More efficient terrain viewshed computation on massive datasets using external memory

Chaulio R. Ferreira  
Universidade Fed. de Viçosa  
Viçosa, MG, Brazil  
chaulio.ferreira@ufv.br

Salles V. G. Magalhães  
Universidade Fed. de Viçosa  
Viçosa, MG, Brazil  
salles@ufv.br

Marcus V. A. Andrade  
Universidade Fed. de Viçosa  
Viçosa, MG, Brazil  
marcus@ufv.br

W. Randolph Franklin  
Rensselaer Polytechnic Inst.  
Troy, NY, USA  
wrf@ecse.rpi.edu

André M. Pompermayer  
Universidade Fed. de Viçosa  
Viçosa, MG, Brazil  
andre.pompermayer@ufv.br

ABSTRACT

We present a better algorithm and implementation for external memory viewshed computation. It is about four times faster than the most recent and most efficient published methods. Ours is also much simpler. Since processing large datasets can take hours, this improvement is significant. To reduce the total number of I/O operations, our method is based on subdividing the terrain into blocks which are stored in a special data structure managed as a cache memory. The viewshed is that region of the terrain that is visible by a fixed observer, who may be on or above the terrain. Its applications range from visual nuisance abatement to radio transmitter siting and surveillance.

Categories and Subject Descriptors

F.2.2 [Nonnumerical Algorithms and Problems]: Geometrical problems and computations

General Terms

Algorithms, Experimentation, Performance

Keywords

Terrain modeling, GIS, External memory processing

1. INTRODUCTION

Terrain modeling has been widely used in Geographical Information Science (GIS) including applications in hydrology, visibility and routing. In visibility applications it is usual to compute which points can be viewed from a given point (the observer) and the region composed of such points, known as viewshed [6]. Some applications include minimizing the number of cellular phone towers required to cover a region [3], optimizing the number and position of guards to cover a region [7], etc.

There are various algorithms for viewshed computation but most of them were designed assuming that the terrain data fits in internal memory. However, the huge volume of high resolution terrestrial data available has become a challenge for GIS since the internal memory algorithms do not run well for such volume of data on most computers. Thus, it is important to optimize the massive data processing algorithms simultaneously for computation and data movement between the external and internal memory since processing data in external memory takes much more time. That is, the algorithms for external memory should be designed (and analyzed) considering a computational model where the algorithm complexity is evaluated based on I/O operations. A model often used was proposed by Aggarwal and Vitter [1] where an I/O operation is defined as the transfer of one disk block of size B between the external and internal memory and the performance is measured considering the number of such I/O operations. The internal computation time is assumed to be comparatively insignificant. An algorithm’s complexity is related to the number of I/O operations performed by fundamental operations such as scanning or sorting N contiguous elements. Those are $\text{scan}(N) = \theta\left(\frac{N}{B}\right)$ and $\text{sort}(N) = \theta\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$ where $M$ is the internal memory size.

This work presents an efficient algorithm, named TiledVS, to compute the viewshed of a point on terrains stored in external memory. The large number of disk accesses is optimized using a new library to manage the data swap between the external and internal memories. This new algorithm was compared against the most recent and most efficient published methods: EMViewshed [2] and io_radial2, io_radial3 and io_centrifugal [4]. Our new method is much simpler and, also, the tests showed that it is more than four times faster than all of them. Since processing large datasets can take hours, this improvement is significant.

2. DEFINITIONS AND RELATED WORKS

A terrain is a tridimensional surface $\tau$ where any vertical line intersects $\tau$ in, at most, one point. In this paper we will consider terrains represented by Raster Digital Eleva-
An observer is a point in the space from where the other terrain points (the targets) will be visualized. Both the observer and the targets can be at a given height above the terrain, respectively indicated by $h_o$ and $h_t$. Usually, it is assumed that the observer has a range of vision $\rho$, the radius of interest, which means that the observer can see points at a given distance $\rho$. Thus, a target $T$ is visible from $O$ if and only if the distance of $T$ from $O$ is, at most, $\rho$ and the straight line, the line of sight, from $O$ to $T$ is always strictly above the terrain. See Figure 1.

