Improved Visibility Computation on Massive Grid Terrains

[Extended Abstract]

Jeremy Fishman
Bowdoin College
Brunswick, ME, USA
jfishman@bowdoin.edu

Herman Haverkort
Eindhoven Univ. of Techn.
Eindhoven, the Netherlands
cs.herman@haverkort.net

Laura Toma
Bowdoin College
Brunswick, ME, USA
ltoma@bowdoin.edu

ABSTRACT
This paper describes the design and engineering of algorithms for computing visibility maps on massive grid terrains. Given a terrain \( T \), specified by the elevations of points in a regular grid, and given a viewpoint \( v \), the visibility map or viewshed of \( v \) is the set of grid points of \( T \) that are visible from \( v \). We describe three new algorithms to compute the viewshed for any given terrain \( T \) and viewpoint \( v \).

The first two algorithms “sweep” the terrain by rotating a ray around the viewpoint while maintaining the terrain profile along the ray. On a terrain of \( n \) grid points, these algorithms run in \( O(n \log n) \) time and \( O(sort(n)) \) I/Os in the I/O-model of Aggarwal and Vitter. The difference between the two algorithms is in the preprocessing before the sweep: the first algorithm sorts the grid points into concentric bands around the viewpoint; the second algorithm sorts the grid points into sectors around the viewpoint. The third algorithm sweeps the terrain centrifugally, growing a star-shaped region around the viewpoint while maintaining the approximate visible horizon of the terrain within the swept region. This algorithm runs in \( O(n) \) time and \( O(scan(n)) \) I/Os and is cache-oblivious.

We tested our algorithms on NASA SRTM data, and found that our fastest new algorithm computes the viewshed of a terrain of 7.6 billion points (28.4 GiB) in 203 minutes on a machine with 0.5 GiB RAM and a laptop-speed hard drive. Depending on the data set, the new algorithm is 20 to 50 times faster than the algorithm from our previous work.

Categories and Subject Descriptors
F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems—Geometrical problems and computations; I.3.5 [Computing Methodologies]: Computational Geometry and Object Modeling—Geometric algorithms, languages, and systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

General Terms
Algorithms, Design, Experimentation, Performance

1. INTRODUCTION
The last decade witnessed an explosion in the availability of terrain data for geographic information systems (GIS). In 2002, for example, NASA’s Shuttle Radar Topography Mission (SRTM) acquired 30 m-resolution terrain data for the entire USA, in total approximately 10 terabytes of data. With more recent technology it is possible to acquire data at sub-meter resolution. This brings tremendous increases in the size of the datasets that need to be processed: Washington state at 1 m resolution, using 4 bytes for the elevation of each sample, would total 689 GiB of data; Ireland would be 262 GiB—only counting elevation samples on land.

One of the applications of terrain elevation models in GIS is the computation of visibility maps: the part of the terrain that is visible from a given viewpoint (see Figure 1). Visibility is a well-known problem, with numerous applications ranging from planning the placement of communication towers or watchtowers, to planning buildings and roads such that they have a good view or such that they do not spoil somebody else’s view, to finding routes on which you can travel while seeing a lot, or without being seen. In this paper we discuss the computation of visibility maps on grid terrains. A grid terrain, likely the most common terrain elevation model in GIS, is a matrix with elevation values for points in a regular grid on the surface of the earth.

Since grid terrains may be as large as hundreds of gigabytes, they often do not fit in the main memory of a computer at once. Hence, visibility computations require efficient algorithms that scale well and are designed to minimize “I/O”: the swapping of data between main memory and disk. Therefore we assess the efficiency of algorithms in this paper not only by studying the number of computational steps they need and by measuring their running times in practical experiments, but also by studying how the number of I/O-operations grows with the input size. To this end we use the standard model that was defined by Aggarwal and Vitter [1]. In this model, a computer has a memory of size \( M \) and a disk of unbounded size. The disk is divided into blocks of size \( B \). Data is transferred between memory and disk by transferring complete blocks: transferring one block is called an “I/O”. Algorithms can only operate on data that is currently in main memory; to access the data in any block that is not in main memory, it first has to be copied from disk. The I/O-efficiency of an algorithm can be assessed by analysing the number of I/Os it needs as a
function of the input size $n$, the memory size $M$, and the block size $B$. The fundamental building blocks and bounds in the I/O-model are sorting and scanning: scanning $n$ consecutive records from disk takes $\text{scan}(n) = \Theta(n/B)$ I/Os; sorting takes $\text{sort}(n) = \Theta(\sqrt{n} \log M/B + n)$ I/Os in the worst case [1]. It is sometimes assumed that $M = \Omega(B^2)$.

We distinguish cache-aware algorithms and cache-oblivious I/O-efficient algorithms. Cache-aware algorithms may use knowledge of $M$ and $B$, while cache-oblivious algorithms, as defined by Frigo et al. [9], do not know $M$ and $B$. As a result, any bounds that can be proven on the I/O-efficiency of a cache-oblivious algorithm hold for any values of $M$ and $B$ simultaneously. Thus they do not only apply to the transfer of data between disk and main memory, but also to the transfer of data between main memory and the various levels of smaller caches.

**Problem definition.**

A terrain $T$ is a surface in three dimensions, such that any vertical line intersects $T$ in at most one point. The domain $D$ of $T$ is the projection of $T$ on a horizontal plane. The elevation angle of any point $q = (x_q, y_q, z_q)$ with respect to a viewpoint $v = (x_v, y_v, z_v)$ is defined as:

$$\text{ElevAngle}(q) = \arctan \frac{q_v - v_v}{\text{Dist}(q)}$$

where $\text{Dist}(q) := [(x_q, y_q) - (x_v, y_v)]$. Observe that a point $u = (x_u, y_u, z_u)$ is visible from $v$ if and only if the elevation angle of $u$ is higher than the elevation angle of any point of $D$ whose projection on the plane lies on the line segment from $(u_x, u_y)$ to $(v_x, v_y)$. We define the elevation angle of any point $(x_q, y_q)$ of $D$ as the elevation angle of the point $q = (x_q, y_q, z_q)$ where the vertical line through $(x_q, y_q)$ intersects $T$.

