Implementation of a $O(n\alpha(n)log(n))$ Point Visibility Algorithm on Digital Terrain Models

Pascal Junod, pascal.junod@switzerland.org

October 1999

Contents

List of Figures

to Mimi

1 Acknowledgments

First of all, I would thank Stefan Eidenbenz for introducing me into the world of digital terrains and for having proposed me to do a semester thesis in this field.

I would thank Christoph Stamm, which was my supervisor, for supporting me, for answering all my dummy questions, for motivating me to write so much lines of code and naturally for trying to understand my bad German.

2 Introduction

Describing a terrain through visibility information, such as, for instance, the points of the terrain which are visible from a selected viewpoint, has a variety of applications. For example, the computation of the number of observation points needed to view an entire region, or the computation of optimal locations for radio transmitters are possible and practical applications.

More recently, the need for interactive tools for managing mobile telecommunications network systems gives more interest to this field, because the propagation of the waves with a frequency of about 1 GHz can be well approximated by using line-of-sight models. Using such models combined with wave propagation models, such as the Okumura/Hata propagation for ultra high frequencies one, allows very good approximations.

Different approaches to compute point visibility in digital terrains have been presented in the scientific literature. In this semester thesis, we have developed and implemented a fast algorithm by using different ideas which were proposed in the last decade, and by modifying an existing algorithm.

The implementation of the algorithm has been designed to be integrated in the RA_3DIO project (see [7] for further details on this project); RA_3DIO is a virtual reality tool developed at the ETHZ that supports the design, the optimization and the management of mobile telecommunications networks.

This semester thesis is organized as follows: first, we present the theoretical background needed to understand the algorithm, and the existing solutions. Then, in a next section, we describe our algorithm, together with the modifications needed for its integration in the RA_3DIO project. In the last section, we present the practical results, together with propositions of possible improvements.

3 Theoretical Background

In this section, we introduce some basic notions about Digital Terrain Models, about the definition of the point visibility problem and about the existing solutions.

3.1 Digital models of a terrain

A natural terrain can be described as a continuous function $z = f(x, y)$, defined over a simply connected subset D of the horizontal $x - y$-plane. Thus, a *Mathematical terrain Model (MTM)*, that we can simply call a *terrain*, can be defined as a pair $M := (D, f)$.

The notion of digital terrain model characterizes a subclass of the mathematical terrain models which can be represented in a compact way, using a *finite* number of data. We can now define the concept of *Digital Terrain Model (DTM)* as follows:

Definition 1 (Digital Terrain Model (DTM))
Let be Σ a plane subdivision of the domain D into a collection of plane regions $R = \{R_1, R_2, ..., R_m\}$. Let be F a family of continuous functions $z = f_i(x, y)$ for $i = 1, 2, ..., m$ each defined on R_i such that on the common boundary of two adjacent regions R_i and R_j, f_i and f_j assumes the same value. Then a DTM can be expressed as a pair $D := (\Sigma, F)$.

DTMs can be classified into *Regular Square Grids (RSGs)* and *Polyhedral Terrain Models*. In a RSG, the domain subdivision is a rectangular grid, while each function f_i is a piecewise linear, quadratic or cubic function obtained by linear interpolation along the edges of the subdivision. PTMs are characterized by a domain subdivision consisting of a straight-line plane graph and by linear interpolating functions.

RSGs are usually built from a finite set of data points $P_i = (x_i, y_i, z_i)$ defining the terrain, are often used to describe gridded data are internally represented by arrays. The model of terrain used in the RA_3DIO project can be viewed as a polyhedral one.

3.2 Visibility problems on terrains

We present here the common terminology used in the visibility problems on terrains.

Given a mathematical terrain model $M = (D, f)$, a *candidate point* is any point $P = (x, y, z)$ belonging to or above the terrain, or more formally, such that $(x, y) \in D$ and $z \geq f(x, y)$. We say that two candi-

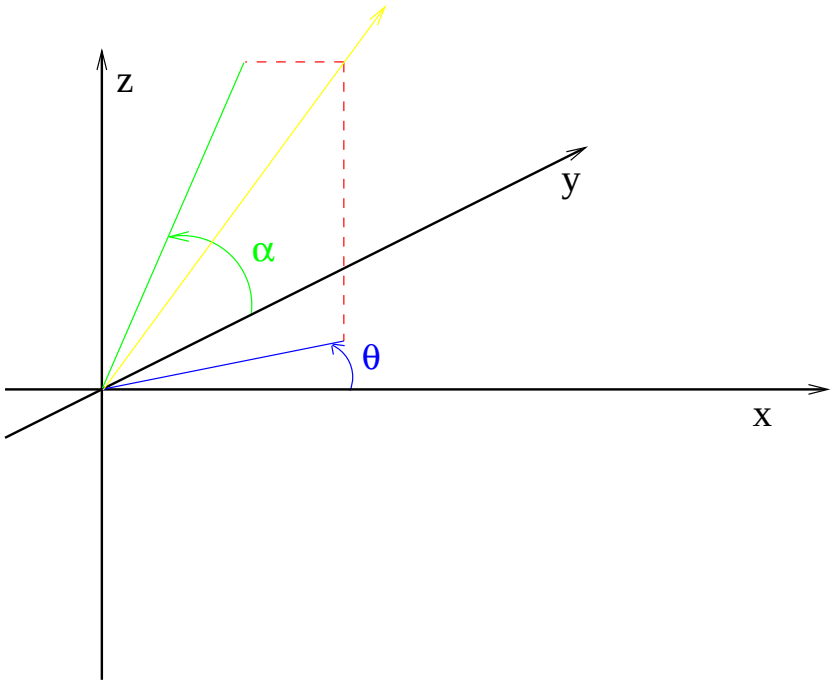

Figure 1: The spherical coordinates system

date points P_1 and P_2 are *mutually visible* if and only if, for every point $Q = (x, y, z) = tP_1 + (1 - t)P_2$ with $0 < t < 1$ and $z > f(x, y)$. In other words, two points are mutually visible when the straight-line segment joining these two points lies above the terrain, and it touches it at most at its two extremes.

An *observation point* (or *viewpoint*) is any arbitrarily chosen candidate point, and a *visual ray* is any ray emanating from a viewpoint. Given a viewpoint V and a spherical coordinates system centered at V, a visual ray r is identified by the pair (θ, α), called a view direction (see Figure 1).

It is possible to classify visibility problems (see [4]) on a terrain on the basis of the dimensionality of their output information into *point*, *line* and *region visibility* problems. Point visibility problems compute the set of points, chosen in a candidate set, visible from a predefined observation point; line visibility problems compute curves on the terrain with special visibility characteristics with respect to an observation point, such as the computation of an horizon line, for example. And finally, region visibility problems consist of the determination of portions of the terrain visible from an observation point.

We use in this semester thesis a algorithm designed for solving a line visibility problem to compute an horizon line, and then for determining the visibility of the set of candidate points with the help of its horizon line.