![Figure 1: Target’s visibility: $T_1$ and $T_3$ are not visible but $T_2$ is.](image)

The viewshed of $O$ corresponds to all points that can be seen by $O$. Since we are working with raster DEMs, we represent a viewshed by a square $(2\rho+1) \times (2\rho+1)$ matrix of bits where 1 indicates that the corresponding point is visible and 0 is not. By definition, the observer is in the center of this matrix.

Earlier works have presented different methods for viewshed computation. Among the methods for DEM terrains, we can point out the one proposed by Van Kreveld [9], and the one by Franklin et al., named RFVS [6]. These two methods are very efficient and are particularly important in this context because they were used as the base for some very recent and efficient methods for the viewshed computation in external memory: Fishman et al. [4] adapted Van Kreveld’s method, and Andrade et al. [2] adapted the RFVS method. This work also presents an IO-efficient adaptation of the RFVS method. Therefore, below we will give a short description of the RFVS method.

In that method, the terrain cells’ visibility is computed along rays connecting the observer to all cells in the boundary of a square of side $2\rho+1$ centered at the observer where $\rho$ is the radius of interest. That is, the algorithm creates a ray connecting the observer to a cell on the boundary of this square, and this ray is counterclockwise rotated around the observer following the cells in that boundary and the visibility of the cells in each ray is determined following the cells on the segment. Thus, suppose the segment is composed by cells $c_0, c_1, \cdots, c_k$ where $c_0$ is the observer’s cell and $c_k$ is a cell in the square boundary. Let $\alpha_i$ be the slope of the line connecting the observer to $c_i$ and let $\mu$ be the highest slope among all lines already processed, that is, when processing cell $c_i, \mu = \max\{\alpha_0, \alpha_2, \cdots, \alpha_{i-1}\}$. Thus, the target on $c_i$ is visible if and only if the slope of the line from $O$ to the target above $c_i$ is greater than $\mu$. If yes, the corresponding cell in the viewshed matrix is set to 1; otherwise, to 0. Also, if $\alpha_i > \mu$ then $\mu$ is updated to $\alpha_i$. We say that a cell $c_i$ blocks the visibility of the target above $c_j$ if cell $c_i$ belongs to the segment $[c_j \alpha_i]$ and $\alpha_i$ is greater or equal to the slope of the line connecting the observer to the target above $c_j$.

### 3. TiledVS Method

As mentioned in section 2, the RFVS sweeps the terrain cells rotating a ray connecting the observer cell to a cell in the boundary of a bounding box and the cells’ visibility is processed along this ray. Thus, the matrix access pattern presents a spatial locality of reference, that is, in a short time interval, the accessed cells are close in the matrix. However, this access pattern is not efficient in external memory since the cells which are close in the (bidimensional) matrix may not be stored close because, usually, a matrix is stored using a linear row-major order.

To reduce the number of non-sequential accesses, we present a new method, called TiledVS, where the basic idea is to adapt the RFVS algorithm to manage the access to the matrices stored in external memory using the library TiledMatrix [8]. In brief, this library subdivides the matrix in small rectangular blocks (tiles) which are sequentially stored in the external memory. When a given cell needs to be accessed, the whole block containing that cell is loaded into the internal memory. The library keeps some of these blocks in the internal memory using a data structure, named MemBlocks, which is managed as a “cache memory” and the replacement policy adopted is based on least recently used - LRU. That is, when a block is accessed it is labeled with a timestamp and if it is necessary to load a new block into the cache (and there is no room for this block), the block with smaller timestamp is replaced with the new block. When a block is evicted, it is checked if that block was updated (it is particularly important for the viewshed matrix); if any cell was updated then the block is written back to the disk.

Now, we will show that it is possible to define the MemBlocks size such that the adopted matrix partitioning associated with the LRU policy can be effective for the RFVS algorithm, that is, we will prove that this process will load a block in the cache, keep it there while it is accessed and it will be evicted only when it will be no longer needed.

