (see, e.g., [WB 1], [WB 2], [Lee 93]). All of these problems rely on geometry (see, e.g. [VB]).

First let us look at what is done in wireless system design in operation today. Currently in the first generation and second generation of wireless system design, an area is usually covered by a tessellation of identical cells. Generally the base station is sited at the center of the cell. A mobile communicates with the closest base station. When the mobile is leaving a cell, the signal received by this base station becomes weaker and weaker, while a nearby base station gets progressively stronger. The system can then detect the direction of the mobile. When the mobile leaves a cell, it will enter the cell covered by the nearby base station, and hence the communication is shifted to that base station. As the wavelength used in communication is assumed to be long enough to bypass obstacles, no obstacles are modeled to be in the coverage area. Therefore reflections is not considered (see, e.g. [WB1], [WB2], [Lee 93]).

As short length radio waves cannot travel far and can be easily reflected by obstacles, it was not considered as a good choice in the first generation and second generation of wireless system design. On the other hand, as the energy for short length wave diminishes much faster than long wavelength wave, less interference occurs between neighbor base stations, improving the quality of the signal. Also it does not require a large antenna to send/ receive signals. Small equipment hanging on a light pole will do the job. In the urban areas we cannot ignore the obstacles, such as buildings. In order to provide high quality wireless communication system, the millimeter wave frequency is a better choice. The millimeter wave propagates mostly in line-of-sight, and deflects when hits an obstacle. Therefore, the shapes of cells are determined by the specific geometry of streets, squares, intersections, etc. As the shape and dimension of the cell becomes irregular, directly calculations of the geometry become the essential component of the wireless communication design of the urban area (see, e.g. [VB], [Lee 3]).

VisPoly can calculate the visibility polygons with reflections given a simple polygon and one source, which is a base station in a wireless system. In wireless communication system, a group of cooperative transmitters/receivers is needed to cover an area (see, e.g. [WB1][WB2]). In order to find the optimal placement for the transmitters/receivers, we should first know the covering area of each transmitter with the prescribed number of reflections. We also need to know the interference of these transmitters under reflections (see, e.g.[VB]). *VisPoly* is able to compute the visibility region of a polygon without holes. In the case of urban area, buildings become obstacles and we need to model the area to be covered as polygon with holes (see, e.g. [VB]). Furthermore, the obstacles such as buildings are often lined up, with numerous collinear corners of buildings. Hence more questions arise in analysing this design problem in detail. In Section 5.1 an enhancement algorithm (see, e.g. [SO86]) to calculate the visibility region in a polygon

with holes will be mentioned, and more discussion of the new problems and suggested solutions will follow (see, e.g. [Perkins]).

4 Implementation issues

4.1 Implementation difficulties

Computational geometry is a science of exactness. On the contrary the computation facility available to us is the digital computer with limited bits of precision (see, e.g. [ARCH]). The problem of **round-off error** (see [NUM] for definitions) has been well observed in many fields of computing. The cumulative effect of round-off error is blamed for the malfunctioning of computer systems in many applications (see, e.g. [NUM]). The problem of non-exactness is especially serious in geometry problems (LEDA). Let us look at a simple example. Suppose that we want to find out the position of a point using a computer program, and assume that the actual point is to the right of a given line. After certain amount of calculation, due to the round-off error, the x-coordinate may be a little bit smaller than the exact one. The computer may report the position of the point as being not to the right of the line but to the left of the line.

Is there a cure to get rid of round off error? Two possible solutions are presented. First of all we can use a multi-precision library (see, e.g.[MPFUN]) in completing the calculation. A multi-precision library is useful in many applications (see, e.g. [MultiApp]), as we can bypass the round-off error, which occurs in every computer. In fact infinite precision is the basic assumption for many proofs (see,e.g. [LP90]). The infinite precision achieved in digital computers with fixed precision is in fact a symbolic computation. Multi-bytes of integer are used to represent a real value. When system finds the new value is out of the range of current representation, it can expand the representation ability by using more space. These calculations cannot benefit from the speed of floating point processors of new computers. Exactness is achieved by paying a penalty in both speed and space. Software using the multi-precision library has poor performance, and the performance penalty would be magnified in this project, since our project already requires $O(n^{k+2})$ in time complexity (see, e.g. [MPFUN]).

