# On IO-efficient viewshed algorithms and their accuracy

Herman Haverkort
Eindhoven U. of Technology
Netherlands

Laura Toma*
Bowdoin College
USA

Bob PoFang Wei
Bowdoin College
USA

## ABSTRACT

Given a terrain T and a point $v$, the viewshed or visibility map of $v$ is the set of points in $T$ that are visible from $v$. To decide whether a point $p$ is visible one needs to interpolate the elevation of the terrain along the line-of-sight (LOS) $vp$. Existing viewshed algorithms differ widely in which and how many points they chose to interpolate, how many lines-of-sight they consider, and how they interpolate the terrain. These choices crucially affect the running time and accuracy of the algorithms. In this paper our goal was to obtain an IO-efficient algorithm that computes the viewshed on a grid terrain with as much accuracy as possible given the resolution of the data. We describe two algorithms which are based on computing and merging horizons, and we prove that the complexity of horizons on a grid of $n$ points is $O(n)$, improving on the general $O(n\alpha(n))$ bound on triangulated terrains. Our finding is that, in practice, horizons on grids are significantly smaller than their theoretical worst case bound, which makes horizon-based approaches very fast.

To measure the differences between viewsheds computed with various algorithms we implement an error metric that averages viewshed differences over a large number of viewsheds computed from a set of viewpoints with topological significance, like valleys and ridges. Using this metric we compare our current approach, Van Kreveld's model used in our previous work [7], the algorithm of Ferreira et al. [6], and the viewshed module `r.los` in the open source GIS GRASS.

## 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*

*LT and BW supported by NSF award no. 0728780.



## General Terms

Algorithms, Design, Experimentation, Performance

## 1. INTRODUCTION

The computation of visibility is a fundamental problem on terrains and is at the core of many applications in Geographic Information Systems (GIS). The basic problem is point to point visibility: Two points $a$ and $b$ on a terrain are visible to each other if the interior of their *line-of-sight* $\overline{ab}$ (the line segment between $a$ and $b$) lies entirely above the terrain. Based on this one can define the viewshed: Given a terrain and an arbitrary (view)point $v$, not necessarily on the terrain, the *visibility map* or *viewshed* of $v$ is the set of all points in the terrain that are visible from $v$.

The key in defining and computing visibility is chosing a terrain model and an interpolation method. The most common terrain models are the grid and the TIN (triangular irregular network). A grid terrain is essentially a matrix of elevation values, representing elevations sampled from the terrain with a uniform grid; the $x, y$ coordinates of the samples are not stored in a grid terrain, they are considered implicit w.r.t. to the corner of the grid. A TIN terrain consists of an irregular sample of points ($x, y$ and elevation values), and a triangulation of these points is provided. Grid terrains are the most widely used in GIS because of their simplicity.

To decide whether a point $p$ is visible on a given terrain model, one needs to interpolate the elevation along the line-of-sight $pv$ between the viewpoint $v$ and $p$ (more precisely, along the projection of the line-of-sight on the horizontal plane) and check whether the interpolated elevations are below the line of sight. Various algorithms differ in what and how many points they select to interpolate along the line-of-sight, and in the interpolation method used. These choices crucially affect the efficiency and accuracy of the algorithms.

In order to be useful in practice, viewshed algorithms need to be fast and scalable to very large terrains. Over the past years vast amounts of terrain data are available at better and better resolution. Since terrain datasets may be as large as tens and even hundreds of gigabytes, they may not fit in the main memory of a computer all at once and most of the data may have to reside on disk during computations. Hence, working with such data requires efficient algorithms that are designed to minimize the I/O: the swapping of data between a fast main memory (or cache) and a larger but slower memory such as a disk. To design and analyze efficient algorithms we use the standard IO model by Aggarwal and Vitter [1]. In this model, a computer has a memory of size $M$ and a disk of unbounded size, divided into blocks of size $B$. Transferring

one block of data beween main memory and disk is called an I/O operation, or simply, an I/O. The I/O-efficiency of an algorithm can be assessed by analyzing the number of I/Os it needs as a function of the input size $n$, the memory size $M$, and the block size $B$. Fundamental building blocks for I/O-efficient algorithms are *scanning* and *sorting*: scanning $n$ consecutive records from disk takes $\mathsf{scan}(n) = \Theta(\frac{n}{B})$ I/Os; sorting takes $\mathsf{sort}(n) = \Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os [1].

