Efficient viewshed computation on terrains in external memory

Marcus V. A. Andrade, Salles V. G. Magalhães, W. Randolph Franklin, Barb Cutler, Mirella A. Magalhães,...

Abstract

Nowadays, a huge volume of data about terrains is available and generally, the data can not be completely stored in computer’s internal memory. So, many GIS applications require efficient algorithms to manipulate the data externally. One of these applications is the viewshed computation that consists in obtain the visible points from a given point \( p \). In this paper, we present an efficient algorithm to compute the viewshed on terrains stored in external memory. In most practical cases (when the observer’s radius of interest is smaller than the terrain size), the algorithm complexity is \( O(\text{scan}(n^2)) \) where \( n^2 \) is the number of points in a \( n \times n \) DEM and \( \text{scan}(n^2) \) is the minimum number of I/O operations required to read \( n^2 \) contiguous items stored in external memory. Also, as shown in the results, our algorithm outperforms the known algorithms described in the literature.

1 Introduction

Terrain modeling is an important area in GIS applications where an interesting problem is the computation of all points that can be viewed from a given point (the observer); the region composed by the visible points is named \textit{viewshed} [15, 18]. This problem has been widely studied in many applications such as to determine the minimum number of cellular phone towers to cover a region [5, 11, 9], to optimize the number and position of guards to cover a region [19, 14], to analyze the influences on property prices in an urban environment [23], to optimize path planning on DEM [24], etc.
The recent technological advances in data collection (such as LIDAR and IFSAR) have produced a huge volume of data about Earth’s surface [27]. For example, a $100\text{km} \times 100\text{km}$ terrain sampled at $1\text{m}$ resolution results in $10^{10}$ points. But, most of computers can not store/process this huge volume of data internally and so, the algorithms need to process the data in the external memory, generally disks. Since the required time to access and transfer data from and to the external memory is much longer than time for internal processing, the algorithms must try to minimize the external memory access [2, 20].

More specifically, algorithms that process data in external memory should be designed (and analyzed) considering a computational model that evaluates the algorithm complexity based on data transfer operations instead of cpu processing operations. One of these models was proposed by Aggarwal and Vitter [1] where the algorithm complexity is measured considering the number of I/O (input/output) operations executed.

In this work, we present an efficient algorithm to compute the viewshed of a point on terrains stored in external memory. The algorithm is an adaptation of Franklin and Ray’s method [18, 17] allowing an efficient manipulation of huge terrains (6GB or more). The large number of disk accesses is optimized using the library STXXL [13]. Comparing our algorithm with the algorithm proposed by Haverkort et al. [21], we can say that our algorithm is much easier to implement and, as tests results showed, it is about 6 times faster than that one.

The paper is organized as follow: the section 2 gives a brief description about works on viewshed computation and also, on I/O-efficient algorithms for general problems and for viewshed computation too; in the section 3, the viewshed concepts are formally presented; in section 4, the I/O-efficient computational model is shortly described; in section 5, the algorithm is described in details and its complexity is presented in section 6; the tests results are given in section 7 and the conclusions in section 8.

## 2 Related Works

### 2.1 Terrain representation

In general, a terrain can be represented by a *triangulated irregular network (TIN)* or a *Raster Digital Elevation Model (DEM)* [25, 16]. A TIN is a
vector based representation of a surface made up of irregularly distributed nodes with three dimensional coordinates \((x, y, z)\) that are connected and arranged in a network of non overlapping triangles. Thus, the surface is approximated by triangle patches and the elevation (the \(z\) coordinate) of any point can be interpolated from the vertices of the planar triangle containing the \((x, y)\) coordinates of the point. A DEM is a digital file or a matrix consisting of terrain elevations for ground positions at regularly spaced horizontal intervals.

There is no consensus about which of these representations is the best and there are many discussion about this theme \([22, 16]\). Anyway, we can say that DEM requires a simple data structure, it is easier to analyze and has high accuracy at high resolution, but it requires high memory space and it is time-consuming processing. On the other hand, TIN has a restricted accuracy, requires more complex algorithms, but it is less memory-consuming and more time-efficient processing. Given its simplicity, in this work, we consider a terrain represented by a DEM.

