Computing Visibility on Terrains in External Memory

Herman Haverkort* Laura Toma† Yi Zhuang‡

Abstract

We describe a novel application of the distribution sweeping technique to computing visibility on terrains. Given an arbitrary viewpoint v, the basic problem we address is computing the visibility map or viewshed of v, which is the set of points in the terrain that are visible from v. We give the first I/O-efficient algorithm to compute the viewshed of v on a grid terrain in external memory. Our algorithm is based on Van Kreveld's $O(n \lg n)$ time algorithm for the same problem in internal memory. It uses $O(sort(n))$ I/Os, where $sort(n)$ is the complexity of sorting n items of data in the I/O-model. We present an implementation and experimental evaluation of the algorithm. Our implementation clearly outperforms the previous (in-memory) algorithms and can compute visibility for terrains of up to 4 GB in a few hours on a low-cost machine.

1 Introduction

In this paper we consider the problem of computing visibility on very large terrains in external memory. Given an arbitrary viewpoint v and a terrain, the basic problem we address is computing the *visibility map* or *viewshed* of v, which is the set of points in the terrain that are visible from v (Figure 1). Visibility has applications in graphics and game design, and mainly in Geographic Information Systems (GIS), ranging from path planning, navigation, landscaping, to placement of fire towers, radar sites and cellphone towers [12, 16].

Visibility has been widely studied in computational geometry and graphics; for a survey of the various problems and results see Cole and Sharir [9] or De Floriani and Magillo [11]. Recently, researchers have obtained a number of results on visibility addressing the watchtower problem and terrain guarding [2, 8]. The standard terrain model used in geometry is the *polyhedral terrain*, which is a continuous piecewise linear function defined over the triangles of a triangulation in the plane. In GIS, however, the most common representation of terrain data is the grid, which samples the elevation of a terrain with a uniform grid and records the values in a 2D-matrix. Thus we are interested in computing visibility on grid terrains.

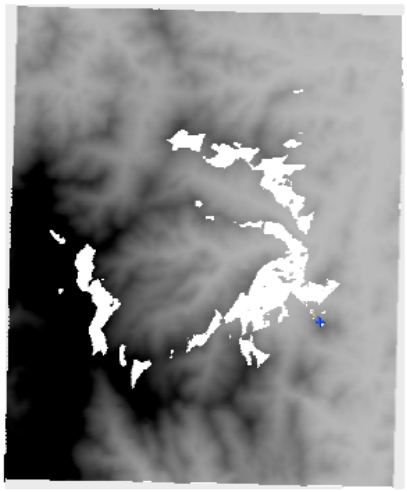


Figure 1: The viewshed of a point on a terrain is shown in white.

In the recent years an increasing number of applications involve high-resolution massive terrain data that is becoming available from remote sensing technology. NASA's Shuttle Radar Topography Mission (SRTM) acquired terrain data with one sample per 30m ($900m^2$) in 2002, in total about 10 terabytes of data. More recently, LIDAR and real-time kinematic global positioning system technology offer the capability to collect geospatial data at 1m-resolution.

When working with massive data, only a fraction of the data can be held in the main memory of a computer. Thus, the transfer of data between main memory and disk, rather than the computation as such, is usually the performance bottleneck. One approach to improving performance is to design *external memory* or *I/O-efficient algorithms* — algorithms that specifically optimize the number of block transfers between main memory and disk. In this paper we present the design and experimental evaluation of an I/O-efficient algorithm for viewshed computation on very large grid terrains.

*Dept. of Mathematics and Computer Science, Technische Universiteit Eindhoven, The Netherlands. herman@haverkort.net

†Dept. of Computer Science, Bowdoin College, Maine, USA. ltoma@bowdoin.edu

‡Dept. of Computer Science, Bowdoin College, Maine, USA. yzhuang@bowdoin.edu

1.1 Problem definition. Let T be a terrain represented as a grid of n square cells. For computational purposes, we assume that the entire grid cell is represented by its center point. In particular, we assume that we are given the elevation of each grid cell's center point. The *line of sight* from a viewpoint v to a grid cell Q is the line segment that connects v to the center q of Q. We use the definition of visibility also applied by Van Kreveld [18]: a grid cell with center q is visible from v if the line of sight vq does not cross any grid cell that appears higher on the horizon—more precisely, if the line segment vq does not cross any grid cell with center q' such that the slope of vq' is higher than the slope of vq. Note that this is a discrete visibility model, where a cell is either completely visible or not. There is no concept of partial visibility: A corner of a high cell is assumed to hide another grid cell Q from the viewpoint v completely if it intersects the line of sight from v to the center q of Q.

Let v be an arbitrary viewpoint. With visibility defined as above, the *visibility map* or *viewshed* of v on a *grid* terrain is the set of all grid cells that are visible from v. In GIS, one is not only interested in computing boolean cell visibility, but also, for cells that are invisible from v, their vertical distance to visibility: how far the cells should be raised to become visible from v. Thus, for a given viewpoint, we want to compute a grid that records the distance of each cell to visibility. We call this the *visibility grid* of v. The viewshed of v is the set of cells with distance 0.