3.3 Concept of horizon

A line visibility problem of practical relevance in geographic applications consists of computing the *horizon* of an observation point on a terrain.

Definition 2 (Horizon)
Given a terrain $M = (D, f)$ and a viewpoint V, the horizon *of the terrain with respect to V is a function $\alpha = h(\theta)$, defined for $\theta \in [0, 2\pi]$, such that, for every radial direction θ, $h(\theta)$ is the maximum value α such that each ray emanating from V with a direction (θ, β), with $\beta > \alpha$, does not intersect the terrain.*

Or in other words, this definition corresponds to the intuitive notion that the horizon of the terrain provides, for each radial direction, the minimum elevation which must have a visual ray emanating from the viewpoint in the given direction to pass above the surface of the terrain.

In a polyhedral terrain, the horizon is a radially sorted list of labeled intervals $[\theta_1, \theta_2]$. If an interval $[\theta_1, \theta_2]$ has label l, then the visual ray defined by a direction $(\theta, f(\theta))$ with $\theta_1 < \theta < \theta_2$ hits the terrain at a point belonging to edge l. See Figure 2 for an example of horizon.

3.4 Existing algorithms for point visibility

Point visibility computations can be reduced to determine the mutual visibility of two candidates points. We present in this part first a "brute-force" approach, and then a faster approach.

3.4.1 A brute-force approach

To compute the discrete visibility region of a point of view V with respect to the set S of points, we need to determine the mutual visibility of V and the points belonging to S. To solve this problem, we can apply an algorithm which tests the mutual visibility of each pair (V, P), where$P$ belongs to S, with a computational cost of $O(n^2)$, where n denotes the cardinality of the set of points.

Given a DTM $D = (\Sigma, F)$ as defined in the previous section, and two candidate points P_1 and P_2 on D, the mutual visibility of P_1 and P_2 through a "brute-force" approach reduces to compute the intersection of the projection

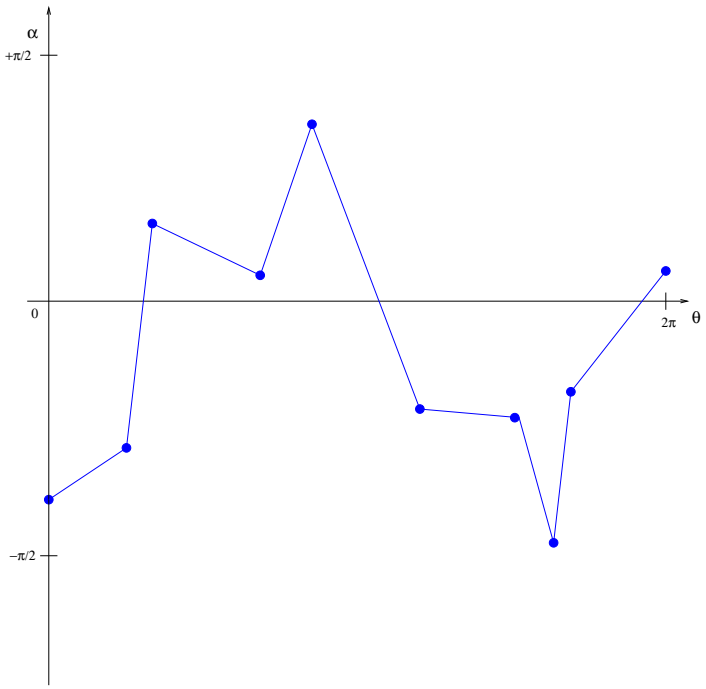

Figure 2: Example of an horizon

on the $x-y$-plane of segment $s \equiv \overline{P_1P_2}$, denoted $\overline{s}$, with the edges of Σ. At each intersection point P between $\overline{s}$ and an edge e of Σ, we test whether s lies above the edge of D corresponding to e. If s is above the corresponding terrain edge at any such intersection point, then P_1 and P_2 are mutually visible.

In general, this process has a linear time complexity, in the worst case, in the number of edges of D, which is $O(n)$, where n is the number of vertices of D.

For a regular square grid, the time complexity reduces to $O(\sqrt{n})$, by using properties of such a kind of terrain. These two kinds of algorithms, which are quite simpler to implement, are used to compare their performances with these of the non-trivial algorithm which we will present.

3.4.2 An $O(n\alpha(n)\log n)$ algorithm

The second approach preprocesses the terrain model with respect to the viewpoint V, and builds a data structure on which the problem of computing the visibility of a point P can be solved in logarithmic time. We explain here how this approach is functioning, and then we will see that we can simplify this algorithm for our purposes.

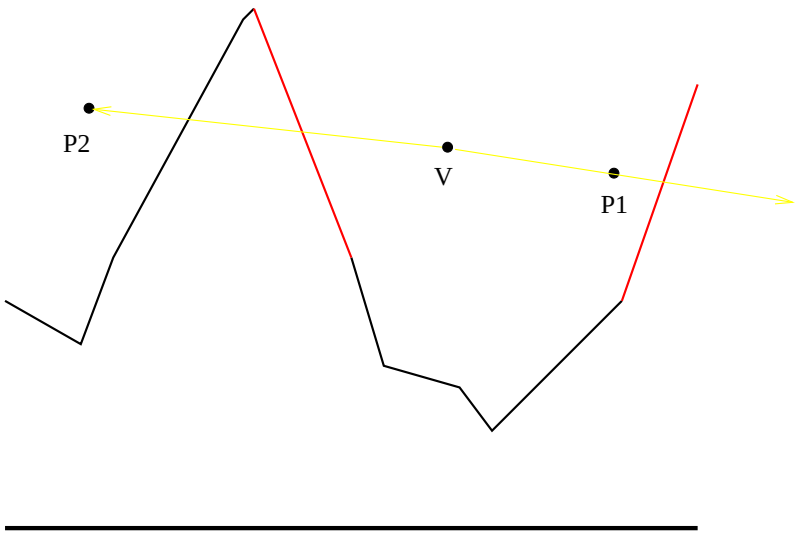

Figure 3: The ray-shooting problem

Despite the "brute force" method described above, this algorithm requires a *polyhedral terrain model*. The data structure has been proposed by Cole and Sharir in [2]. It has be designed to solve the problem of *ray-shooting* on a polyhedral terrain model (see Figure 3):

Definition 3 (Ray-shooting problem)
Given a polyhedral terrain D, a viewpoint P and a view direction (θ, α), the ray-shooting problem *consists of determining the first face of D hit by a ray emanating from V with direction (θ, α).*

The mutual visibility of two points V and P reduces to solve a ray-shooting problem, since we have just to consider as view direction the one defined by the segment $\overline{VP}$ and, when the first face of D hit by the corresponding visual ray has been obtained, we have just to determine whether P and V lie on the same or on the opposite side of the plane of such a face (as represented in Figure 3).