In this paper we consider terrains that are represented by a set of $n$ points whose projections on $D$ form a regular rectangular grid with inter-point distance 1. To decide whether a point $u$ is visible from a point $v$, we need to interpolate the elevation angle of all points of $D$ along the line segment from $(u_x, u_y)$ to $(v_x, v_y)$. To this end, we assume that each grid point $q = (x_q, y_q, z_q)$ represents a square “cell” $D(q)$ on $D$ of size $1 \times 1$, centered on $(x_q, y_q)$. For any given viewpoint $v$, we treat the terrain above $D(q)$ as if each point of $D(q)$ has elevation angle $\text{ElevAngle}(q)$. (This is the interpolation method used by Van Kreveld [12] and in our previous work [10]; other interpolation methods are possible too, which we will discuss in Section 6.) What we want to compute is the following: given any terrain $T$ and any viewpoint $v = (x_v, y_v, z_v)$, compute which grid points of the terrain are visible to $v$ and which are not. This comes down to deciding, for any grid point $u$, whether there is any other grid point $q$ such that the square cell $D(q)$ intersects the line segment from $(u_x, u_y)$ to $(v_x, v_y)$ and $\text{ElevAngle}(q) \geq \text{ElevAngle}(u)$. We assume the terrain is given as a matrix $Z$, stored row by row, where $Z_{ij}$ is the elevation of the point in row $i$ and column $j$. The output should be a visibility map: a matrix $V$, stored row by row, in which $V_{ij}$ is 1 if the point in row $i$ and column $j$ is visible, and 0 otherwise. For ease of presentation we assume that the grid is square and has size $\sqrt{n}$ throughout the paper; of course the actual implementations of our algorithms can handle rectangular grids as well.

**Related work.**

An algorithm for computing the visibility map as described above was given by Van Kreveld [12]. Van Kreveld’s algorithm runs in $O(n \log n)$ time, but it is not I/O-efficient. In our previous work we used a technique called distribution sweeping to turn Van Kreveld’s algorithm into a cache-oblivious I/O-efficient algorithm, which runs in $O(n \log n)$ time and uses $O(\text{sort}(n))$ I/Os [10]. With a cache-aware version of this algorithm we computed a visibility map of a terrain of 1.07 billion points (3.97 GiB, using 4 bytes per elevation value) in 234 minutes (that is 13 µs/point) using at most 0.5 GiB of memory and a 7200 RPM hard drive; a visibility map of a terrain of 0.28 billion points (1.04 GiB) was computed in 45 minutes (9.6 µs/point).

Recently Magalhães et al. [11] published an I/O-efficient version of an algorithm by Franklin [7] which also runs in $O(n \log n)$ time and $O(\text{sort}(n))$ I/Os. However, the maps computed by this algorithm are not the same as Van Kreveld’s. In Van Kreveld’s model, the input points $q$ (and only those input points) whose cells $D(q)$ intersect the line segment from $(v_x, v_y)$ to $(u_x, u_y)$ are potential obstacles for the visibility of $u = (u_x, u_y, u_z)$ from $v = (v_x, v_y, v_z)$. In the model of Magalhães et al., the visibility of the terrain at grid point $(u_x, u_y)$ is estimated by considering obstacles along line segments from $(v_x, v_y)$ to $(u_x, u_y)$, effective, one of more points of $D(u)$ (not necessarily to $(u_x, u_y)$ itself), and not all grid points $q$ whose cells $D(q)$ intersect those line segments are considered as obstacles (more details about their approach are given in Section 6). Magalhães et al. compute a visibility map of a 0.90 billion points’ region (1.68 GiB, using 2 bytes per elevation value) within a larger file (5.67 GiB) in 28 minutes, using at most 1.0 GiB of memory and a 7200 RPM hard drive. This amounts to 1.9 µs per point in the visibility map.

The algorithms mentioned above all work on grid terrain models. For an overview of internal-memory algorithms for visibility computations on the second most common format of terrain elevation models, the triangular irregular network or TIN, we refer to Cole and Sharir, and De Floriani and Magilli [3, 5, 6]. Visibility algorithms on TINs use the concept of a horizon or silhouette $\sigma$ of the terrain, which is the upper rim of the terrain, as it appears to a viewer at $v$. More formally, $\sigma_T$ is a function from azimuth angles (compass direction) to elevation angles, such that $\sigma_T(\alpha)$ is the maximum.
elevation angle of any point on the intersection of $T$ with the ray that extends from $v$ in direction $\alpha$. On a triangulated terrain, the horizon is equivalent to the upper envelope of the triangle edges of $T$, projected on an infinite vertical cylinder centered on the viewpoint; it has complexity $O(n \cdot \alpha(n))$, where $\alpha$ is the inverse Ackermann function [3]. Horizons have been used to solve various visibility-related problems on triangulated polyhedral terrains. For example, the visibility of all the vertices in a TIN can be computed in $O(n \alpha(n) \log n)$ time [3]. A central idea in these solutions is that horizons can be merged in time that is linear in their size, and thus allow for efficient divide-and-conquer algorithms.

**New results.**

In this paper we give three new I/O-efficient algorithms to compute the viewshed on a grid terrain and we describe an experimental evaluation that demonstrates their merits in practice.

The first two algorithms sweep the terrain radially by rotating a ray around the viewpoint while maintaining the terrain profile along the ray, similar to Van Kreveld’s algorithm. The difference between the two new algorithms is in the preprocessing before the sweep: the first algorithm, which we describe in Section 2, sorts the grid points in concentric bands around the viewpoint; the second algorithm, which we describe in Section 3, sorts the grid points into sectors around the viewpoint. Both algorithms run in $O(n \log n)$ time and $O(\text{sort}(n))$ I/Os.

The third algorithm, which we describe in Section 4, uses a complementary approach and sweeps the terrain centrifugally. The idea is to grow a star-shaped region around the viewpoint, while maintaining the horizon of the terrain within the swept region. To maintain the horizon efficiently, we represent it by a grid model itself: we maintain the maximum elevation angle (the “height”) of the horizon for a discrete set of regularly spaced azimuth angle intervals. The horizontal resolution of the horizon model is chosen to be similar to the horizontal resolution of the original terrain model, so that we maintain elevation angles for $O(\sqrt{n})$ azimuth angle intervals. This allows a significant speed-up as compared to algorithms that process events at $O(\sqrt{n})$ different azimuth angles, or work with horizons of linear complexity. The centrifugal sweep algorithm runs in $O(n)$ time and $O(\text{scan}(n))$ I/Os cache-obliviously.

We describe the details of the implementations of the three new algorithms and the results of the experimental evaluation in Section 5. In practice, all algorithm are scalable to volumes of data that are more than 10 times larger than the main memory. The centrifugal sweep algorithm is the fastest of our new algorithms. It computes the visibility map of a terrain of 1.07 billion points (3.97 GiB, using 4 bytes per elevation value) in only 11 minutes, using at most 0.5 GiB of memory and a laptop-speed (5400 rpm) hard drive. This is 0.63 μs/point. Thus it is almost 50 times faster than the algorithm from our previous work, which needs 539 minutes (30 μs/point) in the same experimental set-up. Compared to the results of Magelhães et al. our algorithm is four times faster, even though we use a significantly slower disk.

In Section 6 we comment on our results and discuss how our algorithms could be adapted to alternative models of how to interpret grid points as obstacles to visibility.

### 2. A LAYERED RADIAL SWEEP

This section describes our first approach to computing a visibility map. It is loosely based on Van Kreveld’s radial sweep algorithm, which we present below. After that, we explain how to get good I/O-efficiency with a refined radial sweep algorithm. As we will see in Section 5, the refined algorithm performs much better than the algorithm from our previous work [10].