LEDA provides another possible strategy. LEDA provides two types of object library in parallel. One is the ordinary library using floating point; the other use the data type called “rational number”, unique to LEDA. Each geometry object has two implementations, and either may be selected. The authors of LEDA chose to allow a pair of numbers, the numerator and the denominator, represents a rational value since the floating point never completely solve the round-off error problem. The rational number option provides a good approximation to exactness, and has been used in many LEDA demonstration programs. It performs quite well in solving moderately complex problems, but still does not completely fixed the round-off error problem. Rational values can achieve exactness only in the range of values it represented. It may produce another problem related to the digital computer, i.e. the overflow/underflow problem. Suppose we have two lines, which

are indicated by a pair of rational numbers, the slope and the intercept. The intersection of the two lines will probably have a larger denominator than do the rational numbers representing the two lines. After a serial of computions, we may wind up with a very large denominator, one larger than the maximum integer in the computer, which produces "overflow". Similarly "underflow" cannot be entirely prevented by rational number representation (see, e.g. [LEDA]).

Due to these considerations, none of the above strategies is employed in *VisPoly*. Instead the numerical difficulties are solved case by case. The following two examples will discuss some of the technical methods used in *VisPoly*.

Consider the following example. To compute the portion of polygon in the cone of two reflected light rays of a mirror, we calculate the intersection of the original polygon and the cone. We get a list of polygons, and we need to determine which is the one that has the mirror as an edge. The two end points of mirror are two vertices of the polygon. When *VisPoly* first was implemented, the author tried to find a polygon having these two vertices. But due to round-off errors, the computer reported that none of the polygons satisfied this criterion. As the list of polygons is calculated by polygon intersection, the coordinates of the vertices of the new polygon are not the same as the coordinates of the vertices of the original polygon where the two polygons share vertices. Hence the author changed the criterion to finding the edge which is most likely to be the mirror. First the midpoint of the mirror is calculated, and then all edges of the polygon lists are examined. The edge of closest distance to that midpoint is reported.

While round-off error causes problems in many applications, the test for collinearity (see, e.g. [OR96]) is a particularly sensitive problem in computational geometry. Numerous algorithms branch based on the result of a collinearity test. For example, the edges of a visibility polygon can be grouped into two categories in *VisPoly*: each edge directly visible from the source which will reflect light is called a **mirror**; and the extension of the line segment from a source point and through a reflex vertex cannot reflect light, hence is called a **window**. In the next level of reflection, each mirror produces another reflected visibility polygon; windows do not. Mirrors are kept while windows are discarded. A window is characterized by having both end points collinear with the light source. A test for collinearity that specifies that the area formed by the three points, the two end points and the source, is zero (see, e.g.[OR96]) is never satisfied: due to round-off errors, the area formed by three collinear points is usually a very small number instead of exactly zero when calculated by a computer. So we must test instead whether the three points are almost collinear, i.e. the three areas formed by the three point is less than a predefined small number *epsilon*, where *epsilon* is set to 10^{-7} in the implementation, which is close to the smallest value that can be stored exactly in computer with single precision (see, e.g. [NUM]). An edge satisfying this weaker condition is considered a window. Two types of edge may satisfy the condition given above. One is considered true collinearity, the other is the case of **thin mirror**. A thin mirror is a mirror with very short length causing the area formed by the mirror and source to be tiny. It is almost zero considering the round-off error of the computer.

We must further test the distance between the light source and the edge to differentiate the two conditions. A thin mirror can cause a program to crash due to cumulative effect. The thin mirror usually reaches the limit of representation ability of a computer: one thin mirror can possible generate n thin mirrors after one reflection, proliferating the problem. The choice made in *VisPoly* is to filter out all the thin mirrors, as in practice the light reflected by thin mirror represents a negligible amount of energy in the case of wireless design. We choose to consider it as noise (see [Lee 93] for definition of noise).