A straightforward approach to determine whether a cell is visible from a given viewpoint requires $O(n)$ time per cell, or $O(n^2)$ time for the entire grid—if the grid is square the bounds become $O(\sqrt{n})$ and $O(n\sqrt{n})$ respectively. The bound for the entire grid was improved to $O(n \lg n)$ by Van Kreveld [18] using plane sweeping. There have been many other papers from the GIS community dealing with visibility computation on grids; see for example the papers by Fisher, Franklin, and Ray [13, 14, 15, 16], and the references therein. They describe experimental studies for fast implementations of approximate visibility computations and explore various trade-offs between speed and accuracy; they do not guarantee worst-case bounds better than the straightforward one, nor do they prove any bounds on the quality of the approximation. An overview of the visibility results on grids, as well as on other terrain representations (triangulations), can be found in the work by De Floriani and Magillo [11, 12]. To our knowledge no I/O-efficient results have been reported for visibility computations so far.

1.2 I/O-Model and related I/O-work. We use the standard two-level I/O-model by Aggarwal and Vitter [3]. The model defines two parameters: M is the size of internal memory, and B the size of a disk block. An *Input/Output* (or: *I/O*) is the operation of transferring a block of data between main memory and disk. The *I/O-complexity* of an algorithm is the number of I/Os it performs. The basic bounds in the I/O-model are those for scanning and sorting. The *scanning bound*, $scan(n) = \Theta(\frac{n}{B})$ is the number of I/Os necessary to read n contiguous items from disk. The *sorting bound*, $sort(n) = \Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$ represents the number of I/Os required to sort n contiguous items on disk [3]. For all realistic values of n, B, and M, $scan(n) < sort(n) \ll n$.

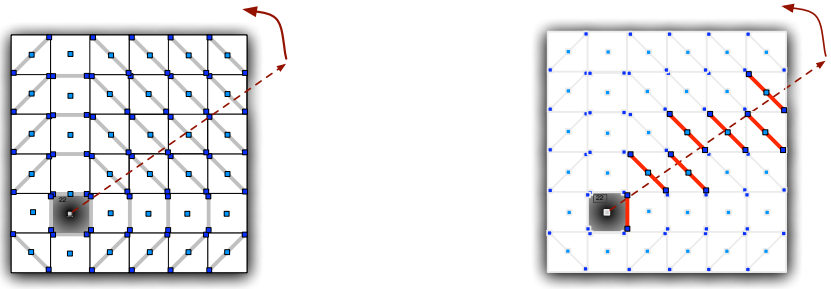
I/O-efficient algorithms have been developed for many problems encountered in GIS, like variants of segment intersection and range searching. The first results were obtained by Goodrich et al. [17], who developed the technique of *distribution sweeping* as an external memory version of the powerful plane sweep paradigm in internal memory. Distribution sweeping was developed for the problem of orthogonal line segment intersection, and has been subsequently applied to other GIS problems, like the (red-blue) line segment intersection and map overlay [5]. For a survey of distribution sweeping and I/O-efficient algorithms for GIS see Arge [4] and Van Kreveld et al. [19].

1.3 Our results. In this paper we show a novel application of the distribution sweeping technique to the computation of visibility on grid terrains. In Section 3 we present an $O(sort(n))$-I/O algorithm for the computation of the visibility grid of a viewpoint in external memory. The algorithm is based on distribution sweeping and the plane sweep algorithm by Van Kreveld [18]. In Section 4 we present an implementation and experimental evaluation of the algorithm. The algorithm is not only theoretically optimal, but simple and practical: our algorithm clearly outperforms the previously known (in-memory) algorithm and can compute visibility on up to 4 GB terrains in less than 5 hours.

2 Van Kreveld's Algorithm

In this section we describe the $O(n \lg n)$ time algorithm by Van Kreveld [18], which gives the best known upper bound for visibility grid computation in internal memory.

The algorithm uses the plane sweep technique. Given a grid and a viewpoint v, the basic idea is to rotate a (half) line around v and compute the visibility of each cell in the terrain when the sweep-line passes over its center. For this we maintain a data structure



(a) Each cell is marked with its 3 events. (b) Active cells.

Figure 2: Van Kreveld's plane sweep algorithm.

(the *active* structure) that, at any time in the process, contains the cells currently intersected by the sweep line (the *active cells*); refer to Figure 2(b). When a cell is intersected by the sweep-line, it is inserted in the active structure; when a cell stops being intersected by the sweep-line, it is deleted from the active structure. When the center of a cell is intersected by the sweep line, the active structure is queried to find out if that cell is visible. Thus, each cell in the grid has three associated events: when it is first intersected by the sweep-line and entered in our data structure, when the sweep-line passes over its center, and when it is last intersected by the sweep-line and removed from our data structure; refer to Figure 2(a).

Consider a cell Q in the grid, and let q be its centerpoint. The height above the horizon of Q with respect to the viewpoint v is defined as the height above the horizon of q, which is

$$\text{height}_q = \arctan \frac{e_q - e_v}{d_{vq}}$$

where e_q and e_v are the elevations of q and v, respectively, and d_{vq} is the distance between q and v. Note that this means a cell is treated as having constant height above the horizon with respect to the viewpoint.