#### 2.1 Van Kreveld’s radial sweep algorithm

The basic idea of Van Kreveld’s algorithm [12] is to rotate a half-line (ray) around the viewpoint $v$ and compute the visibility of each grid point in the terrain when the sweep line passes over it (see Fig. 2). For this we maintain a data structure (the active structure) that, at any time in the process, stores the elevation angles for the cells currently intersected by the sweep line (the active cells). Three types of events happen during the sweep:

- **enter events:** When a cell starts being intersected by the sweep line, it is inserted in the active structure;
- **center events:** When a cell begins being intersected by the sweep line, it is deleted from the active structure;
- **exit events:** When a cell stops being intersected by the sweep line, it is deleted from the active structure;

Thus, each cell in the grid has three associated events. Van Kreveld [12] uses a balanced binary search tree for the active structure, in which the active cells are stored in order of increasing distance from the viewpoint. Because the cells are convex, this is always the same as ordering the active cells in order of increasing distance from the viewpoint to the grid points corresponding to the cells. With each cell we store its elevation angle. In addition, each node in the tree is augmented with the highest elevation angle in the subtree rooted at that node. A query if a point $q$ is visible is answered by checking if the active structure contains any cell that lies closer to the viewpoint than $q$ and has elevation angle at least ElevAngle($q$); if yes, then $q$ is not visible, otherwise it is. To run the complete algorithm, we first generate and sort the $3n$ events by their azimuth angles (the sweep line directions at which they happen). Then we process the events in order of increasing azimuth angle. The whole algorithm runs in $O(n \log n)$ time.

In our previous work we adapted Van Kreveld’s algorithm to make it I/O-efficient [10]. The first step was to generate and sort the events. For each event we stored its location...
in the plane and its elevation angle. Using four bytes per coordinate, this resulted in an event stream of 36m bytes. For large $n$, this is a significant bottleneck.

### 2.2 A new I/O-efficient radial sweep algorithm

The main idea of our new radial sweep algorithm is therefore to avoid generating and sorting a fully specified event stream. The purpose of the event stream was to supply the azimuth angle and the elevation angle of the events in order. Note, however, that the azimuth angle of the events only depends on how the sweep progresses over the grid, but not on the elevation values stored in the input file. Only the elevation angles have to be derived from the input file. Our ideas for making the sweep I/O-efficient are now the following. We can compute the azimuth angles of the events on the fly, without accessing the input file, instead of computing all events in advance. Only when processing an enter event corresponding to a grid point $q$, the elevation of $q$ needs to be retrieved in order to insert $(\text{Dist}(q), \text{ElevAngle}(q))$ into the active structure—for center events the elevation angle can then be found in the active structure and for exit events the elevation angle is not needed. To allow efficient retrieval of elevations for enter events, we pre-sort the elevation grid into lists of elevation values, stored in the order of the enter events that require them. Thus we can retrieve all elevation values in $O(\text{scan}(n))$ I/Os during the sweep. Sorting the complete elevation grid into a single list would be relatively expensive (it would require several sorting passes); we avoid that by dividing the grid into concentric bands around the viewpoint, making one list of elevation values for each band.

**Notation.**

For ease of description, assume that the viewpoint $v$ is in the center of the grid at coordinates $(0, 0, 0)$ and the grid has size $(2m + 1) \times (2m + 1)$, where $m = (\sqrt{n} - 1)/2$. The elevations of the grid points are given in a two-dimensional matrix $Z$ that is ordered row by row, with rows numbered from $-m$ to $m$ from north to south and columns numbered from $-m$ to $m$ from west to east. By $p(i, j)$ we denote the grid point $q = (q_x, q_y, q_z)$ in row $i$ and column $j$ with coordinates $q_x = i$, $q_y = j$, and $q_z = Z_{ij}$; by cell $(i, j)$ we denote the square $D(p(i, j))$. Let $\text{Enter}(i, j)$ denote the azimuth angle of the enter event of cell $(i, j)$.

**Description of the algorithm.**

We now describe our algorithm in detail. Let layer $l$ of the grid denote the set of grid points whose $L_{\infty}$-distance from the viewpoint, measured in the horizontal plane, is $l$. We divide the grid in concentric bands of width $w$ around the viewpoint. Band $k$ (denoted $B_k$), with $k > 0$, contains all grid points of layers $(k - 1)w + 1$ up to $kw$, inclusive; so $p(i, j)$ would be found in band $\lceil \max(|i|, |j|)/w \rceil$ (see Fig. 3).

We choose $w = \Theta(\sqrt{M})$; more precisely, $w$ is the largest power of two such that the elevation and visibility values of a square tile of $(w + 1)\cdot (w + 1)$ points fit in one third of the memory.

Our algorithm proceeds in three phases. The first phase is to generate, for each band $B_k$, a list $E_k$ containing the elevations of all points $p(i, j)$ in the band, ordered by increasing $\text{Enter}(i, j)$ values (recall that $\text{Enter}(i, j)$ denotes the azimuth angle of the enter event of the cell $(i, j)$). Points $p(i, j)$ with the same $\text{Enter}(i, j)$ value are ordered secondarily by increasing distance $\text{Dist}(i, j)$ from the viewpoint. The algorithm that builds the lists $E_k$ is given below. The basic idea is to read the grid points from the elevation grid going in counter-clockwise order around the viewpoint. This is achieved by maintaining a priority queue with points just in front of the sweep line; the priority queue is organised by the azimuth angles of the enter events corresponding to the points to be read. The queue is initialised with all points of $B_0$ that lie straight right of the viewpoint. Then we extract points from the queue one by one in order of increasing $\text{Enter}(i, j)$; when we extract a point, we read its elevation from the elevation grid, write the elevation value to $E_k$, and insert the next point from the same layer in the priority queue (this is the point above, to the left, below, or to the right, depending on which octant the current point is in). In this way, from neighbor to neighbor, all points are eventually reached. Below we describe the algorithm only for the first quadrant (Fig. 3); the others are handled similarly.