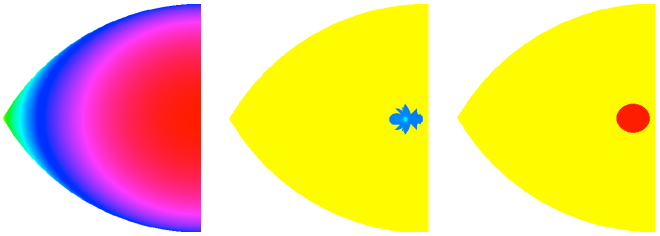
**Related work.** Viewsheds on grids are usually modeled in a discrete way: each point in the grid is marked as visible or invisible, and the viewshed of $v$ is defined as the set of all *grid* points that are visible from $v$. For the rest of the paper we consider a grid of $n$ points, for simplicity assumed to have size $\sqrt{n}$ by $\sqrt{n}$. The standard method for computing viewsheds on grid terrains is the algorithm `R3` by Franklin and Ray [8]. `R3` determines the visibility of each point in the grid as follows: it computes the intersections between the horizontal projection of the line-of-sight and the (horizontal and vertical) grid lines, and computes the elevation of the terrain at these intersection points by linear interpolation. This is considered to be the standard model and `R3` is considered to produce the "exact" viewshed [10]. However, as described by Franklin and Ray, `R3` runs in $O(n\sqrt{n})$ time, which is too slow in practice, especially for multiple viewshed computations.

A variety of viewshed algorithms have been proposed that optimize `R3` while approximating in some way the resulting viewshed: Some algorithms consider only a subset of the $O(n)$ lines-of-sight; others interpolate the line-of-sight only at a subset of the $O(\sqrt{n})$ intersection points with the grid lines; yet others have some other way of determining in $O(1)$ time whether a point in the grid is visible. The optimized viewshed algorithms run in $o(n\sqrt{n})$ time, most often $O(n)$. Examples are the algorithm `io-centrifugal` from our previous work [7], XDraw by Franklin and Ray [8]; Backtrack by Izraelevitz [10]; `R2` by Franklin and Ray [8]; and van Kreveld's radial sweep algorithm [11]—below we describe briefly the last two results which are relevant to this paper.

The algorithm named `R2`, proposed by Franklin and Ray [8], is an optimization of `R3` that runs in $O(n)$ time. The idea of `R2` is to examine the lines-of-sight *only* to the $O(\sqrt{n})$ grid points on the boundary of the grid; a grid point that is not on the boundary is considered to be visible if the nearest point of intersection between a grid line and one of the examined lines-of-sight is determined to be visible. Overall `R2` is fast and, according to its authors, produces a good approximation of `R3` that outweighs its loss in accuracy [8].

Van Kreveld described a different approach for computing viewsheds on grids that could be seen as an optimization of `R3` [11]. In this model the terrain is seen as a tessellation of square cells, where each cell is centered around a grid point and has the same view angle as the grid point throughout the cell; that is, the cell appears as a horizontal line segment to the viewer. This property allows for the viewshed to be computed in a radial sweep of the terrain in $O(n \lg n)$ time [11].

The viewshed algorithms mentioned so far assume that the computation fits in memory and are not IO-efficient. I/O-efficient viewshed algorithms have been proposed most recently by Ferreira et al. [6] and Fishman et al. [7]. The algorithm by Ferreira et al. is based on `R2`, while the algorithm `io-radial3` of Fishman et al. is based on Van Kreveld's model. The code of `io-radial3` (more precisely, of its previous version) was deployed in GRASS, an open source GIS, and the GRASS community pointed out an artifact[1] (Figure 1): When run on an artificially generated terrain in the shape of a semi-ellipsoid, with the viewpoint at the top, the viewshed generated by `io-radial3` looks unintuitive and has significant differences compared to the module that computes visibility in GRASS, `r.los`. We traced this down to a model and interpolation artifact. This simple test shows the importance of considering accuracy issues when evaluating a viewshed algorithm, and when comparing viewshed algorithms that work in different models.