### 2.2 Visibility on terrains

The visibility on terrains has been widely studied in many different areas. For example, Stewart \([26]\) shows how the viewshed can be efficiently computed for every point of a DEM and his interest involves radio transmission towers positioning. Kreveld \([28]\) proposes a sweep-line approach to compute viewshed in \(O(n \log n)\) time on a \(\sqrt{n} \times \sqrt{n}\) grid. In \([17, 18]\), Franklin and Ray describe experimental studies for fast implementations of visibility computation and present several programs that explore trade-offs between speed and accuracy. Kim, Rana and Wise in \([29]\) analyze two strategies to use viewshed for optimization problems. Ben-Moshe et al. \([8, 6, 7]\) have worked on visibility for terrain simplification and for facilities positioning. For a survey on visibility algorithms, see \([15]\).

### 2.3 External memory processing

Some problems related to external memory processing are discussed by Aggarwal and Vitter \([1]\). They proposed a computational model to evaluate the algorithm complexity considering the number of input/output operations executed. In \([20]\), Goodrich et al. presented some variants for the sweep plane
paradigm considering external processing and Arge et al. [4] described a solution for the external processing of line segments in the context of GIS. This technique was also used to solve problems in hydrology such as the computation of the water flow and watershed [3] on huge terrains.

Recently, Haverkort et al. [21] presented an adaption of the Kreveld’s method to compute the viewshed on terrains stored in the external memory. The (I/O) complexity of this algorithm is $O(sort(n))$, where $n$ is the number of points in the terrain. It is worth to say that the algorithm described in this paper is faster and easier to implement than that one.

3 Viewshed Problem

Most of GIS problems related to visibility involve the viewshed computation and in general, they are optimization problems such as the optimal positioning of facilities, the siting guards minimization, path planning, etc.

The visibility problems can be classified into two major categories: visibility queries and visibility structures computation. The visibility queries consist in checking if a given point is visible or not from an observer (another point) on the terrain. This query can be answered assuming that a point $q$ is visible from another point $p$ if and only if the segment connecting the two points, named the line of sight, is strictly above the terrain (except on the ending points $p$ and $q$). See figure 1.

![Figure 1: Points Visibility: $p_1$ and $p_4$ are visible from $p_0$; $p_2$ and $p_3$ are not visible from $p_0$.](image)

The visibility structures computation consists in determining some terrain features such as the horizon, the viewshed, etc. Formally, the viewshed of a
point \( p \) on a terrain \( T \) can be defined as:

\[
\text{viewshed}(p) = \{ q \in T \mid q \text{ is visible from } p \}
\]

Frequently, it is convenient to restrict the viewshed to an observer’s neighborhood, for example, to consider only the points inside a circle centered at \( p \) with radius \( r \), the \textit{radius of interest}. In this case,

\[
\text{viewshed}(p, r) = \{ q \in T \mid \text{distance}(p, q) \leq r \text{ and } q \text{ is visible from } p \}
\]

Unless explicitly said otherwise, when the radius of interest has been defined, we will use \textit{viewshed}(p) to refer to \textit{viewshed}(p, r).

Usually, the viewshed (on a DEM) is represented by a grid whose side is equal to the radius of interest and each cell stores 1 or 0 to indicate if that cell (point) is visible or not, respectively.

4 I/O efficient Algorithms

As mentioned before, when processing a huge amount of data, the data transfer between fast internal memory and slow external storage (such as disks) often becomes the computation bottleneck. However, many GIS algorithms for terrain manipulation are designed assuming internal processing and usually, the design goal is to minimize the internal computation time; hence, these algorithms often do not scale to large datasets.

Therefore, the design (and analysis) of algorithms to process huge amount of data stored in external memory need to be done under a computational model that evaluates the input and output operations and determines the algorithm complexity based on these operations. One of these models frequently used was proposed by Aggarwal and Vitter [1]. Shortly, this model defines each I/O operation as the transfer of one (disk) block of size \( B \) from the external to the internal memory or vice-versa. Then, the measure of performance is determined by the number of such I/O operations executed. The internal computation time is assumed to be free.