**Algorithm BuildBands:**

for $k \leftarrow 1$ to $\lceil m/w \rceil$

do initialise empty list $E_k$ and priority queue $Q$

for $j \leftarrow (k - 1) \cdot w + 1$ to $k \cdot w$

do insert $(\text{Enter}(0, j), -2\pi, 0, j)$ into $Q$

while $E_k$ is not complete

do $(a, i, j) \leftarrow Q.\text{deleteMin}()$

read $Z_{ij}$ from the grid and write it to $E_k$

if $-i < j$

then insert $(\text{Enter}(i - 1, j), i - 1, j)$ into $Q$

else

insert $(\text{Enter}(i, j - 1), i, j - 1)$ into $Q$

clear $Q$

After constructing the lists $E_k$, the second phase of the algorithm starts: computing which points are visible. To do this we perform a radial sweep of all events in azimuth order. Again, we generate the events on the fly with the help of a priority queue, using only the horizontal location of the grid points. We use a priority queue to hold events in front of the sweep line, and an active structure to store the cells that currently intersect the sweep line, sorted by increasing distance from the viewpoint (as in Van Kreveld’s algorithm). The algorithm starts by inserting all enter events of the points straight to the right of the viewpoint into the priority queue. When the next event in the priority queue is an enter event for cell $(i, j)$, the algorithm inserts the corresponding center and exit events in the queue, as well as the enter event of the next cell in the same layer. In addition, it reads the elevation $Z_{ij}$ of $p(i, j)$ from the list of elevation values $E_k$ of the band $B_k$ that contains $p(i, j)$,
and it inserts the cell \((i, j)\) in the active structure with key 
\(\text{Dist}(i, j)\). When the next event in the priority queue is a 
center event for cell \((i, j)\), the algorithm queries the active 
structure for the visibility of the point with key \(\text{Dist}(i, j)\). 
When the next event in the priority queue is an \(\text{EXIT}\) event for 
cell \((i, j)\), the algorithm deletes the element with key 
\(\text{Dist}(i, j)\) from the active structure.

The crux of the above algorithm is the following: when it 
needs to read \(Z_{ij}\), it simply takes the next unread value 
from its band \(E_k\). This is correct, because within each band 
\(B_k\), the above algorithm requires the \(Z_{ij}\) values in the order 
of the corresponding \(\text{ENTER}\) events, and this is exactly the 
order in which these values were put in \(E_k\) by algorithm 
\text{BuildBands}. The output of the second phase is a number 
of lists \(V_k\) with visibility values: one list for each band, in 
order of the azimuth angle of the grid points.

The third phase of the algorithm sorts the lists \(V_k\) into one 
visibility map. To do so we run an algorithm that is more 
or less the reverse of algorithm \text{BuildBands}: we only need 
to swap the roles of reading and writing, and use azimuth 
values for \(\text{CENTER}\) events instead of \(\text{ENTER}\) events.

**Efficiency analysis.**

We will now argue that the above algorithm computes a 
visibility map in \(O(n \log n)\) time and \(O(\text{scan}(n))\) I/Os under 
the assumption that the input grid is square, and we have 
\(M \geq c_1 \sqrt{n}\) and \(M \geq c_2 B^2\) for sufficiently large constants \(c_1\) 
and \(c_2\).

We start with the first phase: \text{BuildBands}. Consider 
the part of band \(B_1\) which lies in the first quadrant. This 
part consists of all points \(p(i, j)\) such that \(0 \leq i \leq w\) 
and \(0 \leq j \leq w\) (except the viewpoint itself). It is a tile of size 
\((w+1)\cdot(w+1)\), which fits in one third of the main memory by 
deinition of \(w\). As the algorithm iterates through the points 
of \(B_1\), it accesses their elevations, loading blocks from disk, 
until eventually the entire tile is in main memory, after which 
there are no subsequent I/O-operations on the input grid.

The number of I/Os to access the tile is 
\(O(w + w^2/B) = O(\sqrt{M} + M/B)\). By the assumption that 
\(M = \Omega(B^2)\), this is \(O(M^2/B) = O(|B_1|/B)\), where \(|B_1|\) 
denotes the number of grid points in \(B_1\). In fact any band \(B_k\) 
with \(k \geq 1\) can be subdivided into \(bk - 4\) tiles of size at most 
\((w+1)\cdot(w+1)\), such that for any band, the sweep line 
will intersect at most two such tiles at any time (see Fig. 3). 
Since a tile fits in at most one third of the memory, two tiles fit in memory 
together. Therefore the algorithm can process each band by 
reading tiles one by one, without ever reading the same tile 
twice. Thus each band \(B_k\) is read in \(O(\text{scan}(|B_k|))\) I/Os, 
and algorithm \text{BuildBands} needs \(O(\text{scan}(n))\) I/Os in total to 
read the input. The output lists \(E_k\) are written sequentially, 
taking \(O(\text{scan}(n))\) I/Os as well. It remains to discuss 
the operation of the priority queue. Note that at any time 
the priority queue stores one cell from each layer, and therefore 
it has size \(m \leq \frac{1}{2} \sqrt{n}\); by assumption this is at most 
\(\frac{1}{2} M/c_1\). Hence, for a sufficiently large value of \(c_1\), 
the priority queue fits in memory together with the two tiles from the input 
file mentioned above (which each take at most one third of 
the memory).

The second phase reads and writes each list \(E_k\) and \(V_k\) in a 
strictly sequential manner. There are \(O(m/w) = O(\sqrt{M}/M)\) 
bands. Under the assumption \(M \geq c_1 \sqrt{n}\) and \(M \geq c_2 B^2\), 
this is only \(O(\sqrt{n}/M) = O(\sqrt{n}/B) = O(M/B)\). This implies 
that one block from each list \(E_k\) or \(V_k\) can reside in 
memory during the sweeping. Thus all lists \(E_k\) and \(V_k\) can 
be read and written in parallel in \(O(\text{scan}(n))\) I/Os in total. 
The priority queue and the active structure have size \(O(\sqrt{n})\) 
and therefore fit in memory by the arguments given above, 
so the second phase needs \(O(n \log n)\) time and \(O(\text{scan}(n))\) 
I/Os in total.

The third phase, sorting the output lists into a visibility 
grid, also takes \(O(n \log n)\) time and \(O(\text{scan}(n))\) I/Os: the 
analysis is the same as for the first phase. Note that in 
practice, the number of visible points is often very small 
compared to the size of the grid. In that case it may be 
better to change the algorithm as follows: instead of writing 
the visibility values of all grid points to separate lists for 
each band and sorting these into a grid, we record only the 
visible grid points with their grid coordinates, write them to 
a single list \(V\), sort this list, and produce a visibility map 
from the sorted output.

**2.3 An algorithm for very large inputs**

The above algorithm computes a visibility map in 
\(\Theta(\text{scan}(n))\) I/Os under the assumption that \(M \geq c_1 \sqrt{n}\), 
and \(M \geq c_2 B^2\). The idea of a layered radial sweep can be 
extended to a recursive algorithm that runs in \(O(\text{sort}(n))\) 
I/Os for any \(n\), without both these assumptions.