(a) (b) (c)

**Figure 1: Accuracy issues with the algorithm in [7]: (a) A top-down view of an artificially generated terrain. Red represents highest elevations, and green the lowest. (b) The viewshed generated by the algorithm in [7]; the viewpoint, in the center of the blob, is shown in light blue. (c) The viewshed generated by `r.los` in GRASS.**

While efficiency is easy to compare, comparing accuracy is much harder. The straightforward way to assess accuracy is to compare the computed viewshed with ground truth data. Ideally one would consider a large sample of viewpoints, compute the viewshed from each one in turn, compare it with the real viewshed at that point, and aggregate the differences. Unfortunately, ground truth viewsheds are hard, if not impossible, to obtain. To our knowledge, an investigation and comparison of the accuracy of the various existing viewshed algorithms is missing.

**Our contributions.** In this paper our goal has been to obtain an IO-efficient algorithm that computes the viewshed on a grid terrain with as much accuracy as possible given the resolution of the data, and evaluate its running time and accuracy. Similar to `R3`, our approach is to determine the visibility of each point $p$ in the grid by computing the line-of-sight $vp$ and evaluating the elevation of the terrain at specific points along $vp$. We distinguish between two models (Figure 2), which we describe in Section 2: In the *gridlines* model we view the points in the input grid to be connected by horizontal and vertical lines, and visibility is determined by evaluating the intersections of the line-of-sight with the grid lines using linear interpolation; this is the model underlying `R3`, and thus our algorithms give an improved and IO-efficient version of `R3`.

We also consider a slightly different model, the *layers* model, in which we view the points in the input grid to be connected in concentric layers around the viewpoint and visibility is determined by evaluating the intersections of the line-of-sight with these layers using linear interpolation. The layers model considers only a subset of the intersections considered by the gridlines model and therefore the viewshed generated will be larger (more optimistic) than the one

[1] See `http://trac.osgeo.org/grass/ticket/390`

generated with the gridlines model. In Section 4 we show that these differences are practically insignificant. The layers model is faster in practice, while having practically the same accuracy as the gridlines model.

We describe our algorithms, VIS-ITER and VIS-DAC, in Section 2.1 and 2.2. They are based on computing and merging horizons in an iterative or divide-and-conquer approach, respectively. Horizon-based algorithms for visibility problems have been described by de Floriani and Magillo [5]. On a triangulated terrain $T$, the horizon is equivalent to the upper envelope of the triangle edges of $T$ as projected on a view screen, and has complexity $H(n) = O(n \cdot \alpha(n))$, where $n$ is the number of vertices in the TIN and $\alpha()$ is the inverse of the Ackerman function [4]. In Section 2.1 we show that we can prove a better bound for our setting: that is, we prove that the upper envelope of a set $S$ of line segments in the plane such that the widths of the segments do not differ in length by more than a factor $d$ has complexity $O(dn)$. From here we show that the horizon on a grid of $n$ points in our model has complexity $O(n)$ in the worst case.

To assess the efficiency of the algorithms in practice we perform an experimental analysis on datasets up to 28 GB, described in Section 4. Our main finding is that, in practice, horizons are significantly smaller than their theoretical upper bound, which makes horizon-based algorithms unexpectedly fast. Our straightforward, iterative algorithm VIS-ITER is faster than VIS-DAC; and they are both faster than the comparable algorithm `io-radial3` in our previous work [7], while their results are more accurate.

Finally, we describe our metric for estimating accuracy and measuring the differences between various viewshed algorithms in Section 3. The error metric averages viewshed differences over a large number of viewsheds computed from a set of viewpoints with topological significance, like valleys and ridges. Each pair of viewsheds is compared by counting the false visible and false invisible points, in a manner similar to Ben-Moshe et al. [3, 2]. Using this metric we compare our new algorithms using the gridlines and layers models with our previous algorithm based on Van Kreveld's model [11, 7], with the algorithm `R2` used in [6], and with the module `r.los` that computes viewsheds in GRASS. The findings are outlined Section 4.2, and we conclude that accuracy considerations play an important role in assessing viewshed algorithms.

## 2. THE MODEL AND ALGORITHMS

This section describes our model and algorithms. We start with the notation. Let $T$ be a terrain represented as a grid. We assume the grid 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 is the viewshed of $v$, that is, 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 $\sqrt{n}$ by $\sqrt{n}$ (of course the actual implementations of our algorithms can handle rectangular grids as well). The *elevation angle* of any point $p = (p_x, p_y, p_z)$ wrt to a viewpoint $v = (v_x, v_y, v_z)$ is defined as

$$\texttt{ElevAngle}(p) = \texttt{arctan}\frac{p_z - v_z}{\texttt{dist}(p, v)}$$

where $\text{dist}(p, v) = |(p_x, p_y) - (v_x, v_y)|$. A point $u = (u_x, u_y, u_z)$ is visible from $v$ if and only if the elevation angle of $u$ is higher than the elevation angle of any point of $T$ whose projection on the plane lies on the line segment from $(u_x, u_y)$ to $(v_x, v_y)$.