The complexity of an algorithm is given based on the complexity of some fundamental problems such as scan or sort of \( N \) contiguous elements stored in the external memory whose complexities related to I/O operations are:

\[
\begin{align*}
\text{scan}(N) & = \Theta\left(\frac{N}{B}\right) \\
\text{sort}(N) & = \Theta\left(\frac{N}{B} \log\left(\frac{M}{B}\right) \left(\frac{N}{B}\right)\right)
\end{align*}
\]
where $M$ is the internal memory size.

It is important to notice that, usually $\text{scan}(N) < \text{sort}(N) << N$ and so, in many practical situations, it is significantly better to have an algorithm doing $\text{sort}(N)$ instead of $N$ I/O operations. Therefore, many algorithms try to reorganize the data in the external memory to decrease the number of I/O operations executed.

5 External Memory Viewshed Computation (EMVS)

Our algorithm, named External Memory Viewshed (EMVS), is based on the method proposed by Franklin and Ray [18] that computes the viewshed of a point on a terrain represented as an internal memory matrix. A short description of this method is given below.

5.1 Franklin and Ray’s Method

Given a terrain represented by a $n \times n$ elevation matrix $T$ and given a point $p$ on $T$, the algorithm computes the viewshed of $p$ restricted to a circle of radius $r$ (the radius of interest) centered at $p$. The algorithm performs a radial sweep of this circle using a ray, a line of sight (LOS), starting at $p$ and walks along each LOS to determine if terrain positions on the LOS are visible from $p$ or not. A terrain position $q$ is visible from $p$ if the segment $pq$ does not intersect any position whose height is higher than $q$.

To simplify the circle sweeping, the algorithm uses a square bounding box of side $2r$ centered at $p$ and the lines of sight are defined connecting $p$ to each cell in the square border. Initially, all cells inside the bounding box are set as not visible and for each line of sight $l$, the algorithm starts at $p$ setting the height of $l$ as $-\infty$ (i.e., a big negative number). So, this height is updated (increased) whenever a higher cell is reached. That is, supposing the current height of $l$ is $h$ and the next cell $c$ has height $h'$, if $h < h'$ then the cell $c$ is marked as visible and the height of $l$ is updated to $h'$; on the other hand, if $h \geq h'$, the cell status and the height of $l$ are preserved. The viewshed is stored as a $2r \times 2r$ bit matrix where the visible positions are indicated by 1 and the not visible by 0 (the positions inside the square but outside the circle are set as not visible).
Of course, this algorithm can be used to compute the viewshed on terrains in external memory (just consider the terrain grid stored in a file). But, since the cells are accessed in a sequence defined by the radial sweeping, this would require a random access to the file and the execution time would be unacceptably long. This random access order can be avoided using the adaptation described below.

5.2 The EMVS algorithm

The basic idea is to generate a list with the terrain positions (points) sorted by the processing order, that is, the points will appear in the list in the sequence given by the radial sweeping and by the processing order in each line of sight. Thus, during the viewshed computation, the algorithm follows the list avoiding to access the file randomly.

It is important to say that the list is also stored in the external memory, but it is managed by a special library STXXL (Standard Template Library for Extra Large Data Sets) [13] that implements containers and algorithms to process huge volumes of data. This library allows an efficient manipulation of data stored externally and, as stated by the authors, “it can save more than half the number of I/Os performed by many applications”.

More specifically, the algorithm creates a list $L$ of pairs $(c, i)$ where $c$ is a matrix cell (a terrain position) and $i$ is an index that indicates “when” the cell $c$ should be processed. That is, if a cell $c$ has an index $k$ then $c$ will be the $k$-th cell to be processed.