The idea is the following: we divide the problem into 
\(\Theta(M/B)\) bands, scan the input to distribute the grid points 
into separate lists for each band, then compute visibility 
recursively in each band, and merge the results. More 
precisely, for each band we will compute a list of “locally” 
visible points and a “local” horizon: these are the points and 
the horizon that would be visible in absence of the terrain 
between the viewpoint and the band. The list of visible points 
is stored in azimuth order around the viewpoint. The horizon 
is a step function of which the complexity is linear in 
the number of points of the terrain; it is also stored as a list 
of points in azimuth order around the viewpoint. Now we 
can merge two adjacent bands as follows. Let \(V_1\) and \(H_1\) be 
the list of visible points and the horizon of the inner band, 
and let \(V_2\) and \(H_2\) be the list of visible points and the horizon 
of the outer band. The merge proceeds as follows. We 
scan these four lists in parallel, in azimuth order, and 
output two lists in azimuth order. First, a list of visible points 
containing all points of \(V_1\), and all points \(V_2\) that are visible 
above \(H_1\). Second, the merged horizon: the upper envelope 
of \(H_1\) and \(H_2\). This correctly computes visibility because a 
point is visible if and only if is visible in its band, and is 
not occluded by any of the bands that are closer to \(v\). The 
idea of the merge step is easily extended to merging \(M/B\) 
lists, resulting in an algorithm that runs in \(O(\text{sort}(n))\) I/Os. We 
leave the details and analysis for the full paper.

**3. A RADIAL SWEEP IN SECTORS**

This section describes our second algorithm for computing 
the visibility map of a point \(v\). It does not achieve better 
asymptotic bounds on running time and I/Os than the 
algorithm from the previous section, but, as we will see in 
Section 5, it is faster. Like the algorithm from Section 2, 
our second algorithm sweeps the terrain radially around the 
viewpoint. As before, the azimuth angles of the events are 
computed on the fly using a priority queue. Elevation values 
of grid points are only needed when their \(\text{ENTER}\) events 
are processed. To make access to elevation values efficient, 
we first divide the elevation grid into sectors of \(\Theta(M)\) grid
points each—this is the main difference with the algorithm from the previous section, which divided the elevation grid into concentric bands.

The algorithm proceeds in three phases. First, for any pair of azimuth angles \( a, b \), let \( S(a, b) \) be the set of grid points whose corresponding \( \text{ENTER} \) events have azimuth angle at least \( a \) and less than \( b \). The first phase of our algorithm starts by computing a set of azimuth angles \( \alpha_0 < \ldots < \alpha_s \), where \( \alpha_0 = 0 \) and \( \alpha_s = 2\pi \), such that for any \( 1 \leq k \leq s \) we have that the coordinates and elevation values of \( S(\alpha_{k-1}, \alpha_k) \) fit in one third of the main memory. Note that this can be done without accessing the elevation grid: the algorithm only needs to know the size of the grid and the location of the viewpoint in order to be able to divide the full grid into memory-size sectors. We then scan the elevation grid and distribute the grid points based on their \( \text{ENTER} \) azimuth angle into lists: one list \( E_k \) for each sector \( S(\alpha_{k-1}, \alpha_k) \). (Cells straight right of the viewpoint need to be entered at the beginning of the sweep and are additionally put in \( E_1 \).)

In the second phase we do the radial sweep as before, sector by sector, with two modifications: (i) whenever we enter a new sector \( S(\alpha_{k-1}, \alpha_k) \), we load the complete list \( E_k \) into memory and sort it by the azimuth angle of the \( \text{ENTER} \) events; (ii) we do not keep a list of visibility values per sector, but instead we write the row and column coordinates of the points that are found to be visible to a single list \( L \).

Finally, in the third phase we sort \( L \) and scan it to produce a visibility map of the full grid. Thus the full algorithm is as follows:

**Algorithm SectoredSweep:**

**First phase—distribution:**

Compute sector boundaries \( \alpha_0, \ldots, \alpha_s \) analytically such that each sector \( S(\alpha_{k-1}, \alpha_k) \) fits in one third of the memory. Scan grid and write each point \( Z_{ij} \) (except \( v \)) to its sector list \( E_k \).

**Second phase—sweep:**

initialise empty active structure \( A \) and priority queue \( Q \)

initialise empty output list \( L \)

for \( j = 1 \) to \( m \)

\( k = 1 \); load \( E_1 \) in memory and sort it by \( \text{ENTER}(i, j) \)

while not all visibility values have been computed

\( (\alpha, \text{type}, i, j) \leftarrow Q.\text{deleteMin}() \)

if \( \text{type} = \text{ENTER} \)

then if \( E_k \) contains no more unread elements

then delete \( E_k \); \( k \leftarrow k + 1 \); load \( E_k \) in memory

sort \( E_k \) and set read pointer at beginning

\( z \leftarrow \text{read next unread value from } E_k \) (= \( Z_{ij} \))

\( \beta \leftarrow \text{arctan}(z/\text{Dist}(i, j)) \) (= ElevAngle(\( p(i, j) \)))

insert \( (\text{Dist}(i, j), \beta) \) into \( A \)

insert \( (\text{Center}(i, j), \text{Center}(i, j), i, j) \) in \( Q \)

insert \( (\text{EXIT}(i, j), \text{EXIT}(i, j), i, j) \) into \( Q \)

if \( |i| < j \text{ or } i = j > 0 \) (next cell is north)

then insert \( (\text{ENTER}(i + 1, j), \text{ENTER}(i + 1, j), i, j) \) in \( Q \)

[... similar for west, south, and east ...]

else if \( \text{type} = \text{CENTER} \)

then query \( A \) if element with key \( \text{Dist}(i, j) \) is visible;

if yes, write \( (i, j) \) to \( L \)

else \( (\text{type} = \text{EXIT}) \)

delelete element with key \( \text{Dist}(i, j) \) from \( A \)

**Third phase—produce visibility map:** Sort \( L \) by \( (i, j) \) and fill in invisible points.

**Efficiency analysis.**

We will now briefly argue that the above algorithm computes a visibility map of the first quadrant in \( O(n \log n) \) time and \( O(\text{scan}(n) + \text{sort}(t)) \) I/Os, where \( t \) is the number of visible grid points, under the assumption that the input grid is square and \( M^2/B \geq cn \) for a sufficiently large constant \( c \).

The first phase of the algorithm reads the elevation grid once and writes elevation values to \( O(n/M) = O(M/B) \) sector lists. Therefore we can keep, for each sector, one block of size \( \Theta(B) \) in memory and thus the first phase produces the sector lists in \( O(\text{scan}(n)) \) I/Os. The running time of the first phase is \( \Theta(n) \). During the second phase, we read the sector lists one by one, in \( O(\text{scan}(n)) \) I/Os in total. The priority queue and the active structure can be maintained in memory by the arguments given in the previous section. Creating and sorting \( L \) takes \( O(\text{sort}(t)) \) I/Os, after which it is scanned to produce a visibility map. Thus the algorithm runs in \( O(n \log n) \) time and \( O(\text{scan}(n) + \text{sort}(t)) \) I/Os.

An algorithm for very large inputs.

When the assumption \( M^2/B \geq cn \) does not hold, a radial sweep based on distribution into sectors is still possible: one can use the recursive distribution sweep algorithm from our previous work [10] and apply the ideas described above to reduce the size of the event stream. The result is an algorithm that runs in \( O(n \log n) \) time and \( O(\text{sort}(n)) \) I/Os. We refer to our previous work for details on the recursive distribution sweeping approach [10].