Because of a series of practical adjustments made in *VisPoly*, *VisPoly* is quite reliable software now. The author tested dozens of polygon of different shapes, and the reflection level is set to up to ten reflections. *VisPoly* calculated visibility regions successfully in every instance.

4.2 Possible discrepancy after intensive calculation

As the round-off problems are not completely solved by *VisPoly*, we cannot guarantee that *VisPoly* will handle all situations without a problem. A log-file is therefore kept to keep the trace of calculation. The status of important steps is all kept to indicate possible problems in calculation.

4.3 Possible improvements:

Our algorithm has time complexity of $O(n^{k+1} \log n)$, while the algorithm presented in [ADD+95] is $O(n^{2k} \log n)$. The reason for this discrepancy is as follows. In [ADD+95], Aronov *et al.* calculated the union of all visibility polygons with at most k reflections. They first calculated each visibility polygon, and then merged these polygons. While merging the polygons, lots of intersection points appear. These intersection points increase the combinatorial complexity, and therefore the complexity of the algorithm in [ADD+95] is higher than ours. In *VisPoly* we just calculated the visibility polygons and did not merge them.

Although the complexity of our algorithm is lower than that in [ADD+95], it could still be prohibitive in a large system. Once we analyze the special requirements of each application, we may come up with a better solution. First let us look at some of the characteristics of wireless communication. The energy of a radio wave drops by a factor proportional to the square of the distance from the source, and at each reflection a larger portion of energy will be absorbed (see, e.g. [Lee 93]). Hence, the wave strength after a certain number of reflections is negligible. Also when the length of the total path (direct path and reflected paths) reaches a certain a threshold, it can also be discarded from our consideration. This information may help us to prune the polygon tree (see [Algo] for definition of tree pruning), and reduce computing.

The advantage of computing the union of all visibility polygons as in [ADD+95] is that the query time (time spent in point location queries) is much lower. For example, if we

want to know only whether a point is visible, we can first find out the contour of the union of reflected visibility polygons first and then apply standard point location methods (see, e.g. [DS] or [PS85]). We can solve a point location query in O(*2k log n*) time, which is basically O(*log n*).

In wireless system design we need more information than just whether a point is illuminated (see, e.g. [Lee 93]). We need at least to keep the overlay (see [BKOS97] for definition of overlay) of these polygons, so that we may report the mirror, the reflection paths, etc. The overlay will be more complex than the simple union of $O(n^{2k})$ polygons, as we need to keep the internal intersection points as well as the outer boundary and the holes. Therefore the geometric complexity of visibility region with k reflections will be at least $\Omega(n^{2k})$. The overlay approach may speed up the query but will not be able to reduce the space complexity, nor the time complexity of calculating the visibility region.

5 Enhancement of algorithms to solve more difficult problems

5.1 Visibility region in a polygon with holes

An algorithm to calculate the visibility region in a polygon with holes can be found in [SO86]. The idea is as follows: holes and the outer boundary of the polygon are considered as lists of segments, and then one calculates the visibility region amidst line segments using an angular sweep (see [BO79] for sweep-line algorithm). Assume that an initial ray starts from the light source in the negative horizontal direction. We sweep the ray clockwise and report the first segment it touches. After we finish a complete rotational sweep, the lists of segments and the line segments connecting consecutive end points of two segment form the boundary of the visibility region with obstacles. Figure 8 is an illustrative example.

Figure 7: Using sweep-line to calculate the visibility region amidst line segments.

The time complexity for this algorithm is O(*n log n*), which Suri and O'Rourke [SO86] proved is worst-case optimal, i.e., there are instances which need Ω (*n log n*) time in order to calculate the visibility region.