To compute the indices, the lines of sight (originating at the observer $p$) are numbered in the counterclockwise order starting in the horizontal left to right line of sight which receives the number 0 - see figure 2. Thus, the cells are numbered increasingly along each line of sight; when a line of sight ends, the enumeration proceeds from the observer (again numbered) following the next line of sight. Of course, a same cell (point) can receive multiple indices since it can be intercepted by many lines of sight. It means that a same point can appear in multiple pairs in the list $L$, but each pair will have a different index. Also, if the observer is near to the terrain border, that is, if the distance between the observer and the terrain border is smaller than the radius of interest $r$, some “cells” in a line of sight can be outside the terrain. In this case, those “cells” still will be numbered but they will be ignored (i.e. they will not be inserted in the list $L$). This is done to simplify the indices computation avoiding many additional conditional tests.
Notice that if the cells indices were computed as described above, the file still would be randomly accessed as in the original algorithm. So, to build the list $L$, the algorithm reads the terrain cells sequentially from the external file and for each cell $c$, it determines (the number of) all lines of sight that intercept that cell.

Since a cell is not “undimensional”, we can determine the cells intercepted by a line of sight using a process similar to the line rasterization [10]. That is, let $s$ be the side of each (squared) cell and suppose the cell is referenced by its center. Also, let $a$ be a line of sight whose slope is $\alpha : 0 < \alpha \leq 45^\circ$. So, given a cell $c = (c_x, c_y)$, see figure 3, the line of sight $a$ “intersects” the cell $c$ if and only if the intersection point between $a$ and the vertical line $c_x$ is between the points $(c_x, c_y - 0.5s)$ and $(c_x, c_y + 0.5s)$; more precisely, given $(q_x, q_y) = a \cap c_x$, $a$ intersects $c$ if and only if $c_y - 0.5s \leq q_y < c_y + 0.5s$. In this case, the dashed line in the figure 3 will be assumed to intersect the cell above $c$.

Then, as one can see, all lines of sight intersecting the cell $c$ are those between the two lines connecting the observer to the points $(c_x, c_y - 0.5s)$ and $(c_x, c_y + 0.5s)$ - figure 3. Let $k_1$ and $k_2$ be the numbers of these two lines respectively. Considering the line of sight definition (a segment connecting

\footnote{For $45^\circ < \alpha \leq 90^\circ$, use a similar idea interchanging $x$ and $y$.}
the observer and the center of cells on the square border) and the line enumeration, the number of all lines intersecting the cell $c$ are given by the cells whose center are between $k_1$ and $k_2$ - see figure 4. For example, in this figure, the cell $c$ is intersected by the lines 3 and 4.

Now, given a cell $c$, let $r$ be the number of a line of sight intercepting $c$. Then, the index $i$ of the cell $c$ associated to $r$ is given by the formula $i = r \ast n + d$, where $n$ is the number of cells in each ray (this number is constant for all rays) and $d$ is the (horizontal or vertical) distance between the points $c$ and $p$ - see figure 5. Notice that the distance $d$ is defined as the maximum between the number of rows and columns from $p$ to $c$.

Next, the list $L$ is sorted by the elements' index and then, the cells are processed in the sequence given by the sorted list. When a cell $c$ is processed, all the “previous” cells that could block its visibility were already processed. So, the visibility of $c$ can be computed, as described before, just checking the height of the cells along the line of sight. When a cell located on the square border is processed, it means that the processing of a line of sight has finished and the next cell in the list will be the observer’s cell indicating that the processing of a new line of sight will start.

For efficiency purpose, the algorithm uses another list $L'$ (also stored externally and managed by STXXL) to keep only the visible cells. So, when
the algorithm determines that a cell c is visible, this cell is inserted in the list $L'$. In general, this list is much smaller than $L$ since usually many points are not visible and also, it does not keep the indices.

Given the list $L'$, the algorithm stores the viewshed in an external file where the visible positions are indicated by 1 and the not visible by 0s. To avoid a random access, before storing the file, the list $L'$ is sorted lexicographically by x and y.

Finally, it is worth to say that an efficiency improvement is achieved storing a piece of the terrain matrix in the internal memory. The idea is to store the cells around the observer since these cells are processed more times than the farther ones. So all cells inside a square centered at the observer position are stored in the internal memory and they are not inserted in the list $L$. In this way, when a cell needs to be processed, the algorithm checks if it is in the internal memory. If yes, the cell is processed normally; otherwise, it is read from the list $L$.