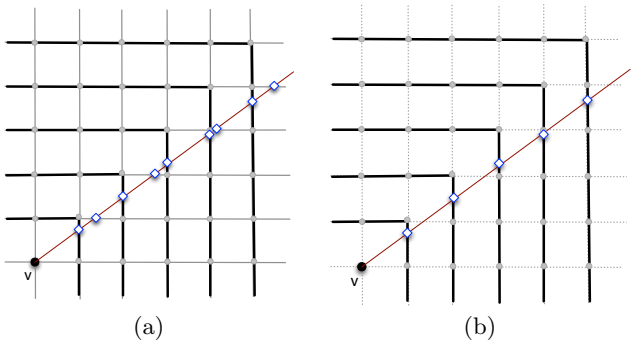



**Figure 2: (a) The gridlines model: visibility is determined by the intersections of the LOS with all the grid lines. (b) The layers model: visibility is determined by the intersections of the LOS with the layers.**

Our algorithms are based on the concept of *horizon*. Put simply, the horizon $H_v$ (wrt to viewpoint $v$) is the upper rim of the terrain as it appears to a viewer at $v$. Suppose we recenter our coordinate system such that $v = (0, 0, 0)$, and consider a *view screen* around the viewer that consists of the Cartesian product of the vertical axis and the square with vertices $(1, 0)$, $(0, 1)$, $(-1, 0)$ and $(0, -1)$. The projection of a point $p = (p_x, p_y, p_z)$ towards $v$ onto the view screen has coordinates: $p/(|p_x| + |p_y|)$. Note that any line segment that does not cross the north-south or east-west axis through $v$, will appear as a line segment in the projection onto the view screen. We now define the horizon of the terrain as it appears in the projection. More precisely, for $t \in [0, 2]$, we define the horizon $H_v(t)$ as the maximum value of $p_z/(|p_x| + |p_y|)$ over all terrain points $p$ such that $p_x/(|p_x| + |p_y|) = 1 - t$ and $p_y \leq 0$ (this defines the horizon of the terrain south of the viewpoint). For $t \in [2, 4]$, we define the horizon $H_v(t)$ as the maximum value of $p_z/(|p_x| + |p_y|)$ over all terrain points $p$ such that $p_x/(|p_x| + |p_y|) = 3 + t$ and $p_y \geq 0$ (this defines the horizon of the terrain north of the viewpoint).

**The model.** We consider two models, shown in Figure 2: In the *gridlines* model the grid points are connected by vertical and horizontal lines in a grid, and visibility is determined by evaluating the intersections of the LOS with the grid lines. The gridlines model is the model used by `R3`. We also consider a slightly different model, the *layers* model, in which the grid points are connected in concentric layers around the viewpoint and visibility is determined by evaluating the intersections of the LOS with the layers. The layers model is a relaxation of the gridlines model because it considers only a subset of the intersections (obstacles) considerd by the gridlines model; any point visible from $v$ in the grid model is also visible in the layers model (but not the other way), and the viewshed generated by the grid model is a subset of the viewshed generated with the layers model.

We start by describing how to compute viewsheds in the layers model in Section 2.1 and 2.2; in Section 2.3 we show how both algorithms can be extended to the gridlines model while maintaining the same worst-case complexity.

## 2.1 An iterative algorithm: VIS-ITER

This section describes our first viewshed algorithm in the layers model, VIS-ITER. The main idea of VIS-ITER is to traverse the grid in layers around the viewpoint, one layer at a time, while maintaining the horizon of the region traversed so far. The horizon is maintained as a sequence of points $(t, H_v(t))$, sorted by $t$-coordinate, between which we interpolate linearly. When traversing a point $p$, the algorithm uses the maintained horizon to determine if $p$ is visible or invisible. In order to do this IO-efficiently, it divides the grid in bands around the viewpoint and processes one band at a time. The output visibility grid is generated band by band, and is sorted into a grid in the final phase of the algorithm. The size of the band is chosen such that a band fits in memory. Below we explain these steps in more detail.

**Notation.** For ease of description, we 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 $q_x = j, q_y = -i$ and $q_z = Z_{ij}$.