4. A CENTRIFUGAL SWEET ALGORITHM

In this section we describe our third algorithm for computing the visibility map. It uses a complementary approach to the radial sweep in the previous sections and sweeps the terrain centrifugally, by growing a region \( R \) around the viewpoint. This region is kept star-shaped around \( v \): for any point \( u \) inside \( R \), the line segment from \( (u_x, u_y) \) to \( (v_x, v_y) \) lies entirely inside \( R \). The idea is to grow \( R \) point by point until it covers the complete grid, while maintaining the horizon \( \sigma_R \) of \( R \). Recall that the horizon \( \sigma_R \) is a function from azimuth angles to elevation angles, such that \( \sigma_R(\alpha) \) is the maximum elevation angle of any point on the intersection of \( R \) with the ray that extends from \( v \) in direction \( \alpha \).

Whenever a new point \( u \) is added to \( R \), we decide whether it is visible. The star shape of \( R \) guarantees that all points along the line of sight from \( v \) to \( u \) have already been added, so we can in fact decide whether \( u \) is visible by determining whether \( u \) is visible above the horizon of \( R \) just before adding it (see Fig. 5). The key to a good performance is to have
a way of growing $R$ that results in an efficient disk access pattern, and to have an efficient way of maintaining the horizon structure.

To maintain the horizon efficiently, we represent it by a grid model itself: more precisely, it is maintained in an array $S$ of $32m$ slots, where slot $i$ stores the highest elevation angle in $R$ that occurs within the azimuth angle range from $i \cdot 2\pi/32m$ to $(i+1) \cdot 2\pi/32m$.

For growing the region $R$ the idea is to do so cache-obliviously using a recursive algorithm. Initially we call this algorithm with the smallest square that contains the full grid and whose width is a power of two. When called on a square of size larger than one, it makes recursive calls on each of the four quadrants of the square, in order of increasing distance of the quadrants from $v$. For a square tile with upper left corner $(i, j)$ and width $s$, this distance $\text{TileDist}(i, j, s)$ is the distance from $v$ to the closest point of the tile.

When called on a square of size 1, that is, a square that contains only a single grid point $p(i, j)$, we proceed as follows. We retrieve the elevation $Z_{ij}$ of $p(i, j)$ from the input file and compute its azimuth angle and its elevation angle. Then we check if $p(i, j)$ is visible: this is the case if and only if $p(i, j)$ appears higher above the horizon than the current horizon in the direction of $p(i, j)$; that is, if and only if $\text{ElevAngle}(p(i, j)) > S[(\text{AzimAngle}(p(i, j))/2\pi \cdot 32m)]$. The visibility of $p(i, j)$ is recorded in the output grid $V$. Next we update the horizon to reflect the inclusion of $p(i, j)$ in $R$. To this end we check all slots in the horizon array whose azimuth angle range intersects the azimuth angle range of cell $(i, j)$; let $\mathcal{A}(p(i, j))$ denote this set of slots. For each slot of $\mathcal{A}(p(i, j))$ that currently stores an elevation angle lower than $\text{ElevAngle}(p(i, j))$, we raise the elevation angle to $\text{ElevAngle}(p(i, j))$. We thus have the following algorithm:

**Algorithm** \textsc{CentrifugalSweep}:

create horizon array $S[0..32m-1]$

for $k \leftarrow 0$ to $32m-1$ do $S[k] \leftarrow -\infty$

$s \leftarrow$ smallest power of two $\geq 2m+1$

\textsc{SweepRecursively}($-m, -m, s$)

**Algorithm** \textsc{SweepRecursively}(i, j, s):

(Recursively computes visibility for the tile with upper left cell $(i, j)$ and width $s$)

if $s = 1$

then $\alpha \leftarrow \text{AzimAngle}(p(i, j))$

$\beta \leftarrow \arctan(Z_{ij}/\text{Dist}(i, j))$ (= $\text{ElevAngle}(p(i, j))$)

if $\beta > S[(\alpha/2\pi \cdot 32m)]$ then $V_{ij} \leftarrow 1$ else $V_{ij} \leftarrow 0$

$\alpha^- \leftarrow$ smallest azimuth of any corner of cell $(i, j)$

$\alpha^+ \leftarrow$ largest azimuth of any corner of cell $(i, j)$

for $k \leftarrow \lfloor \alpha^-/2\pi \cdot 32m \rfloor$ to $\lfloor \alpha^+/2\pi \cdot 32m \rfloor - 1$

do $S[k] \leftarrow \max(S[k], \beta)$

else let $Q$ be the four subquadrants:

$s \leftarrow s/2$

$Q \leftarrow \{(i, j, s), (i+s, j, s), (i, j+s, s), (i+s, j+s, s)\}$

sort the elements $(i, j, s)$ of $Q$ by incr. $\text{TileDist}(i, j, s)$

for $(i, j, s) \in Q$

do \textsc{SweepRecursively}(i, j, s)

**Accuracy of the centrifugal sweep.**

Note that when the algorithm updates the horizon array, the elevation angle of $p(i, j)$ may be used to raise the elevation angles of a set of horizon array slots $\mathcal{A}(p(i, j))$, of which the total azimuth range may be slightly larger than that of the cell corresponding to $p(i, j)$—this is due to the rounding of the azimuth angles $\alpha^-$ and $\alpha^+$ in the algorithm. However, this is not a problem: The azimuth angles of grid points that lie next to each other (as seen from the viewpoint) differ by at least roughly $1/m$. The size of the horizon array is chosen such that its horizontal resolution is more than four times bigger: it divides the full range of azimuth angles from 0 to $2\pi$ over $32m$ slots, each of which covers an azimuth angle range of $2\pi/32m < 1/4m$. Therefore, if the resolution of the horizon array would be insufficient, then surely the resolution of the original elevation grid would not be sufficient.

**Efficiency of the centrifugal sweep.**

The number of recursive calls made by the region-growing algorithm is $O(n)$. The only part of any recursive call that takes more than constant time is the updating of the horizon. We analyse this layer by layer, where this time layer $l$ is defined as the cells $(i, j)$ such that $|i| + |j| = l$. There are $O(\sqrt{n})$ layers, and on each layer, each of the $O(\sqrt{n})$ slots of the horizon array is updated at most twice. Thus the total time for updating the horizon is $O(n)$, and the complete algorithm runs in $O(n)$ time.

The number of I/Os under the tall-cache assumption $(M = \Omega(B^2))$ can be analysed as follows. Let $w$ be the largest power of two such that the elevation and visibility values of a square tile of $w \times w$ points fit in half of the main memory. There are $O(n/w^2) = O(n/M)$ recursive calls on tiles of this size, and for each of them the relevant blocks of the input and output files can be loaded in $O(w(w/B + 1)) = O(w^2/B + w)$ I/Os. Thus all I/O for reading and writing blocks of the input and output files can be done in $O(n/M \cdot M/B) = O(\text{scan}(n))$ I/Os in total.