6 Algorithm complexity

Let $T$ be a terrain represented by a $n \times n$ elevation matrix. So, $T$ has $n^2$ cells (points). Also, let $p$ be the observer’s position and let $r$ be the radius of interest. As described in section 5.2, the algorithm considers the cells that are inside the $2r \times 2r$ square centered at $p$. Assuming that each cell’s side is
then there are, at most, \( \frac{2r}{s} \) cells in each square’s side which implies there are \( \frac{8r}{s} \) cells on the square’s perimeter. Let \( K = \frac{r}{s} \). Thus, the algorithm shoots \( 8K \) lines of sight and since each line of sight has \( K \) cells, the list \( L \) has, in the worst case, \( O(K^2) \) elements.

In the first step, the algorithm does \( \frac{n^2}{B} \) I/O operations to read the cells and to build the list \( L \). Next, the list with \( O(K^2) \) elements is sorted and then it is swept to compute the cell’s visibility. Thus, the total number of I/O operations is:

\[
O \left( \frac{n^2}{B} \right) + O \left( \frac{K^2}{B} \log \left( \frac{K}{B} \right) \left( \frac{K^2}{B} \right) \right) + O \left( \frac{K^2}{B} \right)
\]

Usually, the radius of interest \( r \) is (much) smaller than \( n \) (the terrain matrix side) then \( K \) is smaller than \( n \) and so, generally, the number of I/O operations is given by \( O(\frac{n^2}{B}) = O(scan(n^2)) \). But, in the worst (not usual) case, if the radius of interest is big enough to cover almost the whole terrain, the number of I/O operations is \( O \left( \frac{K^2}{B} \log \left( \frac{K}{B} \right) \left( \frac{K^2}{B} \right) \right) = O(sort(K)) \).

The algorithm also uses an additional external list \( L' \) to keep the visible cells and this list needs to be sorted. But, since the list size is (much) smaller

\[\text{Figure 5: The index cell computation.}\]
than the size of $L$, the number of I/O operations executed in this step does not change the algorithm complexity.

7 Results

The algorithm EMVS was implemented in C++, using $g++ 4.1.1$, and the tests were executed in a PC Pentium with 2.8 GHZ, 1 GB of RAM, 80 GB 7200 RPM serial ATA HD running Mandriva Linux.

The data sets used in the tests were downloaded from USGS page [?] and correspond to the regions 1, 2 and 3 showed in figure 6 sampled with a resolution of 1 arc of second (30m). In the tests we used many pieces of these terrains with different sizes; each piece was obtained selecting the observer position and the radius of interest (that defines the piece size)\footnote{We used pieces with size similar to those used by Haverkorth et al. to compare the efficiency of both algorithms.}. It is worth to say that these datasets contain a very small percentage (less than 1.5%) of no-data, that is, points for which the elevation is unknown or invalid. Usually, the no-data points are “marked” with a special value and ignored when processing the terrain. So, the amount of invalid data on a terrain can significantly influence the running time.