For $l \geq 0$, let *layer* $l$ of the grid, denoted $L_l$, denote the set of grid points whose $L_\infty$-distance from the viewpoint, measured in the horizontal plane, is $l$. By definition, $L_0$ consists of only one point, $v$. We divide the grid in concentric bands around the viewpoint: For $k > 0$, band $k$, denoted $B_k$, contains all points in layers $w_{k-1}$ (inclusive) to $w_k$ (exclusive), where $w_k (k \geq 0)$ denote the indices of layers corresponding to the band boundaries. Thus band $B_1$ contains all points in layers $w_0 = 1$ to $w_1$, band $B_2$ contains all points in layers $w_1$ to $w_2$, and so on.

The algorithm starts with a preprocessing step which, given an arbitrary constant $K$, computes the band boundaries $w_k (k \geq 1)$ such that a band has size $\Theta(K)$ as follows: it cycles through each layer $i$ in the grid, computes (analytically) the number of points in that layer, and checks whether including this layer in the current band makes the band go over $K$ points. If yes, then layer $i$ marks the start of the next band. Otherwise, it adds the points in layer $i$ to the current band and continues.

The maximum size $K$ of a band is chosen such that a band fills roughly a constant fraction of memory, and each band is at least one layer wide. More precisely, we choose $K = c_1 M$ and assume $\sqrt{n} \leq c_1 M$, for a sufficiently small constant $c_1$ which will be defined more precisely later. Thus the number of bands, $N_{bands}$, is $O(N/M)$.

Once the band boundaries are set, the 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 in the band. It does this by scanning the grid in row-column order: for each point $p(i, j)$, it calculates the index $k$ of the band that contains the point and writes $Z_{ij}$ to $E_k$. We note that the first phase writes the lists $E_k$ sequentially, and thus list $E_k$ contains the points in the order in which they are encountered during the (row-by-row) scan of the grid. The algorithm is given below.

**Algorithm** BUILDBANDS:

load list containing band boundaries in memory
**for** $k \leftarrow 1$ **to** $N_{bands}$
**do** initialize empty list $E_k$
**for** $i \leftarrow -m$ **to** $m$
**do** **for** $j \leftarrow -m$ **to** $m$
  **do** read next elevation $Z_{ij}$ from grid
    $k \leftarrow$ band containing point $(i, j)$
    append $Z_{ij}$ to $E_k$

Given the lists $E_k$, the second phase of the algorithm computes which points are visible. To do this it traverses the grid one band at a time, reading the list $E_k$ into memory. Once a band is in memory, it traverses it layer by layer from the viewpoint outward, counter-clockwise in each layer. The output of the second phase is a set of lists $V_k$ with visibility values, one list for each band. While traversing the grid in this fashion the algorithm maintains the horizon of the region encompassed so far. More precisely, let $L_{1,i} (i \geq 1)$ denote the set of points in layers $L_1$ through $L_i$. Before traversing the next layer $L_{i+1}$, the algorithm knows the horizon $H_{1,i}$ of $L_{1,i}$. While traversing the points in $L_{i+1}$, the algorithm determines for each point $p$ if it is above or below the horizon $H_{1,i}$ and records this in $V_k$. At the same time it updates $H_{1,i}$ on the fly to obtain $H_{1,i+1}$. To do so, the algorithm computes, for each point $p$, the projection $h$ onto the view screen of the line segment that connects $p$ to the previous point in the same layer, the algorithm computes the intersection of $h$ with the current horizon as represented by $H_{1,i}$, and then updates $H_{1,i}$ to represent the upper envelope of the current horizon and $h$. After traversing the entire grid in this manner, the set of points that have been marked visible during the traversal constitute the viewshed of $v$. The algorithm is given below only for the first octant; the other octants are handled similarly:

**Algorithm** VISBANDS-ITER:

$H_{1,0} \leftarrow \emptyset$
**for** $k \leftarrow 1$ **to** $N_{bands}$
**do** load list $E_k$ in memory
  create list $V_k$ in memory and initialize it as all invisible
  **for** $l \leftarrow w_{k-1}$ to $w_k$ //for each layer in the band
  **do** //traverse layer $l$ in ccw order
    **for** $r \leftarrow 0$ to $-l$ //first octant
    **do** get elevation $Z_{rl}$ of $p(r, l)$ from $E_k$
      determine if $Z_{rl}$ is above $H_{1,l-1}$
      if visible, set value $V_{rl}$ in $V_k$ as visible
      $h \leftarrow$ projection of $p(r - 1, l)p(r, l)$
      merge $h$ into horizon $H_{1,l-1}$
    $H_{1,l} \leftarrow H_{1,l-1}$