In the following, we will suppose that the matrix partitioning creates square blocks with $\omega \times \omega$ cells and these blocks are grouped in vertical bands with $\omega$ columns of cells. See figure 2. And, given a ray $r$ defined by the RFVS algorithm, without loss of generality, in the demonstrations below, we will consider rays whose slope is, at most, $45^\circ$. For rays with greater slope just replace rows with columns.

![Figure 2: Matrix partitioning: (a) square blocks with 3 x 3 cells; (b) vertical bands with 3 columns.](image)

**Lemma 3.1.** Any ray intersects, at most, $\frac{\omega}{\rho} + 2$ bands where $\rho$ is the radius of interest (in number of terrain cells).

**Proof** For the viewshed computation, the RFVS algorithm defines a square bounding box of side $2\rho+1$ with the observer
on its center and creates rays connecting the observer to the center of the cells in the square border. Since any ray whose slope is, at most, 45° intersects ρ + 1 columns in this square, this ray intersects \( \lceil \frac{\rho + 1}{\omega} \rceil + 1 \) vertical bands. The additional +1 is because the observer may not be in the central column of a band (notice that, if the observer in the Figure 2(b) is moved one cell to the right, ray \( r_0 \) will cross the last band boundary and will intersect an additional band). Since \( \lceil \frac{\rho + 1}{\omega} \rceil = \lfloor \frac{\rho + 1}{\omega} \rfloor + 1 \) then \( \lfloor \frac{\rho + 1}{\omega} \rfloor + 1 \leq \frac{\rho}{2} + 2 \).

**Lemma 3.2.** Let \( r_k \) and \( r_{k+1} \) be two consecutive rays in the RFVS algorithm sweeping. Then these two rays intersect, at most, \( 2 \left( \frac{\rho}{2} + 2 \right) \) blocks.

**Proof** Since the RFVS algorithm uses the Bresenham rasterization method, there is exactly one cell for each column intersected by a ray. Let \( l_r \) and \( c_r \) be respectively the number of rows and columns intersected by a ray \( r \). As the ray slope is, at most, 45° then \( l_r \leq c_r \).

Given two consecutive rays \( r_k \) and \( r_{k+1} \), the vertical distance between them is, at most, one cell side - see Figure 3(a). As, for each vertical band, they intersect \( \omega \) columns, they can intersect, at most, \( \omega + 1 \) rows in that band. Thus, in each band, they can intersect, at most, two blocks (since the block height is \( \omega \) rows). Therefore, from Lemma 3.1, rays \( r_k \) and \( r_{k+1} \) can intersect, at most, \( 2 \left( \frac{\rho}{2} + 2 \right) \) blocks.

**Lemma 3.3.** Let \( r_0 \) be the first ray in the sweeping sequence. Given a block \( B \) not intersected by \( r_0 \), let \( r_k \) and \( r_{k+1} \) be two consecutive rays. If \( r_k \) intersects \( B \) and \( r_{k+1} \) doesn't, then no other ray after \( r_k \) will intersect block \( B \).

**Proof** It is straightforward from the fact that the algorithm uses a radial sweeping sequence and the blocks are convex. And it doesn’t work for the blocks intersected by ray \( r_0 \) because, considering the radial sweeping, these blocks can be intersected again by the last rays. See Figure 3(b).

**Theorem 3.4.** Given a block \( B \) not intersected by \( r_0 \), if the MemBlocks size (in number of blocks) is, at least, \( 2 \left( \frac{\rho}{2} + 2 \right) \) then the LRU policy will evict block \( B \) from MemBlocks only if it is no longer needed.