Table 1 shows the EMVS execution time on these terrains. In all tests we consider the worst case, that is, we used a radius of interest big enough to cover the whole terrain. Since the external processing is an important component for the execution time, this value is shown separately. And also, as the external processing time is mostly influenced by the viewshed size (i.e. the number of visible points), for each piece of terrain, the observer was positioned at different heights (1, 50, 100, 1000 and 10000 meters) above the terrain to generated different numbers of visible points. Although the values 1000 and 10000 are not very usual in practical applications, we used them to confirm the algorithm’s scalability.
Figure 6: Regions used in the tests: 1, 2 and 3.
<table>
<thead>
<tr>
<th>Terrain Pieces</th>
<th>Obs.</th>
<th>Visible Pts</th>
<th>Time (sec.)</th>
<th>Visible Pts</th>
<th>Time (sec.)</th>
<th>Visible Pts</th>
<th>Time (sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Pts</td>
<td>Size</td>
<td>Height</td>
<td>#</td>
<td>%</td>
<td>Ext.</td>
<td>Tot.</td>
<td>#</td>
</tr>
<tr>
<td>5609²</td>
<td>120 MB</td>
<td>1</td>
<td>5.2 x 10⁶</td>
<td>1.7</td>
<td>10</td>
<td>20</td>
<td>3.4 x 10⁴</td>
</tr>
<tr>
<td>100</td>
<td>6.6 x 10⁹</td>
<td>20.8</td>
<td>33</td>
<td>47</td>
<td>2.4 x 10⁴</td>
<td>76.8</td>
<td>70</td>
</tr>
<tr>
<td>1000</td>
<td>2.3 x 10⁷</td>
<td>72.2</td>
<td>69</td>
<td>90</td>
<td>2.5 x 10⁴</td>
<td>78.0</td>
<td>72</td>
</tr>
<tr>
<td>10000</td>
<td>2.4 x 10⁷</td>
<td>76.0</td>
<td>69</td>
<td>90</td>
<td>2.4 x 10⁴</td>
<td>78.0</td>
<td>72</td>
</tr>
<tr>
<td>9029²</td>
<td>311 MB</td>
<td>1</td>
<td>2.5 x 10⁶</td>
<td>3.1</td>
<td>27</td>
<td>55</td>
<td>8.6 x 10⁵</td>
</tr>
<tr>
<td>50</td>
<td>8.7 x 10⁶</td>
<td>10.7</td>
<td>53</td>
<td>87</td>
<td>2.8 x 10⁴</td>
<td>35.0</td>
<td>124</td>
</tr>
<tr>
<td>100</td>
<td>1.1 x 10⁷</td>
<td>13.5</td>
<td>61</td>
<td>95</td>
<td>6.2 x 10⁴</td>
<td>76.3</td>
<td>244</td>
</tr>
<tr>
<td>1000</td>
<td>5.7 x 10⁷</td>
<td>69.4</td>
<td>225</td>
<td>285</td>
<td>6.3 x 10⁴</td>
<td>78.0</td>
<td>243</td>
</tr>
<tr>
<td>10000</td>
<td>6.2 x 10⁷</td>
<td>77.1</td>
<td>238</td>
<td>301</td>
<td>6.3 x 10⁴</td>
<td>78.0</td>
<td>245</td>
</tr>
<tr>
<td>17150²</td>
<td>1122 MB</td>
<td>1</td>
<td>4.2 x 10⁶</td>
<td>0.2</td>
<td>66</td>
<td>199</td>
<td>1.3 x 10⁵</td>
</tr>
<tr>
<td>50</td>
<td>2.6 x 10⁸</td>
<td>1.0</td>
<td>89</td>
<td>223</td>
<td>1.5 x 10⁸</td>
<td>55.0</td>
<td>643</td>
</tr>
<tr>
<td>100</td>
<td>9.5 x 10⁸</td>
<td>3.4</td>
<td>121</td>
<td>274</td>
<td>2.1 x 10⁸</td>
<td>75.1</td>
<td>759</td>
</tr>
<tr>
<td>1000</td>
<td>1.5 x 10⁹</td>
<td>55.2</td>
<td>653</td>
<td>898</td>
<td>2.2 x 10⁸</td>
<td>78.1</td>
<td>770</td>
</tr>
<tr>
<td>10000</td>
<td>2.2 x 10⁹</td>
<td>77.4</td>
<td>779</td>
<td>1054</td>
<td>2.2 x 10⁸</td>
<td>78.1</td>
<td>801</td>
</tr>
<tr>
<td>33433²</td>
<td>4264 MB</td>
<td>1</td>
<td>1.2 x 10⁶</td>
<td>0.1</td>
<td>1651</td>
<td>2810</td>
<td>4.2 x 10⁵</td>
</tr>
<tr>
<td>50</td>
<td>7.3 x 10⁶</td>
<td>0.7</td>
<td>1821</td>
<td>3181</td>
<td>1.1 x 10⁷</td>
<td>1.0</td>
<td>1866</td>
</tr>
<tr>
<td>100</td>
<td>1.5 x 10⁷</td>
<td>1.3</td>
<td>1942</td>
<td>3278</td>
<td>1.8 x 10⁸</td>
<td>16.7</td>
<td>2449</td>
</tr>
<tr>
<td>1000</td>
<td>6.4 x 10⁸</td>
<td>57.0</td>
<td>4112</td>
<td>5966</td>
<td>8.5 x 10⁸</td>
<td>76.3</td>
<td>4605</td>
</tr>
<tr>
<td>10000</td>
<td>8.7 x 10⁸</td>
<td>77.5</td>
<td>4642</td>
<td>6568</td>
<td>8.6 x 10⁸</td>
<td>76.8</td>
<td>4546</td>
</tr>
<tr>
<td>40000²</td>
<td>6103 MB</td>
<td>1</td>
<td>3.5 x 10⁶</td>
<td>0.0</td>
<td>2827</td>
<td>4512</td>
<td>3.9 x 10⁵</td>
</tr>
<tr>
<td>50</td>
<td>2.9 x 10⁹</td>
<td>0.2</td>
<td>2924</td>
<td>4660</td>
<td>7.6 x 10⁷</td>
<td>4.7</td>
<td>3431</td>
</tr>
<tr>
<td>100</td>
<td>1.2 x 10⁹</td>
<td>0.8</td>
<td>4659</td>
<td>4812</td>
<td>1.9 x 10⁸</td>
<td>11.7</td>
<td>4133</td>
</tr>
<tr>
<td>1000</td>
<td>7.7 x 10⁸</td>
<td>48.1</td>
<td>5625</td>
<td>8108</td>
<td>9.7 x 10⁸</td>
<td>60.6</td>
<td>6006</td>
</tr>
<tr>
<td>10000</td>
<td>1.2 x 10⁹</td>
<td>77.2</td>
<td>6378</td>
<td>9079</td>
<td>1.2 x 10⁹</td>
<td>77.4</td>
<td>6678</td>
</tr>
</tbody>
</table>