It remains to discuss the I/Os that are caused by swapping parts of the horizon array in and out of memory. To this end we distinguish (i) recursive calls on tiles of size $w \times w$ at distance at least $c \cdot \sqrt{n}/M$ from the viewpoint (for a suitable constant $c$), and (ii) calls on the remaining tiles around the viewpoint. For case (i), observe that each tile $G$ of size $w \times w$ at distance at least $c \cdot \sqrt{n}/M$ from the viewpoint has an azimuth range of $O(w/\sqrt{n}/M) = O(M/\sqrt{n})$; since the horizon array has $O(\sqrt{n})$ slots, $G$ spans $O(M/\sqrt{n} \cdot \sqrt{n}) = O(M)$ slots of the horizon array. Therefore, when $c$ is sufficiently
large, the part of the horizon array that is relevant to the call on \( G \) can be read into the remaining half of the main memory at once, using \( O(\text{scan}(M)) \) I/Os. In total we get \( O(n/M) \cdot O(\text{scan}(M)) = O(\text{scan}(n)) \) I/Os for reading and writing the horizon array in instances of case (i). For case (ii), note that we access the horizon array \( O(n) \) times in total (as shown in our running time analysis above). Because the tiles of case (ii) contain only \( O(n/M) \) grid points in total, the accesses to the horizon array are organised in \( O(n/M) \) runs of consecutive horizon array slots. The total number of I/O-operations induced by these accesses is therefore \( O(n/B + n/M) = O(\text{scan}(n)) \).

Adding it all up, we find that the centrifugal sweep algorithms runs in \( O(n) \) time and \( O(\text{scan}(n)) \) I/Os. The algorithm does not use or control \( M \) and \( B \) in any way: it is cache-oblivious. The I/O-efficiency analysis for the maintenance of the horizon array is purely theoretical as far as disk I/O is concerned: the complete horizon array easily fits in main memory for files up to several trillion grid points. However, the I/O-efficiency analysis also applies to the transfer of data between main memory and smaller caches.

5. EXPERIMENTAL RESULTS

In this section we describe the implementation details and the results of the experiments with our algorithms. We denote by \( \text{io-radial2} \) the layered radial sweep algorithm described in Section 2; \( \text{io-radial3} \) is the radial sweep algorithm from Section 3; and \( \text{io-centrifugal} \) is the centrifugal sweep algorithm from Section 4; \( \text{io-radial1} \) is the algorithm from our previous work [10].

\( \text{io-radial2} \) performs 2 passes over the elevation grid. It first maps the elevation grid file in (virtual) memory and creates the sorted layers \( E_k \). During this phase, the elevation grid and the sorted arrays \( E_k \) are kept in memory, and \( \text{io-radial2} \) relies on the virtual memory manager (VMM) to page in blocks from the elevation grid as necessary when accessing the points in band \( k \). Note that the accesses to the elevation grid are not sequential (although they amount to \( O(\text{scan}(n)) \) I/Os). To help the VMM we implemented the following strategy: whenever the current band needs to access an elevation from the grid, we load an entire square tile of \( \Theta(M) \) points in memory, and keep track of the two most recent tiles. Once all \( E_k \) are computed the elevation grid is freed. During the second phase (the sweep) the elevations are accessed sequentially from the bands \( E_k \) and the output grid is kept in (virtual) memory as a bitmap grid.

\( \text{io-radial3} \) also performs 2 (sequential) passes over the elevation grid. The first pass scans over the elevation grid and places each point \((i,j,Z_{ij})\) in its sector. Sectors are stored as streams on disk. The second pass sorts and sweep the points in one sector at a time by \texttt{ENTER}(i,j). The output grid is kept in (virtual) memory as a bitmap grid. Except for the output grid, \( \text{io-radial3} \) does not use the VMM.

\( \text{io-radial1}, \text{io-radial2} \) and \( \text{io-radial3} \) use the same data structures: a heap as a priority queue; a red-black tree for the active structure [4]; and the same in-memory sorting (optimized quicksort).

\( \text{io-centrifugal} \) is implemented in one (non-sequential) pass over the elevation grid. The implementation is recursive, as described in Section 4. Theoretically the algorithm could run completely cache-obliviously with help of the VMM, but this turned out to be slow. Therefore we implemented a cache-aware version: whenever the recursion enters a tile \( G \) of the largest size that fits in memory, we load the elevation values for the entire tile into memory; when the algorithm returns from the recursive call on \( G \), the visibility values for \( G \) are written to disk.

The implementations of all of our algorithms avoid taking square roots and arctangents, and do not store any angles. Instead of elevation angles, they use the signed squared tangents of elevation angles, and instead of azimuth angles, they use tangents of azimuth angles relative to the nearest axis direction (north, east, south, or west).

Platform.

The algorithms are implemented in C and compiled with gcc/g++ 4.1.2 with optimization level -O3. All experiments were run on HP 220 blade servers, with an Intel 2.83 GHz processor and a 5400 rpm SATA hard drive (the HP blade servers come only with this HD option) The hard disk, which is standard speed for laptop hard-drives, is considerably slower than in our previous experiments [10]. We ran experiments rebooting the machine with 512 MiB and 1 GiB of RAM.

Datasets.

The algorithms were tested on real terrains ranging up to over 7.6 billion elements, see Table 1 for some examples. The largest datasets are SRTM1 data, 30m resolution, available at \texttt{ftp://e0srp01u.ecs.nasa.gov/srtm/}. For a visual comparison, the SRTM datasets are depicted in Figure 6. On all datasets smaller than 4 GiB (Washington), viewshed timings were obtained by selecting several viewpoints uniformly on each terrain and taking the average time.

Results.

Figure 7 shows the total running times with 512 MiB RAM. Sample running times of our algorithms are given in Table 1. First, we note that all our algorithms, being based on I/O-efficient approaches, are scalable to data that is more than ten times larger than the memory of the machine. This is in contrast with the performance of an internal-memory algorithm, which would start thrashing and could not handle terrains moderately larger than memory, as showed in [10]. \( \text{io-radial1}, \text{io-radial2} \) and \( \text{io-radial3} \) are all based on radial sweeps of the terrain, and theoretically they all use \( O(\text{sort}(n)) \) I/Os. In practice, however, both our new algorithms are significantly faster than \( \text{io-radial1} \). On Washington (3.97 GiB), \( \text{io-radial1} \) runs in 32,364 seconds with 16\% CPU², while \( \text{io-radial2} \) runs in 13,780 seconds (22\%)

²The numbers for \( \text{io-radial1} \) are different than the ones
CPU), and io-radial3 in 3.009 seconds (89% CPU). This is a speed-up factor of more than 10.