A cell is visible if its line of sight from the viewpoint does not intersect any cell that is higher above the horizon. Therefore, a cell is visible if, when the sweep-line passes over its center, there are no active cells closer to the viewpoint and with greater height above the horizon. To query the active cells efficiently Van Kreveld [18] uses a balanced binary search tree for the active structure, in which the active cells are stored from left to right in order of increasing distance from their centers to the viewpoint. In addition, each node in the tree is augmented with the greatest height above the horizon in the subtree rooted at that node; for leaves, this is simply the height above the horizon of the cell stored at the leaf.

The augmented binary search tree can be maintained in $O(\lg n)$ time per insertion and deletion in a straightforward way. To determine whether a cell Q is visible, we search for it in the tree. As we follow the search path, all the cells that could obstruct visibility are to the left of the path; to find the cell among them that appears highest above the horizon, we collect the maximum height stored in all the subtrees to the left of the path in $O(\lg n)$ time. From here we can infer whether Q is visible, and, if not, what is its vertical distance to visibility. Overall, there are $3n$ events, and each is handled in $O(\lg n)$ time. If we wanted to compute only the viewshed of v, the algorithm would still take $\Theta(n \lg n)$ time: Irrespective of the size of the viewshed, we would still sort the events and traverse the entire grid. We have the following.

THEOREM 2.1. *(Van Kreveld [18]) The visibility grid of an arbitrary viewpoint can be computed in $O(n \lg n)$ time for a grid of size n.*

3 Computing Visibility in External Memory

Van Kreveld's algorithm uses four structures: the elevation grid, the visibility grid, the event list and the active structure. Even if the event list is stored as a stream on disk, and sorted I/O-efficiently, the algorithm would still use $\Omega(n)$ I/Os to maintain and query the active structure.

In Section 3.1 we describe simple modifications to the plane sweep algorithm to obtain I/O-efficiency, under the assumption that the active structure fits in memory. While the resulting algorithm is not guaranteed to be worst-case optimal, it widely extends the size of the problems that can be tackled in practice (as showed by the experimental evaluation). In Section 3.2 we extend the approach to an algorithm that needs only $O(sort(n))$ I/Os in the worst case.

3.1 The base case The inefficiency of Van Kreveld's algorithm is caused by three problems. First, the elevation grid is loaded in memory in row-column order, but the grid is accessed (read) in rotating sweep order to determine each cell's height above the horizon as it is entered or queried for in the active structure. Second, the visibility grid is loaded in memory in row-column order and is accessed (written) in sweep order. Third, there is little structure in the way in which the active structure is accessed.

The first problem can be solved, for example, by augmenting the events in the event list with information about the elevation of the cell, so that the input grid does not need to be accessed at all once the event list has been built. The second problem can be solved by recording the visibility of each cell in a list in sweep order, and sorting the list into grid order after completing the sweep. With these two modifications, the plane sweep does not need to load the input and output grid into memory. Assuming the event list is stored in a stream, the algorithm can now use all available memory for the active structure. If the active structure is small enough to fit in memory, the algorithm runs in $O(sort(n))$ I/Os.

If the active structure does not fit in memory, an immediate idea would be to implement it with a B-tree, but this would give a running time of $O(n \log_B n)$ I/Os. Below we explain how to do better.

3.2 An $O(sort(n))$ I/O algorithm. In this section we describe how to compute a visibility grid in worst-case $O(sort(n))$ I/Os without any assumptions on the input. The algorithm is based on the distribution sweeping technique, which we describe below.

Distribution sweeping. The general idea in distribution sweeping is to divide the input into $O(M/B)$ and $\Omega((M/B)^\epsilon)$ strips (slabs), each containing an equal number of input objects. Using these strips we decompose the solution to the problem into a part that can be found recursively in each strip, and a part that involves interactions between strips. The recursive part of the solution we can find by solving the problem recursively in each strip. The recursion stops when the strips are small enough to fit in main memory, which happens after $O(\log_{M/B} n)$ iterations. The challenging part in distribution sweeping is finding the part of the solution that involves interactions between strips. To get an algorithm that uses only $O(sort(n))$ I/Os we need to handle interactions in $O(n/B)$ I/Os in total per level of recursion. Two approaches for this problem have been described in the literature: handling the interactions by a plane sweep based on maintaining active lists for each strip (this approach was taken by Goodrich et al. [17] to compute intersections of orthogonal line segments), and using $O(\sqrt{M/B})$ fan-out and multi-slabs, used by Arge et al. [5] for red-blue line segment intersections. We show how to handle strip interactions for the visibility problem by combining a radial (rotational) and a concentric sweep.

Our algorithm follows the general approach of distribution sweeping described above: we divide the grid into $\Theta(M/B)$ radial "strips" (sectors) around the viewpoint. A cell can be *narrow*, that is, crossing at most one sector boundary, or *wide*, that is, crossing at least two sector boundaries and thus spanning at least one sector completely—refer to Figure 3). On every level of recursion, we first determine how *wide* cells affect the visibility of cells in the sectors spanned by them, and then compute the visibility grid recursively in each sector. The recursion stops when the number of cells gets small enough to run the base case algorithm described in Section 3.1 while keeping the active structure in memory. Below we detail the steps and their analysis.