This sweep-line algorithm can also be used to calculate visibility region in a simple polygon without holes. The time complexity of the sweep-line algorithm is O(*n log n*) in the case of simple polygon without holes, therefore it is not the optimal algorithm. If we use the sweep-line algorithm to calculate the k reflected visibility region, we will find the *log n* penalty is not crucial. As we have shown in previous section, we have $O(n^{k+1})$ polygons after k reflections. If we use the optimal algorithm with the time complexity of O(*n*) for each polygon, the time complexity to calculate all the visibility polygons within k reflections is the $O(n^{k+2})$. If we use the sweep-line algorithm instead, the time complexity to calculate all the visibility polygons within k reflections then $O(n^{k+2} \log n)$. We have only a *log n* penalty for using sweep-line algorithm. As the sweep-line algorithm can easily be parallelized (see Section 5.2 for details), the sweep-line algorithm is not a bad choice even in the case of calculating visibility polygons without holes within k reflections.

5.2 Parallel algorithm for visibility polygons

Another possible improvement may be achieved by using parallel algorithms. Parallel algorithms have been designed to calculate visibility region within simple polygon without holes (see, e.g. [GSG92]). The parallel algorithm proposed in [GSG92] reduces the time complexity from linear time to O(log n). The total time for the $O(n^{k+1})$ polygons is therefore $O(n^{k+1} \log n)$. Compared with algorithm used in VisPoly, the parallel algorithm improves the time complexity from $(n^{k+2} \log n)$ in VisPoly to on-line time complexity $(n^{k+1} \log n)$. This parallel algorithm, on the other hand, requires the manipulation of a complex data structure ([GSG92]).

The sweep-line algorithm discussed above can also be parallelized [SO86]. Let us revisit the algorithm [EA81] first. As we traverse clockwise through the vertices of the polygon, more and more information is generated. We cannot get complete information of a vertex before visiting every other vertex. Each potential vertex of the visibility polygon is kept on a stack. We may backtrack or scan forward to decide whether an edge is a mirror or window. No mirror is output before we have visited every vertex. This algorithm depends on global information and thus cannot easily be parallelized. On the other hand the sweep-line algorithm decides each mirror locally. Once a portion of line segment is the closest line segment to the source, it can be identified as the mirror, and we can begin immediately to calculate the visibility polygon under the reflection of this mirror. This approach reduces the total on-line time according to the number of parallel machines available. If we have massively parallel machine, i.e. a machine has lots of identical CPU, the time reduction is dramatic.

5.3 New topics identified:

Addition problems arise naturally from other applications. I will discuss some of the problems and suggest of the solutions in the following sections.

5.3.1 Art gallery problem under reflection.

The optimal placement of the transmitters/receivers is central to wireless communication system design. Given an arbitrary polygon with holes, what is minimal number of transmitters needed to cover the area subject to reflections [Perkins]?

To decide this question, we need a clear definition of what constitutes coverage. What metric measures the ability to cover? A spot is covered only if it can receive a certain minimal amount of energy. To simplify the problem, we can initially consider a spot covered if it can be lighted within k reflections. A simplified question then becomes what is the minimal number of transmitters/receivers needed to illuminates a polygonal area. In case of k=0, the problem become the art gallery problem [Perkins].

A current result for the art gallery problem shows that $\lfloor n/3 \rfloor$ sources are sufficient to cover a simple polygon with n vertices, and for some polygons the number is necessary, i.e. $\lfloor n/3 \rfloor$ sources are required to cover the n-gons. But the $\lfloor n/3 \rfloor$ bound usually is much higher than that actual number of guards needed (see, e.g. [OR95]).

A related problem is to calculate a set of points that can cover the whole area (see, e.g. [OR95]). When the area is a simple polygon, the time complexity to calculate such a set is O(n) (see [LP79] for details). This set of points will guarantee the coverage of the area, but may not be optimal.

The problem of computing the optimal coverage of a simple polygon is NP-complete (see [OR 87], chapter 9). For k>0, the complexity is unlikely to improve.

5.3.2 Partition of the original polygon into like areas.

In the implementation of *VisPoly*, a search of the entire polygon tree is required for each query, requiring $O(n^{k+2})$ time. This calculation is very costly, far from the goal of performing point location in O(*log n*). Therefore, if it were possible to divide the visibility area into disjoint areas with all points in one area sharing the certain key properties, we might be able to speed up the queries [Perkins].