The data structure of Cole and Sharir, called an *horizon tree*, has size $O(n\alpha(n)\log n)$, where $\alpha(n)$ is the inverse Ackermann's function, which can be considered in practice as a constant. Ray-shooting queries can be answered in time $O(\log^2 n)$ on such a structure. A necessary condition on the polyhedral terrain is that it has to be possible to sort the edges using a partial distance order. More in this subject is coming with the next section.

The horizon tree is a balanced binary tree with a depth which is logarithmic in the number of terrain edges. In this tree, every node corresponds to a

subset of edges and stores a partial horizon; the root corresponds to the whole set of edges of the terrain. Each left child corresponds to the half of the edges associated to its parent, that are closest to the viewpoint V, and each right child corresponds to the remaining half. Every node of the tree stores the partial horizon computed on the edges associated with the left child of the node.

This horizon tree can be computed in optimal $O(n\alpha(n)\log n)$ time, since each partial horizon can be computed by a single application of the algorithm of Atallah (see [1]) for determining the horizon of a viewpoint on a polyhedral terrain. A complete description of the algorithm of Atallah is done in the next section. For further information about the Cole-Sharir data structure and its use, [4] is a good introduction.

A serious analysis of this method has shown that we can already extract the needed information (visibility of the candidate points) in the building phase of this Cole-Sharir data structure. In other words, we don't need to solve a ray-shooting problem for computing the visibility of the candidate point. So we don't need for our purposes the query phase. This fact is easily understandable, because we don't need the whole information, like the face hit by the visual ray, for example. Furthermore, such a structure is appropriate to compute the visibility of points which are not parts of the terrain. This is not a need in our case.

In the next section, we show how to modify this method to extract the needed information. Furthermore, a complete description of the Atallah algorithm is done. Last, we present our simplified algorithm.

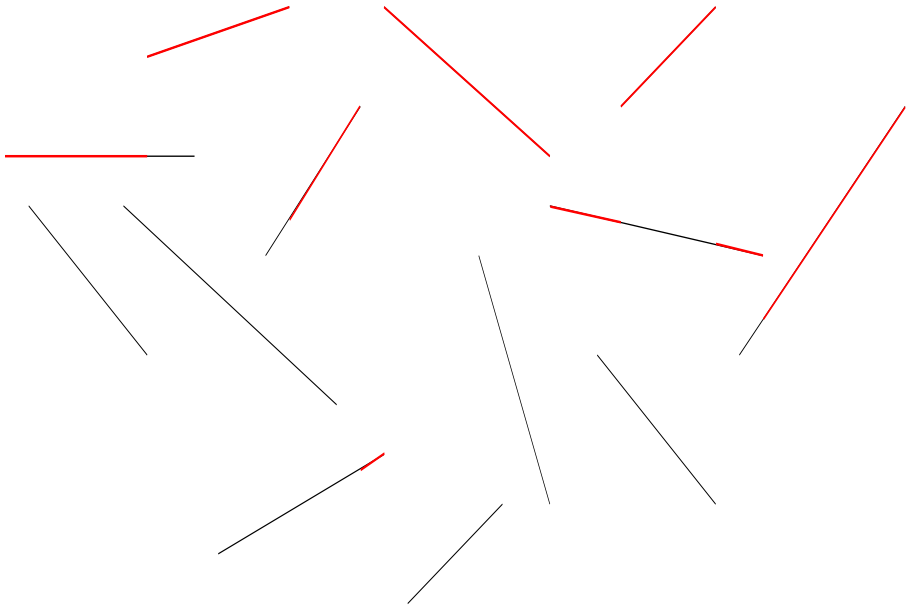

Figure 4: The upper envelope of a set of segments in the plane

4 Description of the Algorithm

In this section, we present our algorithm for computing point visibility on a polyhedral terrain.

4.1 Introduction

As described in the previous section, we use a line visibility algorithm to compute the horizon with respect to the viewpoint V. During the build phase of the horizon, it is possible to extract the needed information, i.e. to know if a point is visible or not.

The computation of the horizon of a viewpoint on a polyhedral terrain reduces to the computation of the *upper envelope* of a set of segments in the plane.

Definition 4 (Upper envelope of a set of segments in the plane) *Given p segments in the plane, i.e. p linear functions $y = f_i(x)$ with $i = 1, ..., p$, each defined on an interval $[a_i, b_i]$, the* upper envelope *of such segments is a function $y = F(x)$, defined over the union of the intervals $[a_i, b_i]$, and such that $F(x) = \max_{i|x \in [a_i,b_i]} f_i(x)$*

Or, in other words, the upper envelope maps any x value in the segment having maximum y value over x, if such a segment exists. Figure 4 shows

an example of such an upper envelope of a set of segments.

To reduce the horizon on a polyhedral terrain to the upper envelope of a set of segments, we can express the edges of the terrain in the spherical coordinates system centered at the viewpoint defined in the previous section. Note that we consider only the two angular coordinates θ and α. So we lose information, since two points on the same ray emanating from the viewpoint have the same coordinates. But if the edges are sorted using a distance order, as we will see it later, this information get not lost.

It has been shown in [3] that the complexity of the upper envelope of p segments in the plane is $O(p\alpha(p))$, and thus, the complexity of the horizon of a polyhedral terrain with n vertices is $O(n\alpha(n))$. There is different possibilities to compute the envelope of p segments in the plane; either with the help of a static divide-and-conquer approach leading to a $O(p\alpha(p)\log p)$ worst-time case complexity, like the Atallah algorithm ([1]), or with the help of a dynamic, incremental one, with a complexity of $O(p^2\alpha(p))$. In [4], such an incremental method is presented, as well as a randomized version, which has a time complexity of $O(p\alpha(p)\log p)$, too.

4.2 Topological sorting of the edges

As written before, we need to sort the edges using the following partial distance order

Definition 5 (Distance partial order)
Given a plane subdivision Σ and a point O in the plane, an edge e_1 of Σ is said to be before *an edge e_2 (and e_2* behind *e_1) with respect to O if and only if there exists a ray r emanating from O and intersecting both e_1 and e_2, such that the intersection of r and e_1 lies nearer to O than the intersection of r and e_2.*

Figure 5 illustrates this definition. We note that this order relation is only defined if there is an intersection between the ray and both edges. By using the dual graph of the polyhedral terrain, it is possible to sort the set of edges. This set will be the input of our algorithm.

4.3 A divide-and-conquer approach

The principle of the algorithm is the following: it recursively splits the set of segments into two halves, and pairwise merges the results. Merging two envelopes is performed through a sweep-line technique for intersecting two chains of segments.