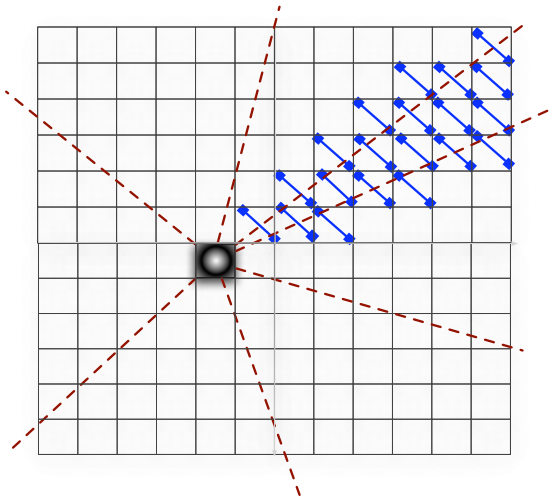


Figure 3: Distribution sweeping. The grid is divided into M/B radial slabs (sectors); cells may cross zero, one or more sector boundaries.

We start by scanning the grid to identify the events; for each cell Q, we determine the point $p_1 = (r_1, \theta_1)$ where Q will be hit by the rotating sweep line first, the point $p_2 = (r_2, \theta_2)$ that will be the last point of Q to intersect the sweep line, and the center point $q = (r_q, \theta_q)$ of Q. The points are represented in polar coordinates with respect to the viewpoint. As we scan the grid, we create two lists of events, E_r and E_c. The list E_r

consists of all the event points, in the entire grid. The list E_c contains two copies of each cell Q: one copy is marked as *query* and stored with the center point q of Q; the other copy is marked as *obstacle* and stored with the points p_1 and p_2 where the rotating sweep line will first and last intersect it. With each cell we also store the height above the horizon of its center point.

After we finish scanning all the cells, we sort E_r by radial order around the viewpoint, and the cells in E_c by distance from the viewpoint to their center points. We sort E_r and E_c only once at the beginning—we construct the sorted event lists for recursive calls by distributing the events of E_r and E_c among the recursive calls while keeping them in order.

Given the two sorted lists E_r and E_c, the algorithm proceeds in two phases: a *radial sweep*, which processes events from list E_r in order of the polar angle with respect to the viewpoint, and a *concentric sweep*, which processes the events from list E_c in increasing order of their distance from the viewpoint. The radial sweep is used to partition the events into approximately M/B equal-sized sectors. The concentric sweep is used to process wide obstacles and to distribute the query events and narrow obstacles. By processing events concentrically we are able to maintain, while distributing query events, if there is any wide obstacle closer to the viewpoint that may occlude the center of the query cell.

The details are slightly different, depending on whether we want to compute a visibility grid (distances to visibility) or only a viewshed (whether cells are visible or not). We first discuss the details for the viewshed computation.

The radial sweep: First, we need to determine approximately M/B sectors such that each sector contains $O(n/(M/B))$ event points. This is easily done by scanning E_r to identify the sector boundaries. While doing so we also compute, for each sector, a list of the events in that sector in radial order. We will use these lists later as we recurse on the sectors to find radial partitions within each sector.

The concentric sweep: During this sweep, we scan and process the events (queries and obstacles) in E_c in order of increasing distance from the viewpoint, and for each sector we construct a list of events in that sector in the same order. To do this I/O-efficiently, we keep, for each sector, a buffer of one block of data in memory. Initially the buffers are all empty. They will get filled with events during the sweep—whenever a buffer runs full, its contents are output to disk and the buffer is emptied.

Furthermore, we keep for each sector S the radii that form its boundaries in memory, and a variable $High_S$ that holds the greatest height above the horizon among all the wide obstacles that completely span that sector and that have been reached by the concentric sweep. Initially $High_S$ is set to $-\infty$ for each sector.

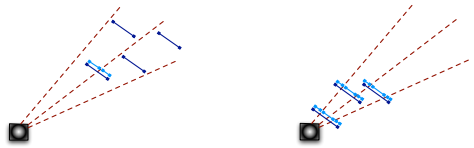
The concentric sweep now proceeds as follows. We go through the events in E_c in order of increasing distance from the viewpoint. Events may be queries or obstacles.

If the event is a query cell Q, we determine which sector S contains its center point, and check if the height of Q above the horizon is at least $High_S$. If so, we write Q to the event buffer of S; if not, we output Q to a list of points that have been found to be invisible, and do not copy it to any event buffer.

If the event is an obstacle E, it may intersect several sectors.

For each sector S that is intersected by E, but not completely spanned by it, we check if the height of E above the horizon is more than $High_S$. If so, we write E to the event buffer of S. If not, we ignore E for this sector, because it cannot occlude any query cells in S that are not already occluded by the wide obstacle that determined $High_S$.

For each sector S that is spanned by E (that is, E touches or intersects both radii that delimit S), we update $High_S$ by setting it to the maximum of S and the height above the horizon of (the center point of) E (this takes no I/O, as all the necessary information is kept in memory). As a result, all query cells in S that are occluded by E will be filtered out of the event stream of S.