For example, if we could divide the polygon into disjoint polygons where all points in each subpolygon can be reached from the source by following the same sequence of mirrors, and if the paths using this mirror sequence are the shortest, then we would be able to calculate the shortest path quicker. If there are m subdivisions inside the polygon, we would need O(*log m*) to find out the location of the point. As we would know the sequence of mirrors ahead of time, we would need constant time for each mirror in

calculating the shortest path. Then the total cost would be O(*log m + A*), where *A* is the number of mirrors in the path.

Subdividing the polygon according the above criterion would be difficult. Currently we have algorithms that can divide the polygon into regions only according to whether the region is visible from the source by allowing up to k reflections, or not (see, e.g. [BKOS97]). To work out this division problem, we first calculate all reflected visibility polygons, and then merge them together. The resulting region is the portion visible from the source within k reflections in any sequence of mirrors, and the rest are blind spots. We do not know how to do the more refined calculation.

6 Conclusion

How best to calculate visibility polygons with reflection is an interesting research topic. It raises more new questions than the author can solve. The current version of *VisPoly* can calculate the visibility polygon with fixed k reflections. Although the performance of *VisPoly* seems good under current testing, it does not use the optimal method and our test cases so far have had thirty vertices at most. The time complexity for calculating the visibility region within k reflection in *VisPoly* is $O(n^{k+2} \log n)$, and the space used is $O(n^{k+2})$. Many open problems remain in the applicationality of visibility polygons to wireless communication design [Perkins].

7 Reference

[ADD95] B.Aronov, A.R.Davis, T.K.Dey, S.P.Pal, and D.C.Prasad, "Visibility with reflection." *Proc. 11th ACM Symp.Comput. Geom.*, 1995, pp. 316-325.

[ADD+95]B.Aronov, A.R.Davis, T.Kdey, S.P.Pal and D.C.Prasad, "Visibility with multiple reflection." *Discrete Comput. Geom., to appear.*

[EA81] H.ElGindy and D.Avis, "A linear algorithm for computing the visibility polygon from a point." *J. Algorithms*, 2:1981, pp. 186-197.

[OR97] J.O'Rourke., "Visibility." *CRC Handbook of Discrete & Computational Geomtry.* J.E.Goodman and J.O'Rourke, eds. CRC press, 1997.

[DS] Dobkin and Souvaine, "Computational Geometry: A User's Guide." Chapter 2 in *Advances in Robotices*, J.Schwartz and C.K. Yap, eds. Lawrence Erlbaum, 1987, pp. 43-93.

[GSG92] M.T.Goodrich, S.Shauck, and S.Guha, "Parallel methods for visibility and shortest path problems in simple polygons." *Algorithmica*, 6:1992, pp. 461-486.

[LEDA] *LEDA User Manual Version 3.6*, Max-Planck Institut, Saarbruecken.

[SO86] Subhash Suri, Joseph O'Rourke, "Worst-Case Optimal Algorithms for Constructing Visibility Polygons with Holes." *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, 1986, pp. 14-23.

[OR96] Joseph O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1993.

[OR87] Joseph O'Rourke, *Art Gallery Theorems and Algorithms.* Oxford University Press, New York, 1987.

[Perkins] Greg Perkins, "Private communication." March, 1998.

[Lee93] W.C.Y.Lee, *Mobile Communication Design Fundamentals.* John Wiley&Sons, New York, N.Y., 1993.

[BKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 1997, pp. 19-41.

[WB1] "How wireless works." Cellular Telecommunications Industry Association, http://www.wow-com.com/consumer.

[WB2] "Wireless Technology Explained." Cellular Telecommunications Industry Association, http://www.wow-com.com/consumer.

[VB] Fernando J.Velez, Jose M. Brazio, "A computational-Geometry-Based Tool for the Cellular Design of Millimeterwave Mobile Communication Systems in Urban Environments." Universidade da Beira Interior, http://demnet2.ubi.pt/~fjv/.