**Proof** Suppose that MemBlocks has \( 2 \left( \frac{\rho}{2} + 2 \right) \) slots to store the blocks. Let \( r_k \) and \( r_{k+1} \) be two consecutive rays such that \( r_k \) intersects block \( B \). At some point during the processing of ray \( r_k \), block \( B \) will start to be processed and it is stored in the MemBlocks (if \( r_k \) is the first ray intersecting block \( B \) then \( B \) will be loaded in MemBlocks). Now, if ray \( r_{k+1} \) also intersects block \( B \), this block needs to be processed again. But, the MemBlocks size is enough to avoid block \( B \) eviction because, let \( B'_0, B'_1, \ldots, B'_j \) be the sequence of blocks that need to be processed among the twice processing of \( B \), that is, it is the sequence of blocks to be processed after \( B \) in the ray \( r_k \) and before \( B \) in ray \( r_{k+1} \). From lemma 3.2, \( j \leq 2 \left( \frac{\rho}{2} + 2 \right) \) and since \( B \) is not included in the sequence then \( j < 2 \left( \frac{\rho}{2} + 2 \right) \). Thus, if MemBlocks size is \( 2 \left( \frac{\rho}{2} + 2 \right) \) then it has slots to store all blocks that need to be processed and \( B \) will not be evicted. In other words, the LRU policy will not evict block \( B \) because the distinct blocks that need to be accessed can be stored in MemBlocks.

On the other hand, if ray \( r_{k+1} \) doesn’t intersect block \( B \) then, from lemma 3.3, no other ray after \( r_k \) will intersect \( B \) and thus, it can be evicted since it is no longer needed.

There is a special situation for the blocks intersected by \( r_0 \) because, after being evicted, they can be loaded again when processing the last rays. But, notice that these blocks can be loaded, at most, twice. See Figure 3(b) where block \( B' \) is loaded in the processing of \( r_0 \), is evicted after the processing of \( r_m \) and it is loaded again when processing \( r_n \).

It is possible to demonstrate that the TiledVS algorithm does \( \theta(\text{scan}(N)) \) I/O operations and takes \( \theta(N) \) time to process a terrain with \( N \) cells considering that the memory can store \( 2 \left( \frac{\rho}{2} + 2 \right) \) blocks. This complexity works even if the radius of interest \( \rho \) is large to cover the whole terrain.

![Figure 3: (a) Blocks intersected by two consecutive rays; (b) Block \( B' \) is loaded because of ray \( r_0 \), is evicted after ray \( r_m \), and loaded again for ray \( r_n \).](image)

### 4. EXPERIMENTAL RESULTS

The TiledVS method was implemented in C++ and compiled with g++ 4.3.4. It was compared against the most efficient algorithms recently described in literature: io-radial2, io-radial3 and io-centrifugal, proposed by Fishman et. al. [4], and EMViewshed, proposed by Andrade et al. [2].

Lacking access to Fishman’s programs, we compared our algorithm to his published results. We executed our algorithm using the same datasets and also a same platform as that one used by those authors, i.e. a computer with an Intel Core i7 2 Duo E7500 2.93GHz processor, 4GiB of RAM memory, and a 5400RPM SATA HD (Samsung HD103S3) which was rebooted with 512MiB RAM. The operational system used was Linux, Ubuntu 10.04 32bits distribution.

Our results are presented in Table 1 and Figure 4 where we reproduce the times presented in [4]. Notice that our algorithm is faster than the others in all situations and, on huge terrains, it is about 4 times faster. Also, the table includes the processing time of our algorithm on very huge terrains generated by interpolation of Region02.

We also compared our new algorithm TiledVS against our previous one EMViewshed [2]. We used different datasets generated from two distinct USA regions sampled at different resolutions using 2 bytes per elevation value. The results are presented in Table 2. Note that our new algorithm TiledVS is about 7 times faster than our previous one.

Table 3 presents the TiledVS running time (in seconds) for different terrain sizes using only 128MiB and 512MiB of RAM. As can be noticed, our algorithm is scalable to data that is much bigger than the machine internal memory.