(a) (b)

Figure 4: Segments corresponding to (a) narrow cells. (b) wide cells.

In recursion: After completing the concentric sweep, each sector is processed recursively. Throughout the recursion cells are marked as invisible and discarded. All query cells that are visible survive until the final level of recursion. The recursion stops when we can run the algorithm from Section 3.1 while keeping the active structure in memory.

Note that as a result of the early discards, the lists E_c and E_r for a sector may get out of sync. That is,

E_r may contain events whose corresponding cells have already been discarded in the course of a concentric sweep. This is not a problem, it just means that the size of each recursive problem may actually be less than $O(n/(M/B))$.

Computing a visibility grid: The algorithm above can be extended to compute the vertical distance to visibility for each invisible cell: As we process a query cell Q, we keep track of $High_Q$, the greatest height above the horizon among the wide obstacles that occluded (the center point of) Q through the recursive steps. When a query cell Q is occluded, we do not discard it, but we update its $High_Q$ variable, insert it in the event list of its sector, and let the recursive calls determine its distance to visibility.

Analysis: Initial sorting of the input into two event streams E_r and E_c, and sorting the output into grid order, takes $O(sort(n))$ I/Os. At each level of recursion a query is inserted in the event stream of at most one sector, and an obstacle is inserted in the event stream of at most two sectors. Hence both the radial and the concentric sweeps use $O(n/B)$ I/Os per level of recursion, and there are $O(\log_{M/B} n/M)$ levels of recursion. Thus the entire algorithm uses $O(sort(n))$ I/Os and linear space. We have the following.

THEOREM 3.1. *The visibility grid of an arbitrary viewpoint on a grid of size n can be computed with $O(n)$ space and $O(sort(n))$ I/Os.*

Note that the radial sweep and the sorted list E_r are only necessary for simplicity. It is possible to partition the events in E_c radially by using an $O(n/B)$ pivot-finding algorithm, as in Aggarwal and Vitter [3]. Since the algorithm is complicated, maintaining the events in radial order throughout the recursion is probably more likely to be efficient in practice. If the boundary of the grid has an easy shape, for example if the grid is rectangular and all cells are valid, one may simply compute the sector boundaries analytically in memory, instead of determining them by a radial sweep or pivot-founding algorithm.

4 Experimental results

This section presents an experimental comparison of the efficiency of our I/O-efficient algorithm (`ioviewshed`) to compute a viewshed with Van Kreveld's plane sweep algorithm (`kreveld`) and with a module that implements this functionality in the widely used open source GIS `GRASS` (`r.los`).

We implemented the algorithms in C++ using the g++ 4.0.1 compiler with optimization level -O3. For the I/O-efficient algorithm we used an external memory library `IOStreams`. This library is derived from the `TPIE` library [7], and provides basic file functionality along with an I/O-optimal external mergesort [3] and an I/O-efficient priority queue [6].

`kreveld`**:** The implementation of the `kreveld` algorithm is straightforward. It maintains the elevation grid (input) and visibility grid (output) in memory, together with an event array and an active structure. For the active structure we used a standard implementation of red-black trees [10].

`ioviewshed`**:** The implementation starts by loading the input elevation grid into a stream, building the event stream, and then sorting it. During the sweep `ioviewshed` labels each cell as visible or not. For each cell (i, j) this information is written to an output stream; at the end, the output stream is sorted in (i, j) order to produce the output grid. We observe that, most of the time, only a small fraction of the terrain is visible for very large terrains; when we compute the viewshed, we optimize the algorithm by recording only the cells that are visible, and assume that cells not recorded are invisible to the observer. This optimization significantly decreases the time needed to sort the output.

We found that in all of our experiments, with data sets up to 4 GB, we could run the basic modified plane sweep algorithm on the complete data set and still fit the active structure completely in main memory (see Table 2). As a result, `ioviewshed` never goes into recursion.

`GRASS`**:** The open source GIS `GRASS` provides viewshed computation via a module called `r.los`. The module uses the straightforward $O(n^2)$ algorithm. To handle the I/O-bottleneck it uses a `GRASS` memory- and I/O-management tool called the *segment library*. This library mimics a virtual memory manager that moves data between memory and disk in segments so that the size of memory `r.los` can access is limited by the disk space only. From the experiments we see that `r.los` *always* runs, and never aborts with a `malloc` fail. However, it is extremely slow. We include this module in our experiments because `GRASS` is the most widely used open source GIS; but mainly because this module illustrates the thrashing of the straightforward internal memory algorithm when the amount of virtual memory is infinite.

Datasets: Table 1 describes the data sets, representing real terrains of various characteristics ranging from 1 to over 1 000 million elements. Real terrains of-

ten contain points for which the elevation is unknown or invalid; these points are marked as invalid, or no-data points. Typically invalid points are marked with a special elevation value and are ignored when processing the terrain, so the amount of invalid data on a terrain can significantly influence the running time. The amount of valid data for each terrain is also given in Table 1.