[Nutshell] Daniel Gilly, *Unix in a Nutshell: A Desktop Quick Reference for System V & Solaris 2.0,* O'Reilly & Associates, 1992.

[Physics] Anthony.J.Buffa, Jerrry D.Wilson, *College Physics, 3/e*, Prentice Hall Engineering, Science & Math, 1997.

[C++] Bjarne Stroustrup, *The C++ Programming Language, Third Edition.* Addison-Wesley, 1997.

[Algo] Tomas Cormen, Charles Leiserson, and Ronald Rivest, *Introduction to Algorithms.* MIT press, 1990.

[ARCH] John, L.Hennessy and Daniel A. Patterson, *Computer Architecture: A Quantitative Approach, 2nd Endition*. Morgon Kaufmann, 1995.

[NUM] Kendall E. Atkinson, *Introduction to Numerical Analysis, 2nd Edition*. John Wiley, 1989.

[MPFUN] "MPFUN: Fortran software that permits one to perform arithmetic computation to an arbitarily highlevel of numeric precision." http://ftp.llp.fu-berlin.de/lsoft/B/O/MPFUN.html.

[MultiApp] Sarah McMillian, "Multi-precision Combustion Calculation." U.Waterloo, CA. http://sail.uwaterloo.ca/~salmcmill/workshop.html.

[LP90] M.S.Bazaraa, J.J.Jarvis, and H. Sherali, *Linear Programming and Network Flows, 2nd Edition* . John Wiley, 1990.

[PS85] F.Preparata and M.Shamos, *Computational Geometry.* Springer-Verlag, 1985.

[RS] A.F.Beardon, *A Primer on Riemann Surface.* LMS. Lecture Notes.

8 Appendix A: User manual

VisPoly is user friendly geometric software for calculating visibility polygons with a fixed number of reflections. It uses LEDA as the object library. LEDA must be installed first and path for the LEDA dynamic library must first set before using *VisPoly*.

Key in command *VisPoly*, a display window will pop up. The window first asks you the level of reflections you want. The default is zero, i.e., only directly visible polygon is interested. There are five control buttons on the panel. They are "input", "preprocess", "region", "point location", and "quit".

The first step is to input the source point, and the original polygon. To input the source point, user just need to click the left button of the mouse. To input the original polygon, user can use mouse input too. Each time a left button is clicked, a vertex is sending to the system and the right button end the polygon. The input polygon must be simple, and the light source should be in the polygon. Otherwise *VisPoly* will reject the input and ask the user to provide input data again.

Once input data is given, click "preprocess" button to calculate all the visibility polygons. This step is most costly step of the whole process. In the currently system, the speed seems to be satisfactory. The author tested calculating up to one thousand polygons and each with 30 edges, this preprocess is done within 2 minutes. To give user a general view of the polygons after reflection, each reflected light source and the polygons are displayed once it is calculated. For moderate complex polygons after three to four reflections, the original polygon is almost covered by the edges of the reflected polygons. That is a vivid demonstration of the situation after several reflections. Visibility region after several reflections is usually very complex. So if we want to use the properties in wireless cell design, we may need to simplify a system in order to make it practical. We may limit the reflection by applying certain restriction.

After preprocessing, all the information calculated are stored at a polygon tree. We then can use “region” to display the visibility region within certain level of reflection. And use “point location” to find out the visibility property of certain points. The retrieved information of point location is displayed both on the screen and the standard output file. User can direct the standard out to a file of this information for further analysis.

As the polygon tree takes lots of memory, the “input” is in effect only for one correct input data. That is once input data is given, it cannot be changed. Also there is no need to preprocess more than once, the button “preprocess” can be just pressed once. *VisPoly* blocks all illegal operations. Hence if “preprocess” are pressed twice, at the second time the button is pressed, a warning message will pop on the screen, and no operation are performed.

9 Appendix B: An illustrative example of Procedure Poly-Cut

10 Appendix C: Source code for *VisPoly*