If the process is done for the farthest edges first and then is coming to

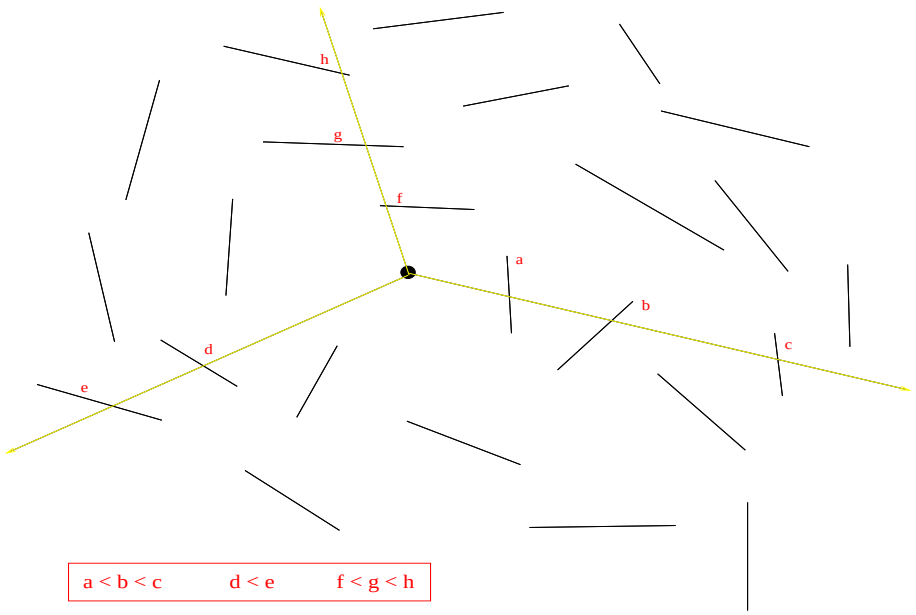

Figure 5: Definition of the distance partial order

the nearest, it is possible to know if a point is invisible. As a matter of fact, while merging two horizons, one nearer and one farther, if a vertex of the farther horizon is *under* the nearer horizon, then this vertex is invisible, and we can note this fact. We can remark that we can only know if a point is visible while the last merge operation, and this if this point is not marked as an invisible one. So, we mark all the points as visible at the begin of the horizon computation and following the merging process, we mark the invisible points.

4.3.1 The divide part

This part is the easily one. At the begin, as written before, the algorithm gets an array of sorted edges, which it will consider as an array of one-piece horizons. Then, it will merge these horizons two by two to get finally the searched horizon. This is illustrated in Figure 6.

4.3.2 The conquer part

We present here the "academic" version of this algorithm. The version which we have implemented, as well as the changes we have done will be presented in the next section. To merge two horizons, as in Figure 7, the Atallah algorithm uses a sweep-line technique implemented with the help of an event list. The sweep-line algorithm moves an imaginary vertical line r from the left

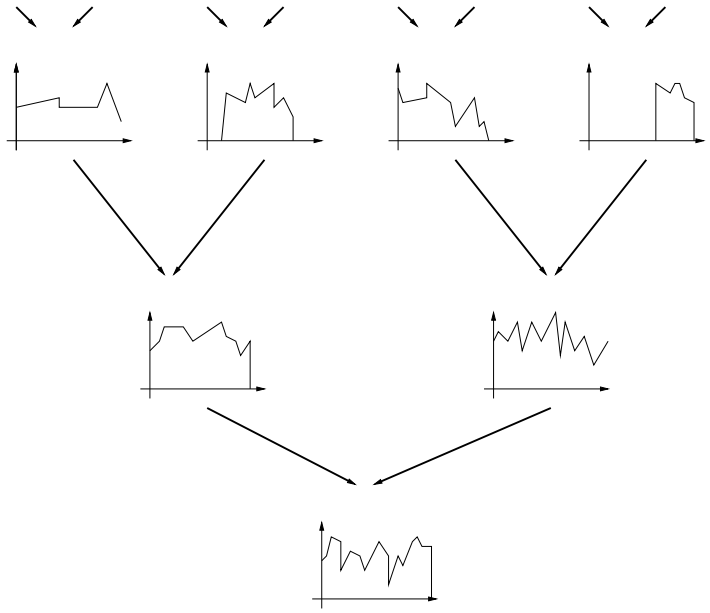

Figure 6: Illustration of the divide phase

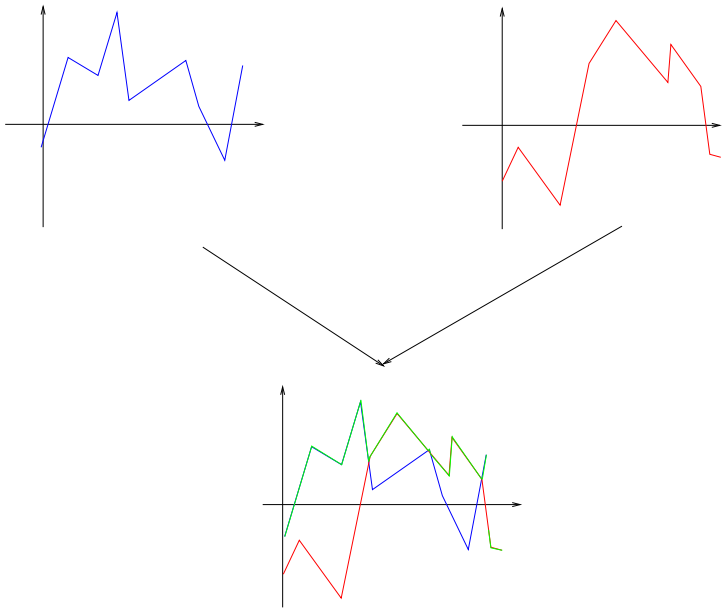

Figure 7: Illustration of the conquer phase

to the right in such a manner that at each step the resulting upper envelope restricted to the left half-plane of r has already been computed, while in the right half-plane, it is still to be determined. The events are represented by the vertices of the two input envelopes, *plus* the possible intersection points between them. The current status of the sweep-line is represented by the pair of segments, one each partial envelope, that are intersected by the sweep-line, ordered according to their heights.

There is five possible kinds of events. We list them here, as well as the necessary update operations needed for the resulting envelope. We use the following colored notation for the illustrations: a blue segment is part of the farther horizon, while a red one is part of the nearest horizon. The green segment is the updated part of the resulting envelope. The sweep line is the brown vertical line, while the last position of this sweep line is the brown dashed line.

- If the event is an intersection point of two segments (Figure 8), then the current segment in the sweep line status are swapped. The resulting envelope is updated. In this case, there is no importance in the distance of the horizons.