Data set	Points	Size	Valid
Kaweah	$1.6{\cdot}10^6$	7 MB	56 %
Puerto Rico	$6.1{\cdot}10^6$	25 MB	19 %
Sierra Nevada	$9.5{\cdot}10^6$	40 MB	96 %
Hawaii	$30 \cdot 10^6$	119 MB	7 %
Cumberlands	$67 \cdot 10^6$	267 MB	27 %
Lower NE	$78 \cdot 10^6$	311 MB	36 %
Midwest USA	$280 \cdot 10^6$	1 122 MB	86 %
Washington	$1\,066 \cdot 10^6$	4 264 MB	95 %

Table 1: Size of terrain data sets. The valid-count excludes undefined (e.g. ocean) data values.

Platform: All experiments were run on Apple Power Macintosh G5 computers with dual 2.5 GHz processors, 512 KB L2 cache per processor, 1 GB RAM, and a Maxtor serial-ATA 7200 RPM hard drive. However only one processor is used since the algorithms are single-threaded. We allowed our algorithm to use 800 MB of the available memory. We also ran experiments with 256 MB RAM (allowing our algorithm to use 200 MB), in which case we booted the machines with this amount of memory.

Results: Table 3 summarizes the overall running time of `r.los`, `kreveld` and `ioviewshed` on the test datasets, together with the corresponding CPU utilization in each case. The viewpoints were selected to be in similar positions with respect to the size and shape of the grid, and for each dataset, the viewpoint was the same across tests with different algorithms. The running times are depicted in Figure 5.

`r.los` is by far the most inefficient of the three algorithms. It takes about an hour on our smallest data set (kaweah), and we let it run for two weeks on Hawaii ($n = 30 \cdot 10^6$ points). The inefficiency of `r.los` is probably due to the overhead of the `GRASS` segment library and to the straightforward $O(n^2)$ algorithm ($O(n\sqrt{n})$ on square grids). This also explains why the algorithm is much slower on Puerto Rico than on Sierra, even though the latter is larger: Puerto Rico is long (aspect ratio 3.2), while Sierra is almost square (aspect ratio 1.4).

As expected, our implementation of `kreveld` performs very well as long as the data fits in main memory. For a grid of n points, the size of the memory necessary during the algorithm is about $11n$ 32-bit integers (elevation grid, visibility grid, and event array). If the data structures fit completely in memory, the algorithm finishes in seconds and its CPU utilization is 100%. On the Hawaii data set it starts thrashing: the CPU utilization drops down to 39%, and the running time becomes 736 seconds. On a larger dataset the required amount of memory exceeds the available virtual memory, so the algorithm does not finish (`malloc` failure).

In contrast, the performance of `ioviewshed` scales nicely with increasing problem size. It manages to compute visibility grids of approximately 10^9 points in about 4.5 hours. This involves sorting an event stream of approximately 60 GB (Table 2), which constitutes the dominant part of the running time. `ioviewshed` can be optimized further by plugging in one of the standard I/O libraries like TPIE [7] or `STXXL` [1].

Data set	Active str.	Event str.
Kaweah	93 KB	92 MB
Puerto Rico	236 KB	343 MB
Sierra Nevada	234 KB	543 MB
Hawaii	410 KB	1 716 MB
Cumberlands	589 KB	3 833 MB
Lower NE	634 KB	4 463 MB
Midwest USA	1 413 KB	16 022 MB
Washington	2 344 KB	60 998 MB

Table 2: `ioviewshed` statistics: maximum size of the active structure and size of the event stream.

As expected, `ioviewshed` is slower than `kreveld` when the data set is small. A practical implementation of `ioviewshed` could detect this situation and run `kreveld` instead—thus always achieving the best running time of `kreveld` and `ioviewshed`.

We also ran experiments with 256 MB of RAM. While a main memory size of 256 MB is certainly not realistic, it illustrates the behaviour and trends of algorithms as the difference between dataset and memory size increases. The total running times for the three algorithms with 256 MB are shown in Figure 6 and Table 4. The results are very similar to those with 1 GB—the crossover point between the internal and external algorithms moves to the left in this case.

Figures 7 and 8 show the performance of `kreveld` and `ioviewshed` comparatively at 256 MB and 1 GB RAM. With 256 MB `kreveld` processes Kaweah in a few seconds, as with 1 GB. However, it starts thrashing (earlier) on Puerto Rico: 112 seconds with 38% CPU utilization as opposed to 38 seconds, 100% CPU utilization at 1 GB. This is not surprising, as the amount of memory