Both io-radial2 and io-radial3 perform two passes over the grids, however io-radial3 is much faster. On a machine with 512 MiB RAM, on SRTM-Region 3 (8.11 GiB), io-radial2 takes 37.982 seconds (16% CPU), while io-radial3 runs in 6.644 seconds (81% CPU). Overall, for io-radial2 roughly 20% of the time is CPU time, while for io-radial3 the CPU utilization is 80% or more (refer to Table 1). The difference may be explained by the fact that the first pass of io-radial2 is non-sequential (although it performs $O(n/B)$ I/Os), while both passes of io-radial3 are sequential. Another difference is that io-radial2 uses the VMM more than io-radial3.

Our third algorithm, io-centrifugal, is the fastest of the three. It finishes a 28.4 GiB terrain (SRTM-Region 6) in 12.186 seconds (203 minutes), while io-radial2 takes 26.193 seconds (437 minutes). For io-radial3, 61% of this time is CPU time, while for io-centrifugal only 18% (see Table 1). The reason is that io-centrifugal does a single pass through the elevation grid. For any grid point $u$, the highest elevation angle of the $O(\sqrt{n})$ cells that may be on the line of sight from $v$ to $u$ is retrieved from the horizon array in $O(1)$ time, and the horizon array is maintained in $O(1)$ time per point on average. As a result, io-centrifugal is CPU-light and the bottleneck is loading the blocks of data into memory. io-radial3, on the other hand, is more computationally intensive—the highest elevation angle on the line of sight to $u$ needs to be retrieved from a red-black tree in $O(\log n)$ time, and that tree is maintained in $O(\log n)$ time per point. In addition, io-radial3 needs time to sort events.

One of our findings is that relying purely on VMM, even for a theoretically I/O-efficient data access, is slow. The analysis of the I/O-efficiency of both io-radial2 and io-centrifugal is based on the assumption that the VMM will automatically load tiles of size $\Theta(M)$ into memory in the optimal way, and that in practice the performance will not be very different (the theoretical foundations for this assumption were given by [9]). In practice this did not work out so well: a fully cache-oblivious, VMM-based implementation of io-centrifugal and io-radial2 turned out to be slow. By telling the algorithms explicitly when to load a memory-size block (and not using the VMM), we obtained significant speedups (without sacrificing I/O-efficiency for the levels of caching of which the algorithm remained oblivious, and without sacrificing CPU-efficiency).

For comparison with Magalhaes et al., we also used io-centrifugal to compute visibility maps of 0.90 billion points’ regions (1.68 GiB, using 2 bytes per elevation value) within SRTM1 region 4 (5.55 GiB), using at most 1 GiB of memory. This took 409 seconds (averaged over ten random viewpoints), which amounts to 0.45 µs per point in the visibility map; the CPU utilisation was 44% on average. We note that io-centrifugal it is not affected by the number or the location of the grid points that are actually visible. Magalhaes et al. needed 1663 seconds for a similar computation. Since we ran our experiments on a slower disk than Magalhaes et al., we conclude that our algorithm is at least four times faster.

### 6. DISCUSSION

In this paper we described new I/O-efficient algorithms for computing the visibility map of a point on a grid terrain. Our algorithms are not only efficient in terms of the asymptotic growth behaviour of the number of I/Os, they are also efficient in practice. On the largest terrains, using as little as 512 MiB of memory, our algorithms perform at most two passes through the input data, and one pass through the output grid. We obtained a significant speed-up compared to previous work [10] and we were able to process a terrain of 28.4 GiB in 203 minutes with a laptop-speed hard-drive.

Our algorithms use the grid interpolation model of Van Kreveld [12], as explained in Section 1. In this model, a point $q = (q_x, q_y, q_z)$ represents a square cell on $D$ (the domain in the plane underlying the terrain) of size 1, centered on $q_x, q_y$. This square $D(q)$ represents a potential obstacle for lines of sight from $v$ whose projections on the plane intersect $D(q)$. For example, when $q_x \geq q_y \geq 0$, these are the lines of sight within the azimuth range $\arctan((y - \frac{1}{2})/(x + \frac{1}{2}))$ to $\arctan((y + \frac{1}{2})/(x - \frac{1}{2}))$; similar formulas could be derived for cells in other octants. Given any line of sight from $v$ to a grid point $u$ whose visibility we may want to determine, there are $O(\sqrt{n})$ obstacles that may intersect it. A crucial property of Van Kreveld’s model is that, by assuming the elevation angle of an obstacle to be constant throughout its azimuth range, one can sweep a ray around $v$ and maintain the $O(\sqrt{n})$ active obstacles (those currently intersected by the ray) in a data structure and retrieve the maximum elevation angle in $O(\log n)$ time. Other authors may use different
models, either to make other types of algorithms possible, or because they prefer another interpolation model.

For an example of the first, Franklin et al. [7, 8] and Magalhães et al. [11] limit the rays considered in a radial sweep to those that lead to grid points on the boundary of the terrain. They only consider a cell \( D(q) \) to be a potential obstacle to the visibility of a grid point \( u \), if and only if \( D(q) \) and \( D(u) \) are potential obstacles along a ray to a grid point on the boundary. This may have the effect that a high point \( q \) hides a low point \( u \) from view even if the line segment from \( v \) to \( u \) does not intersect \( D(q) \) (but some ray from \( v \) through \( D(q) \) and \( D(u) \) to a point on the boundary does). Or the other way round: a point \( u \) that is not on the boundary may be considered to be visible even if the line segment from \( v \) to \( u \) intersects the cell \( D(q) \) of a higher point \( q \). We will give examples that exhibit this effect in the full paper.

For an example of alternative interpolation models, a commonly used approach is Bresenham’s line rasterization algorithm [2] to determine which cells are considered to intersect a line of sight. For example, when \( q_x \geq 0 \), the footprint \( D(q) \) of the obstacle represented by \( q \) is not a square, but a line segment of length 1, oriented from north to south. Thus it may be an obstacle to lines of sight within the azimuth range \( \arctan\left(\frac{y - \frac{1}{2}}{x}\right) \) to \( \arctan\left(\frac{y + \frac{1}{2}}{x}\right) \). Thus the obstacles, as seen from the viewpoint, appear to be narrower than in Van Kreveld’s model (see Fig. 8), and more grid points may be considered to be visible as a result. All of our algorithms are easily adapted to work with this model. They may even be faster in that case, because the active structures of the radial sweep algorithms will be smaller and the centrifugal sweep algorithm updates fewer slots of the horizon.

Finally, it is also possible to consider cells as obstacles to visibility whose elevation angle is not constant throughout their azimuth range, but rather interpolated on the basis of the elevation values of neighbouring grid points. The centrifugal sweep algorithm can easily be adapted to accommodate this—of course within the limits set by the resolution of the horizon array. The radial sweep algorithms would be considerably more difficult to adapt: one would need a kinetic active structure (an active structure in which one can store elevation angles that change over time). We leave experiments with such adaptations as topics for further research.

### 7. REFERENCES