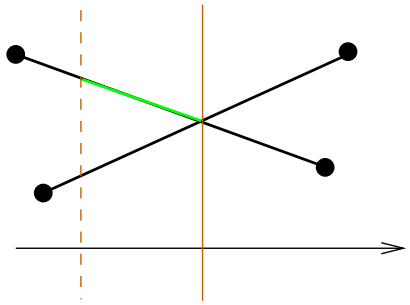

Figure 8: Case of an intersection

- If the event is a right endpoint of a segment and if this segment is the actual upper segment (Figure 9), then the interval between the last event and the current one is inserted in the resulting envelope.
- If the event is a right endpoint of a segment and if this segment is the actual lower segment (Figure 10), then this segment leaves the status; furthermore, if this segment belongs to the farther horizon as in Figure 10, then its right endpoint can be marked as invisible.
- If the event is a left endpoint of a new segment on one of the partial envelopes, then this segment is inserted in the current status (Figure

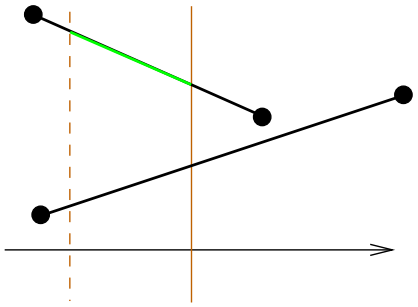

Figure 9: Case of the right endpoint of the upper segment

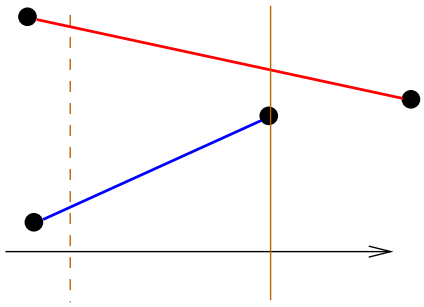

Figure 10: Case of the right endpoint of the lower segment

11), in this case as the lower edge. Furthermore, if the new segment belongs to the farther partial horizon, as in Figure 11, its left endpoint can be marked as invisible. The intersection between the new pair of status segments is also tested and eventually inserted in the event queue.

- If the event is a left endpoint of a new segment on one of the partial envelopes, then this segment is inserted in the current status, in this case as the upper edge (Figure 12). The intersection between the new pair of status segments is tested and eventually inserted in the event queue.

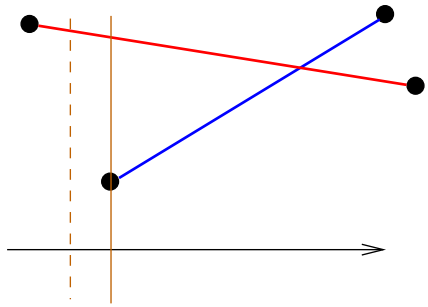

Figure 11: Case of the left endpoint of a new lower segment

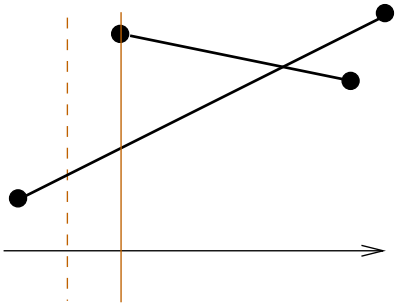

Figure 12: Case of the left endpoint of a new upper segment

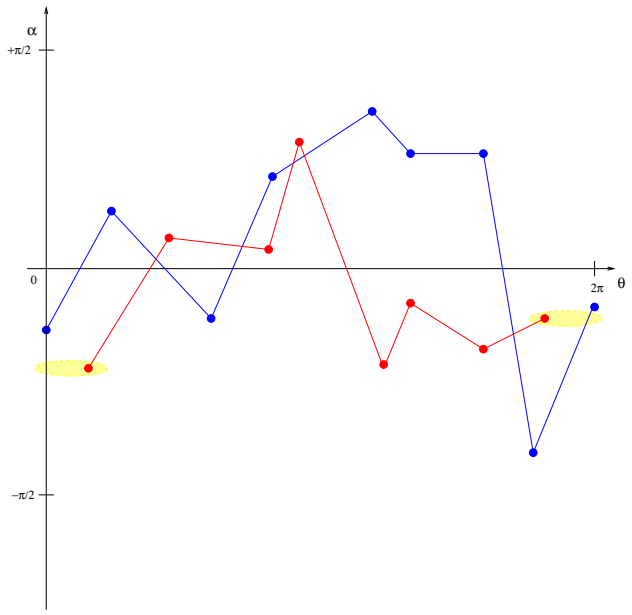

Figure 13: The two partial horizons are not defined over the same interval

5 Implementation and Integration in the RA_3DIO project

In this section, we present first two problems which were not documented in the "academic" version, together with the solutions we have chosen to solve these problems. Then, we give our algorithm in a pseudo-code notation. The integration of this algorithm in a real project has brought some other difficulties, which we present next. Finally, we make a brief presentation of the implementation, together with some comments when needed.

5.1 Problems and solutions

The algorithm of Atallah, as presented in [4], is designed for "friendly" partial horizons, which begin and end both at the same points and continuous. This is naturally not the case in a real-world application. A first problem was to find an acceptable solution in the cases where a partial horizon is not defined, as illustrated in Figure 13. The solution we chose is the following: everywhere where the horizon is not defined, we insert an horizontal *hole-edge*, with an α set to an impossible value, -2 in our implementation. So the horizons are all defined over the interval $[0, 2\pi]$, and holes are treated as they were horizons parts. This solution is illustrated in Figure 14.

Another problem which happens in a real-world application is the possi-

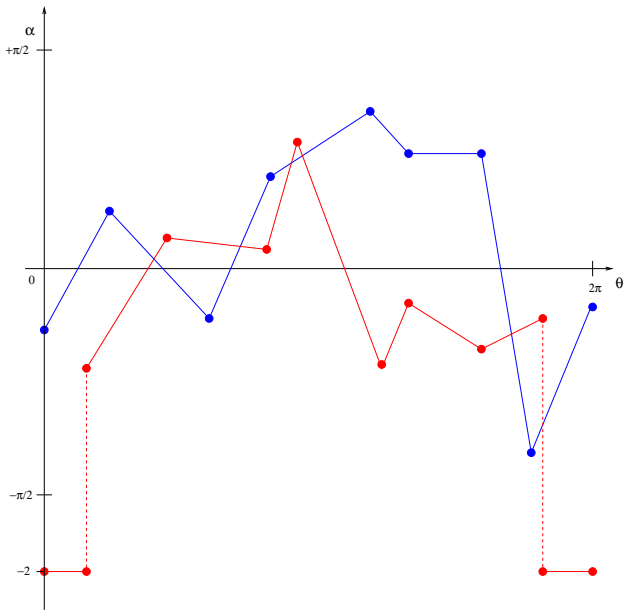

Figure 14: The "hole-edges" solution

bility that the partial horizons are not continuous, in the sense that the right endpoint of a segment and the left endpoint of the next segment have not the same α-value, as illustrated in Figure 15. Or, in other words, we have to keep our algorithm robust even if there is jumps in the partial horizons. The problem is the situation where an edge of the other partial horizon passes through this kind of jump. In this case, we have sometimes to insert a new part in the resulting envelope and to modify the internal status of the sweep line.

5.2 The final version of our algorithm

We give here the pseudo-code (in a `C++`-like notation) version of our algorithm.