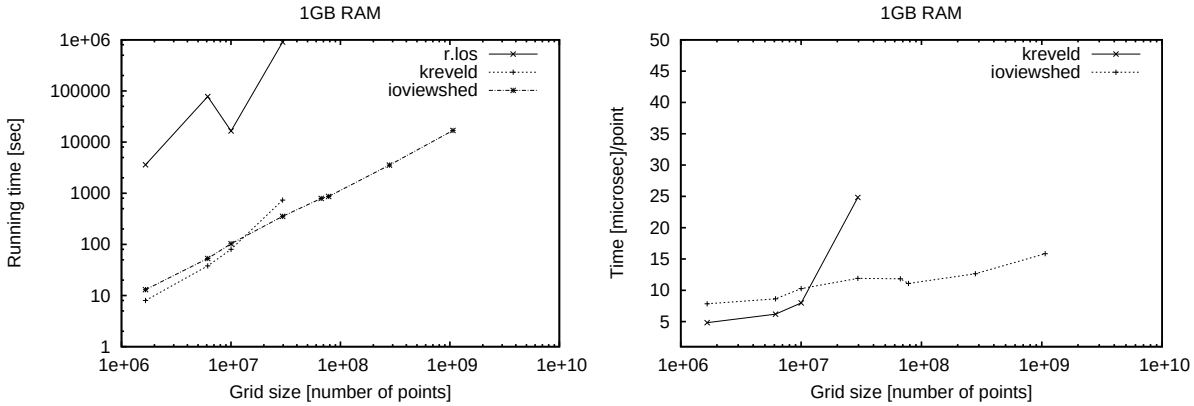


Figure 5: Running time of `r.los`, `kreveld` and `ioviewshed` at 1 GB RAM. (a) Total. (b) Per item.

Data set	`r.los`	`kreveld`	`ioviewshed`
Kaweah	2 928	8 (100 %)	13 (84 %)
Puerto Rico	78 778	38 (100 %)	53 (78 %)
Sierra Nevada	16 493	80 (95 %)	102 (67 %)
Hawaii	>1 200 000	736 (39 %)	353 (63 %)
Cumberlands		`malloc` fails	791 (63 %)
LowerNE			865 (64 %)
Midwest USA			3 546 (64 %)
Washington			16 895 (68 %)

Table 3: Running times (seconds) and CPU-utilization (in parentheses) at 1 GB RAM.

Data set	`r.los`	`kreveld`	`ioviewshed`
Kaweah	2 984	7 (100 %)	13 (77 %)
Puerto Rico	78 941	112 (38 %)	66 (60 %)
Sierra Nevada	19 140	211 (29 %)	115 (57 %)
Hawaii	>1 200 000	1270 (27 %)	364 (63 %)
Cumberlands		`malloc` fails	768 (62 %)
LowerNE			916 (62 %)
Midwest USA			4 631 (52 %)
Washington			40 734 (30 %)

Table 4: Running times (seconds) and CPU-utilization (in parentheses) at 256 MB RAM.

needed to process Puerto Rico is $11 \cdot 6.1 \cdot 10^6 \cdot 4\,\mathrm{B} = 268\,\mathrm{MB}$, which does not fit in main memory. The thrashing gets more pronounced for Sierra and Hawaii, where the CPU utilization drops to 29% and 27%, respectively. The `malloc` failure occurs on Cumberlands.

The performance of `ioviewshed` is practically the same at 256 MB and 1 GB for the smaller datasets. On the largest dataset, Washington, the running time increases from 16 895 seconds with 1 GB to 40 734 seconds with 256 MB. The cause of this increase is that sorting the event stream takes more time (60% of the total running time with 1 GB and 84% of the total running time with 256 MB). Using a more customized I/O-library may improve the running time significantly.

References

[1] http://stxxl.sourceforge.net.

[2] P. K. Agarwal, S. Bereg, O. Daescu, H. Kaplan, S. Ntafos, and B. Zhu. Guarding a terrain by two watchtowers. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 346–355, 2005.

[3] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] L. Arge. External-memory algorithms with applications in geographic information systems. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, pages 213–254. Springer-Verlag, LNCS 1340, 1997.

[5] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995.

[6] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *ACM J. Experimental Algorithmics*, 6, 2001. Article 1.

[7] L. Arge, R. D. Barve, D. Hutchinson, O. Procopiuc, L. Toma, J. Vahrenhold, D. E. Vengroff, and R. Wickremesinghe. TPIE user manual and reference, edition 1.0. Duke University, NC, "http://www.cs.duke.edu/TPIE/", 2005. (In Preparation).

[8] B. Ben-Moshe, M. J. Katz, and J. S. B. Mitchell. A constant-factor approximation algorithm for optimal terrain guarding. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 515–524, 2005.

[9] R. Cole and M. Sharir. Visibility problems for polyhedral terrains. *J. Symb. Comput.*, 7(1):11–30, 1989. ISSN 0747-7171.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., second edition, 2001.

[11] L. de Floriani and P. Magillo. *Geographic Information Systems: Principles, Techniques, Management and Applications*, chapter Intervisibility of Terrains, pages 543–556. John Wiley and Sons, 1999.

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

[13] P. Fisher. Algorithm and implementation uncertainty in viewshed analysis. *International Journal of GIS*, 7:331–347, 1993.

[14] P. Fisher. Stretching the viewshed. In *Proc. Symposium on Spatial Data Handling*, pages 725–738, 1994.

[15] W. R. Franklin. Siting observers on terrain. In *Proc. Symposium on Spatial Data Handling*, 2002.

[16] W. R. Franklin and C. Ray. Higher isn't necessarily better: Visibility algorithms and experiments. In *Proc. Symposium on Spatial Data Handling*, pages 751–763, 1994.

[17] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[18] M. van Kreveld. Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals. In *Proc. Symposium on Spatial Data Handling*, pages 15–27, 1996.