Table 1: EMVS execution time (in seconds) in different pieces of terrains with the observer positioned at different heights to generate viewsheds with different number of visible points. In all cases, the radius of interest cover the whole terrain.
The EMVS performance was compared with Haverkorth et al. (IO_VS) algorithm. Figure 7 summarizes the results considering the results reported in [21] for the observer positioned about 1 meter above the terrain. The EMVS values were averaged from the three corresponding values (for regions 1, 2 and 3) given in table 1. As one can see, the EMVS is more than 6 times faster than IO_VS. And, it is important to say that, the IO_VS tests were executed using a Power Macintosh G5 dual 2.5 GHz, 1GB RAM and 80 GB 7200 RPM that is significantly faster than the machine used in our tests. Thus, we may suppose that our algorithm is still faster than that one. Finally, as an additional advantage, the EMVS algorithm is much simpler to implement than IO_VS.

8 Conclusions

We presented a very efficient algorithm to compute the viewshed of a point in huge terrains represented by a raster DEM stored in the external memory. As tests showed, our algorithm is more than 6 times faster than the other one described in the literature and also, it can process very huge terrains (we used it in 6.1GB terrain). Furthermore, the algorithm is quite simple to understand and to implement. The algorithm implementation is available at http://www.dpi.ufv.br/ marcus/TerrainModeling/EMViewshed/EMVS.tgz as an open source code distributed under Creative Common GNU GPL license [12].

Acknowledgment

This work was partially supported by CNPq - the Brazilian Council of Technological and Scientific Development, by NSF grants CCR-0306502 and DMS-0327634 and by DARPA/DSO/GeoStar.
Figure 7: Execution Time

<table>
<thead>
<tr>
<th>Terrain Size</th>
<th>EMVS</th>
<th>IO_VS</th>
</tr>
</thead>
<tbody>
<tr>
<td>119 MB</td>
<td>18</td>
<td>353</td>
</tr>
<tr>
<td>311 MB</td>
<td>53</td>
<td>865</td>
</tr>
<tr>
<td>1122 MB</td>
<td>198</td>
<td>3546</td>
</tr>
<tr>
<td>4264 MB</td>
<td>2786</td>
<td>16895</td>
</tr>
</tbody>
</table>

References