5.2.1 Implementation of the divide phase

For efficiency purposes, we have chosen to implement the recursion in an iterative way. Since we cannot expect that the number of edges is a power of two, we merge first a part of the array of edges, which are topologically sorted, to reduce the size to the greatest power of two which is smaller than this number of edges. Then, we can use a simple loop to merge the resulting horizons in a tree-like structure. This is illustrated in Algorithm 1. We recall that we have to do the merge operations for the farther horizons first

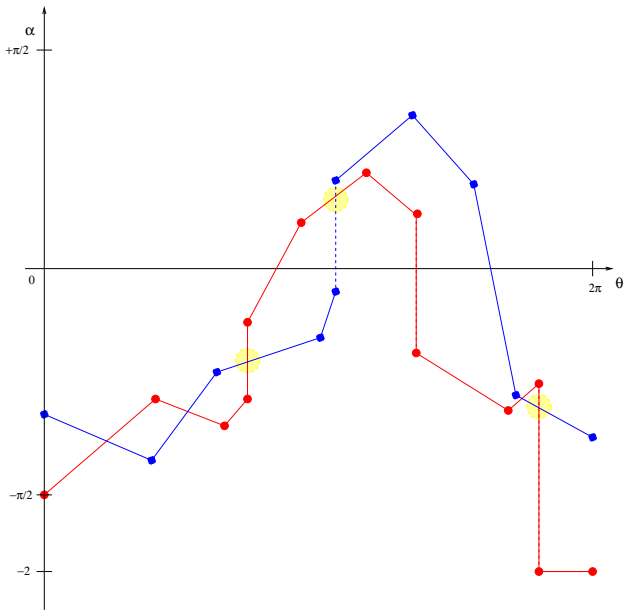

Figure 15: The jump-problem

and at last for the nearest ones, so we don't forgive invisible vertices. The farthest edge is stored in the first case of the array, while the nearest one is stored in the last case.

5.2.2 Implementation of the conquer phase

The conquer phase is by far the most complex operation in the algorithm. This procedure first builds the event list, and then processes the events, updating the resulting envelope when needed. It is naturally in this procedure that the invisibility of some vertices is determined. The pseudo-code version of this procedure is in Algorithm 2 and the end is in Algorithm 3.

Algorithm 1 Implementation of the divide phase

```
// int n ..... number of edges
// int m ..... greater power of two which is smaller as n
// int i, j ... temporary variables

// Reduction of the array to an array with a length
// which is a power of 2. To do this, we merge some
// horizons.

j = 2*m-n;
for(i = 2*m-n; i < n-1; i+=2) {
   array[j] = array[i+1].Merge(array[i]);
   j++;
}

// Merging of the whole array.

i = 0;
while (m > 2) {
   j=0;
   while (i < m) {
      array[j] = array[i+1].Merge(array[i]);
      i += 2;
      j++;
   }
   m /= 2;
   i = 0;
}

// Last merge operation

resulting_horizon = array[1].Merge(array[0]);
```

Algorithm 2 Implementation of the conquer phase (1)

```
// Building of the event list for the farther
// partial horizon.

forall(segment [a, b] in farHorizon) {
   insert a in eventList marked as LEFT_ENDOINT;
   insert b in eventList marked as RIGHT_ENDPOINT;
}
forall(segment [a, b] in nearHorizon) {
   insert a in eventList marked as LEFT_ENDOINT;
   insert b in eventList marked as RIGHT_ENDPOINT;
}
status.inf = UNDEFINED;
status.sup = UNDEFINED;
lastEvent = 0.0;

// Main loop

while(eventList.notEmpty()) {
   currentEvent = eventList.nextEvent();
   switch(e.eventType()) {

      case INTERSECTION:
         insert [lastEvent, currentEvent] in
            the resulting envelope with label
            status.sup;
         lastEvent = currentEvent;
         Swap(status.inf, status.sup);
         break;
      case RIGHT_ENDPOINT:
         if(currentEvent right endpoint
            of status.sup) {
            insert [lastEvent, currentEvent] in
               the resulting envelope with label
               status.sup;
            lastEvent = currentEvent;
            status.sup = status.inf;
            status.inf = UNDEFINED;
          } else {
```

Algorithm 3 Implementation of the conquer phase (2)

```
            if (status.inf belongs to farHorizon) {
               mark right endpoint of status.inf
                  as invisible;
            }
            status.inf = UNDEFINED;
          } // end of else
          break;
      case LEFT_ENDPOINT:
         snew = segment whose left endpoint is
                currentEvent;
         if (snew over status.sup) {
            status.inf = status.sup;
            status.sup = snew;
            if (jump_situation) {
               insert missing part in the resulting
                  envelope;
         } else {
             if (status.inf belongs to farHorizon) {
                mark left endpoint of status.inf
                   as invisible;
             }
             status.inf = snew;
         }
         if (status.sup and status.inf intersect) {
            (x, y) = intersection point;
            insert x in eventList, marked as
              INTERSECTION;
         }
         break;
   } // end of switch statement
} // end of while
```

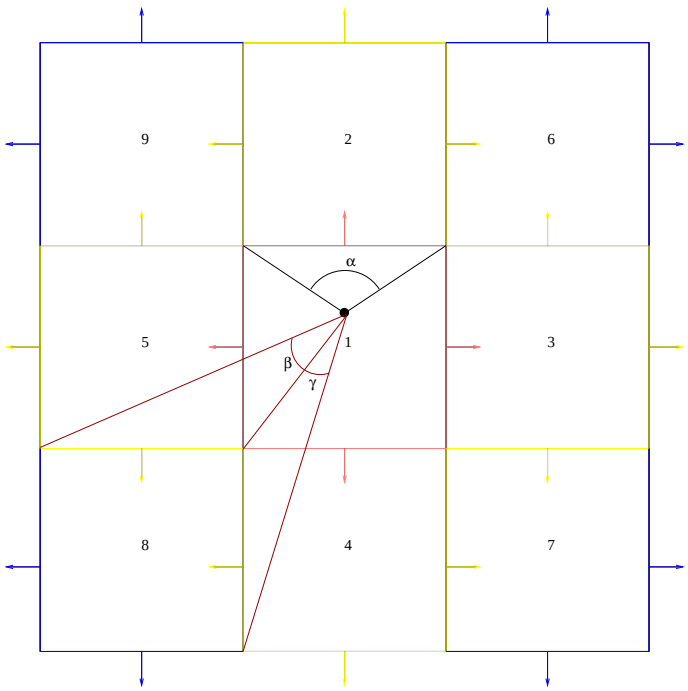

Figure 16: The subdivision of the terrain in patches

5.3 The integration in a patch environment

In the RA_3DIO project, the terrain is subdivided in small pieces, which are easier to load and to display than the whole terrain. Such a small piece of terrain is called a *patch*. Each patch is a square and has about 3000 edges in the actual configuration. We can use the presented algorithm to compute the horizon line and for determining the (in-)visibility of the points, but if we have to compute the visibility of points which are not in the same patch as the viewpoint, the task becomes more difficult.

The situation is illustrated in Figure 16. For example, to compute the horizon line of the patch number 2, we have first to compute the internal horizon line of the patch, and then to merge this horizon line with the portion of the horizon of the patch number 1 delimited by the α value, or, in other words, with the north side of the central patch.