[19] M. van Kreveld, J. Nievergelt, T. Roos, and P. W. (Eds.). *Algorithmic Foundations of GIS*. LNCS 1340. Springer-Verlag, 1997.

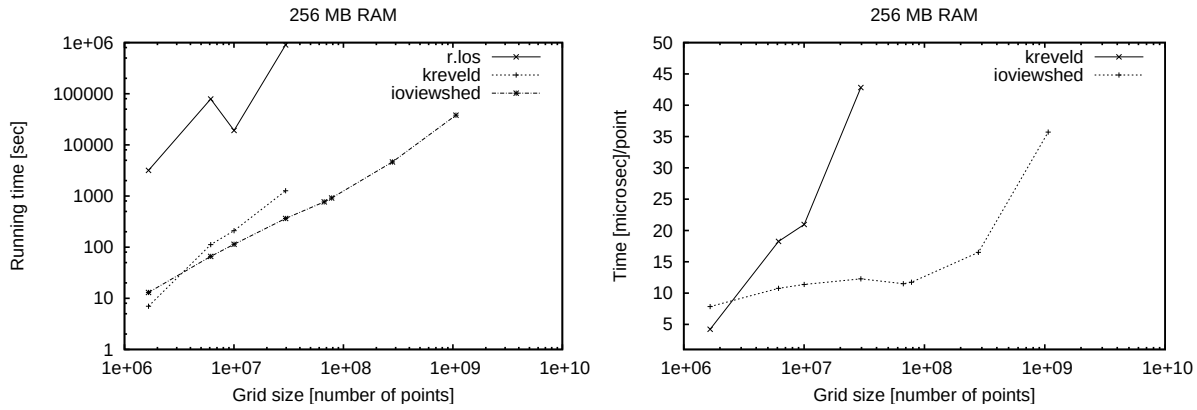


Figure 6: Running time of `r.los`, `kreveld` and `ioviewshed` at 256 MB RAM. (a) Total. (b) Per item.

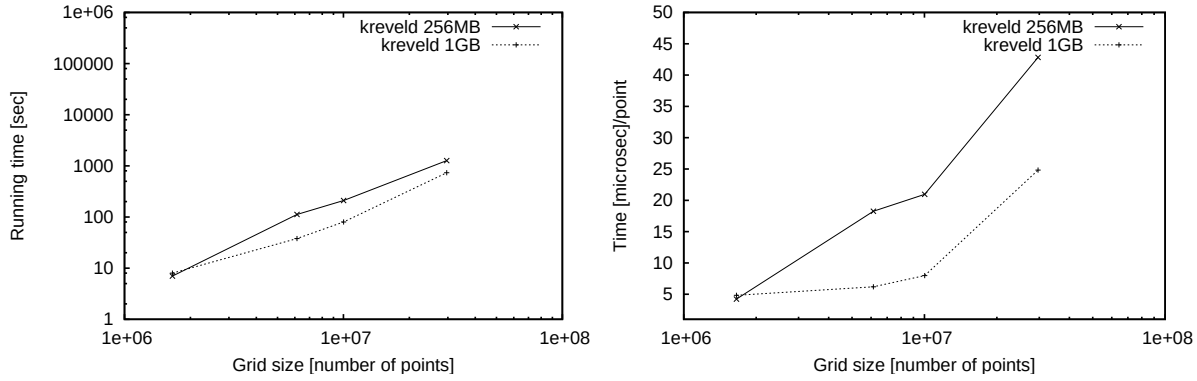


Figure 7: Running time of `kreveld` at 256 MB and 1 GB RAM. (a) Total. (b) Per item.

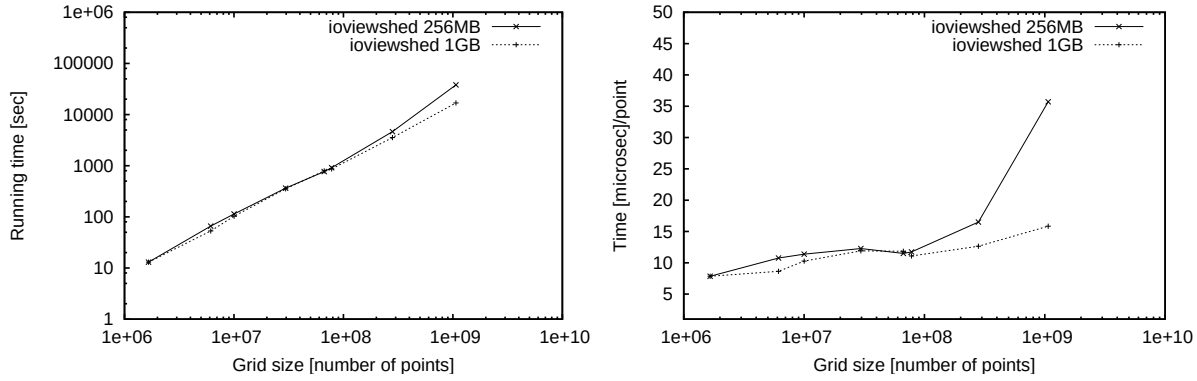


Figure 8: Running time of `ioviewshed` at 256 MB and 1 GB RAM. (a) Total. (b) Per item.