### 5. CONCLUSION

We presented a new algorithm for viewshed computation on huge grid terrains stored in external memory. Our new method uses a special data structure to manage the data.
Table 1: Running time (in seconds) and CPU utilization (in parentheses) for io-radial2, io-radial3, io-centrifugal and TiledVS with 512MiB RAM.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>cols x rows</th>
<th>GB</th>
<th>io-radial2</th>
<th>io-radial3</th>
<th>io-centrifugal</th>
<th>TiledVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cumberlands</td>
<td>8 704</td>
<td>0.25 GiB</td>
<td>72 (84%)</td>
<td>104 (91%)</td>
<td>35 (49%)</td>
<td>17 (97%)</td>
</tr>
<tr>
<td>USA DEM 6</td>
<td>13 500</td>
<td>0.92 GiB</td>
<td>2,804 (13%)</td>
<td>458 (84%)</td>
<td>115 (57%)</td>
<td>85 (77%)</td>
</tr>
<tr>
<td>USA DEM 2</td>
<td>11 000</td>
<td>1.04 GiB</td>
<td>1,883 (34%)</td>
<td>735 (87%)</td>
<td>121 (54%)</td>
<td>98 (77%)</td>
</tr>
<tr>
<td>Washington</td>
<td>31 866</td>
<td>3.97 GiB</td>
<td>13,780 (22%)</td>
<td>3,008 (89%)</td>
<td>676 (41%)</td>
<td>386 (73%)</td>
</tr>
<tr>
<td>SRTM1-region03</td>
<td>50 401</td>
<td>8.11 GiB</td>
<td>37,982 (16%)</td>
<td>6,644 (81%)</td>
<td>2,845 (17%)</td>
<td>991 (69%)</td>
</tr>
<tr>
<td>SRTM1-region04</td>
<td>82 801</td>
<td>11.10 GiB</td>
<td>8,834 (34%)</td>
<td>5,341 (11%)</td>
<td>1,347 (69%)</td>
<td>1,347 (69%)</td>
</tr>
<tr>
<td>Region02 interp.</td>
<td>100 000</td>
<td>28.44 GiB</td>
<td>26,193 (61%)</td>
<td>12,186 (18%)</td>
<td>5,034 (71%)</td>
<td>5,079 (77%)</td>
</tr>
</tbody>
</table>

transference between the internal and external memory reducing the number of I/O operations. For terrains with \( N \) cells, its I/O complexity is \( \Theta(\text{scan}(N)) \).

Table 2: Running time (seconds) for EMViewshed (EMVS) and TiledVS with 1024MiB RAM.

<table>
<thead>
<tr>
<th>Size cols x rows</th>
<th>EMVS</th>
<th>TiledVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 000 x 30 000</td>
<td>1.68</td>
<td>727</td>
</tr>
<tr>
<td>40 000 x 40 000</td>
<td>2.98</td>
<td>3168</td>
</tr>
<tr>
<td>50 000 x 50 000</td>
<td>4.65</td>
<td>5701</td>
</tr>
<tr>
<td>60 000 x 60 000</td>
<td>6.71</td>
<td>8961</td>
</tr>
</tbody>
</table>

Table 3: TiledVS running time (seconds) using a RAM memory with 128MiB and 512MiB.

<table>
<thead>
<tr>
<th>Terrain Size cols x rows</th>
<th>RAM Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>37 000 x 37 000</td>
<td>634</td>
</tr>
<tr>
<td>52 000 x 52 000</td>
<td>1277</td>
</tr>
<tr>
<td>73 500 x 73 500</td>
<td>3324</td>
</tr>
<tr>
<td>104 000 x 104 000</td>
<td>7511</td>
</tr>
</tbody>
</table>

The algorithm was compared against the most recent and efficient algorithms in the literature and, as the tests showed, it was faster than all others. In general, it was about 4 times faster and this improvement is significant because processing huge terrains can take hours. Also, it is much simpler.

Additionally, the algorithm was able to process huge terrains using few RAM memory. For example, the viewshed of a whole terrain of size 40 GiB, using 128 MiB RAM was computed in 7511 seconds.

The algorithm source code (in C++) is available and distributed under Creative Common GNU GPL license at http://www.dpi.ufv.br/~marcus/TiledVS.htm

Acknowledgments

This research was partially supported by FAPEMIG, CAPES, CNPq, GAPSO and NSF grants CMMI-0835762, IIS-1117277.

6. REFERENCES