In a more complicated manner, the computation of the visibility of the points in the patch number 8 needs the following operations: first we have to compute the internal horizon line and then to merge this horizon with the β portion of patch number 5 and with the γ portion of patch number 4.

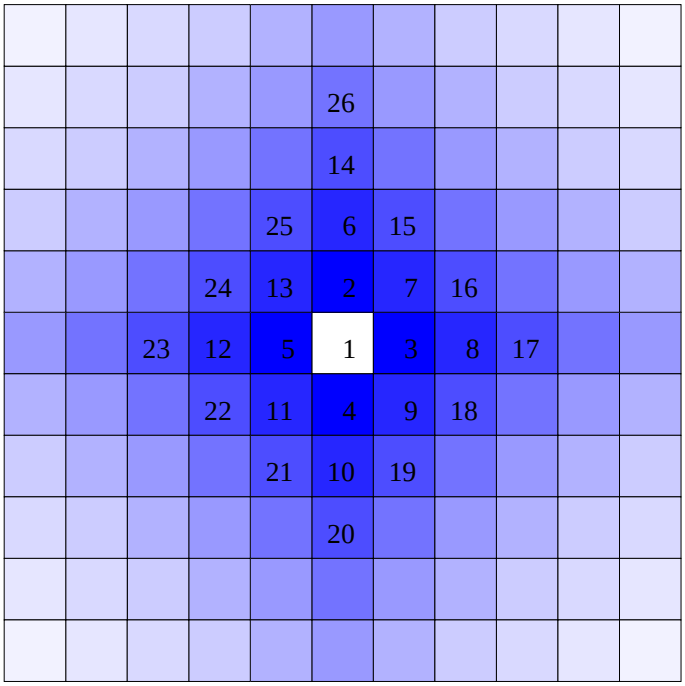

Figure 17: The processing order in a patches environment

Thus, for computing the visibility of the points in a 3x3- or a 5x5-grid of patches, they must be preprocessed in a way, such that the horizons are available when needed. This gives us the processing order illustrated in Figure 17. To merge the southern part of the horizon line of patch number 5 with the eastern one of patch number 4, we have just to append these two horizons. But for computing the horizon line of patch number 6, the operation is more complicated, because the patch is "cut" by the x-axis. So, in this kind of situation, we have to treat these horizons in a special way. These two situations are summarized in Figure 18.

5.4 Description of the classes and methods

In this part, we present briefly the three new classes which were inserted in the RA_3DIO project. The algorithm has been implemented in `C++`. The LEDA library (see [5] for a description) has been employed for the common data structures. During the development time, [6] has been used as a reference book.

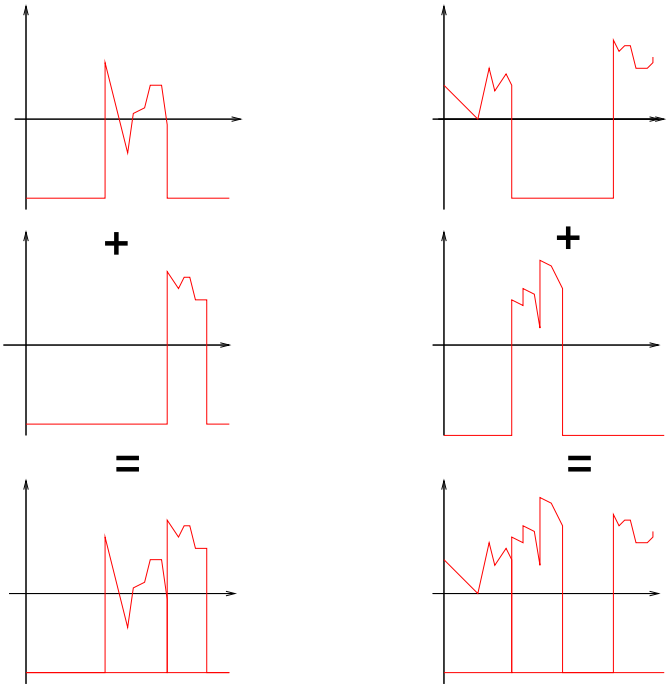

Figure 18: Handling of normal horizons and "cut" horizons

5.4.1 The `CEventItem` class

The `CEventItem` class defines the properties of the events which are used during the merge procedure. The main goal of this class is to store information needed by the algorithm for its (imaginary) sweep line. The methods are all procedures which permit to read and to set the class variables.

5.4.2 The `CHorizonItem` class

An horizon is in fact a list of `CHorizonItem`s, which can be viewed as the segments in an horizon. Again, the methods are all procedures which permit to set and to read the class variables.

5.4.3 The `CHorizon` class

This class is by far the most complicated and the most interesting one. It models the concept of an horizon line. The `CHorizon` class is a subtype of a LEDA double-linked list. It has the following methods:

- `CHorizon(CHorizonItem& horizonItem)` is a constructor used to build an horizon object with a single `horizonItem` object. It is used to

transform a set of edges in a list of one-piece horizons, inserting the necessary holes.

- `CHorizon(leda_list<CHorizonItem>& array, CTransmitter* tx, CPatch *patch, CHorizon& horizon = CHorizon())` is a constructor used to build a new horizon. It computes the internal horizon line, and if needed, merges this new horizon line with the one stored in the input variable `horizon`. The implementation of the divide phase is done in this method.

- `CHorizon Merge(CHorizon& horizon, double leftLimit, double rightLimit, bool last)` is the conquer part of the algorithm. For efficiency reasons, `Merge()` takes an interval as input and merges the two partial horizons only between these two values, which considerably reduces the number of events in the most number of the cases.

- `bool Over(const leda_segment sinf, const leda_segment ssup, double x) const` decides whether an edge is over another, in an horizon. Note that the decision is defined when the two edges are holes, too.

5.4.4 The `PatchPreprocess()` method

`void CWaveModelPointVis::PatchPreprocess(CTransmitter* tx, CPatch* patch)` is the method called by RA_3DIO when the user puts a viewpoint on the terrain. It is called in the good order, as defined in Figure 17. The following operations are done: it creates first a directed copy of the (undirected) dual graph of the polyhedral terrain, then the topological sorting of the triangle edges is done, it computes the visibility of all the points, marks the invisible ones and merges the patches horizons, when needed. Another method, while displaying the terrain, will test for each vertex if it is visible or not.

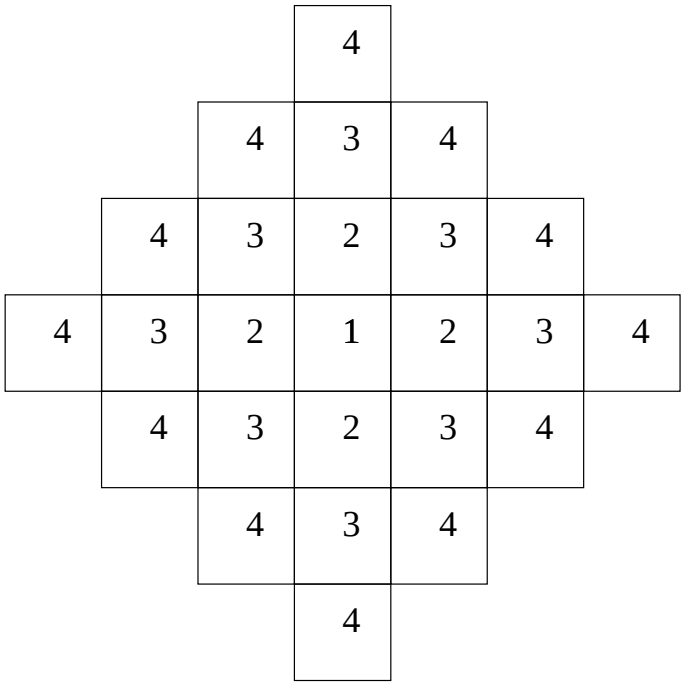

Figure 19: The "radial distance"

6 Results

In this section, we present a little time complexity analysis, together with a performance comparison between three point visibility algorithms.

6.1 Time complexity

As written before, the merge procedure has a time complexity of $O(n\alpha(n))$, where n is the number of edges in a patch and $\alpha(n)$ the inverse Ackerman-n's function. For the central patch, we call this conquer procedure $O(\log n)$ times, which gives us a time complexity of $O(n\alpha(n)\log n)$. Then, we have to do an internal merge for each external patch; m being the number of patches, it gives us a time complexity of $O(m \cdot n\alpha(n)\log n)$.

Last but not least, we have to compute the time complexity of the merge operations between patches. We englobe the preceding reflexions in our analysis. For this purpose, let define the following variables:

- i for the "radial distance" of the patches, as illustrated in Figure 19.
- v for the number of internal patches
- t for the number of border patches

- l for the maximal length of the horizon
- c for the average length of the horizon per border patch
- H for the time needed to compute the horizon

We have following values for these different variables:

i	u	v	t	l	c	H
1	1	1	1	$n\alpha(n)$	1	1
2	4	1	5	$5n\alpha(5n)$	$5/4$	$4 \cdot 1 = 4$
3	8	5	13	$13n\alpha(13n)$	$13/8$	$4 \cdot 5/4 + 8 \cdot 5/4 = 15$
4	12	13	25	$25n\alpha(25n)$	$25/12$	$4 \cdot 13/8 + 16 \cdot 13/8 = 32\frac{1}{2}$
5	16	25	41	$41n\alpha(41n)$	$41/16$	$4 \cdot 25/12 + 24 \cdot 25/12 = 58\frac{1}{3}$

We have the following equalities between these variables:

$$u_i = 4(i-1)$$

$$v_i = 1 + \sum_{j=1}^{i-1} u_j = 2i^2 - 6i + 5$$

$$t_i = u_i + v_i = 2i^2 - 2i + 1$$

$$H_i = u_i n\alpha(n)\log n + 4c_{i-1} + (u_i - 4) \cdot 2 \cdot c_{i-1} = u_i n\alpha(n)\log n + 2c_{i-1}(u_i - 2)$$

We can approximate H_i as follows:

$$H_i = O(i)n\alpha(n)\log n + 2O(i-1)n\alpha(O(i-1)^2 n)(4i-6)$$

which is finally

$$O(i)n\alpha(n)\log n + 2O(i^2)n\alpha(i^2 n)$$

To get the final time complexity estimation, we have to sum these H_i's:

$$\sum_{i=2}^{m} \left(n\alpha(n)\log n + O(i) \cdot n \cdot \alpha(i^2 n)\right) + n\alpha(n)\log n$$

which gives us:

$$O(mn\alpha(n)\log n + m^2 n\alpha(m^2 n))$$

We can remark that when the number of patches tends to the number of edges pro patches, then our algorithm is not very interesting. This is not the case in the reality, the number of patches being relatively small compared to the number of edges in a patch.

6.2 Comparison between the three algorithms

We make now a comparison between the three algorithms presented in this semester thesis. The two first are easily implementable and are available in the RA_3DIO project. We recall that the first one, which use a "brute-force" strategy, has a complexity of $O(m \cdot n^2)$. The second one, which uses specific properties of polyhedral terrains, has a time complexity of $O(m \cdot n^{3/2})$. The following table summarize time measures (in seconds) for the computation of point visibility for different numbers of patches. The number of edges per patch n is equal to about 3000.

# patches	$O(m \cdot n^2)$	$O(m \cdot n\sqrt{n})$	$O(mn\alpha(n)\log n + m^2 n\alpha(m^2 n))$
1	0.5	0.2	2.8
9	12.5	6.2	24.5
25	71.6	31.7	52.2
49	179.8	71.9	108.9
81	353.9	130.6	196.9

The first remark we can do is that the algorithm with a quadratic complexity becomes fast the slowest one, and that it is not practical. Secondly, it is clear that the second algorithm is faster than the third. We can understand this fact as follows: the implementation of the second one is very simple and thus the constant term is quite small, while our algorithm is quite complicated. Therefore, the little number of edges per patch handicaps the relative performance of the theoretical faster one.

Now, it is possible to improve its performances by many ways. First of all, the divide-and-conquer strategy could allow a parallelization of the computation of the horizon line. Then, it is always possible to improve the performances by optimizing the code. But it is quite time consuming !

7 Conclusion

It is now time to conclude this semester thesis. We have developed and implemented a (theoretical) fast algorithm for computing point visibility on a polyhedral terrain. This algorithm has been integrated in the RA_3DIO project, which means that a lot of not documented problems have requested robust solutions.

However, our algorithm is not faster than a more trivial one, which is better for the parameters used in a typical RA_3DIO utilization. This fact illustrates well the differences between a theoretical proposition and a practical application of such a "fast" algorithm.

In a more personal way, I have enjoyed this work a lot, because it has brought to me experience in programming in `C++`, as well in managing a relative big implementation. Furthermore, it is quite interesting to work in the border of the theory and the practice. Finally, I will keep excellent memories of the house, of the little desk under the roof and of the life in a little team.

References

[1] M. Atallah. Dynamic computational geometry. In *Proc. 24th Symposium on Foundations of Computer Science*, pages 92–99. IEEE Computer Society, Baltimora, 1983.

[2] Richard Cole and Micha Sharir. Visibility problems for polyhedral terrains. *Journal of Symbolic Computation*, 7(1):11–30, January 1989.

[3] H. Edelsbrunner. The upper envelope of piecewise linear functions: tight bounds on the number of faces. *Discrete and Computational Geometry*, 4:337–343, 1989.

[4] L. De Floriani and P. Magillo. Visibility algorithms on triangulated terrain models. *International Journal of Geographic Information Systems*, 8(1):13–41, 1994.

[5] Leda homepage : `http://www.mpi-sb.mpg.de/LEDA`.

[6] Stanley B. Lippman and Josée Lajoie. *C++ Primer, 3rd Edition*. Addison-Wesley, 1998.

[7] The project homepage : `http://www.ra3dio.ethz.ch`.