The third and final phase of the algorithm creates the visibility grid $V$ from the lists $V_k$. We note that in phase 2 the lists $V_k$ are stored in the same order as $E_k$, that is, the order in which the points in the band are encountered during a row-by-row scan of the grid; keeping points in this order is convenient because it saves an additional sort, and in the same time this is precisely the order in which they are needed by phase 3. Phase 3 is the reverse of phase 1: for each point $(i, j)$ in the grid in row-major order, it computes the band $k$ where it falls, accesses list $V_k$ to retrieve the visibility value of point $(i, j)$, and writes this value to the output grid $V$. The crux in this phase is that it simply reads the lists $V_k$ sequentially. The algorithm is given below:

**Algorithm** CollectBands:

load list containing band boundaries in memory
initialize empty list $V$
**for** $i \leftarrow -m$ **to** $m$
**do for** $j \leftarrow -m$ **to** $m$
**do** $k \leftarrow$ band containing point $(i, j)$
get value $V_{ij}$ of point $(i, j)$ from $V_k$
append $V_{ij}$ to list $V$

### *Efficiency analysis of* vis-iter .

We now analyze each phase in vis-iter under the assumption that $n \leq cM^2$ for a sufficiently small constant $c$.

The pre-processing phase runs in $O(n)$ time and no I/O (does not access the grid). The output of this step is a list of $O(N_{bands}) = O(n/M)$ band boundaries, which fits in memory assuming that $n \leq cM^2$ for a sufficiently small constant $c$.

The first phase, BuildBands, reads the points of the elevation grid in row-column order, which takes $O(n)$ time and $O(\mathsf{scan}(n))$ I/Os. With the list of band boundaries in memory, the band containing a point $(i, j)$ can be computed with, for example, binary search in $O(\lg n/M) = O(\lg n)$ time and no I/O. The lists $E_k$ are written to in sequential order. If one block from each band fits in memory, which happens when $n \leq cM^2/B$ for a sufficiently small constant $c$ (so that $N_{bands} = O(n/M) = O(M/B)$), then writing the lists $E_k$ directly takes $O(\mathsf{scan}(n)) = O(\mathsf{sort}(n))$ I/Os (note that $O(\mathsf{sort}(n))$ and $O(\mathsf{scan}(n))$ are equal if $n = O(M^2/B)$). If we cannot keep one block of each band in memory, that is, $n > cM^2/B$, then we perform a hierarchical distribution as follows: we group the $N_{bands}$ bands in $O(M/B)$ super-bands, keep a write buffer of one block for each super-band in memory, distribute the points in the grid to these super-bands, and recurse on the super-bands to distribute the grid points to individual bands. A pass takes $O(\mathsf{scan}(n))$ I/Os, overall it takes $O(\log_{M/B} N_{bands}) = O(\log_{M/B} N/M)$ passes, and thus the first phase has I/O-complexity $O(\mathsf{sort}(n))$. In total, the first phase takes $O(n \lg n)$ time and $O(\mathsf{sort}(n))$ I/Os.

The second phase, VisBands-ITER, takes as input the lists $E_k$ and computes the visibility bands $V_k$. We choose $K = c_1 M$ such that the elevations $E_k$ and the visibility map $V_k$ of any band $B_k$ of size $K$ fits in 2/3 of the memory; the remaining 1/3 of the memory is saved for the horizon structure. While processing a band $B_k$ in the second phase, the points in $E_k$ and $V_k$ are not accessed sequentially. However, given the band boundaries, the location of any point in a band can be determined analytically, and thus the value (elevation or visibility) of any point in a band can be accessed in constant time, without any search structure, and without any I/O. Let us denote by $H_{tot}$ the total cumulative size of all partial horizons $H_{1,l}$: $H_{tot} = \sum_{l=1}^{\sqrt{n}} |H_{1,l}|$. The horizon $H_{1,l}$ is maintained as a list $\{(t, h)\}$ of horizontal and vertical coordinates on the view screen, sorted counter-clockwise (ccw) around the viewpoint. As the algorithm traverses a layer $l$ in ccw order, it also traverses $H_{1,l-1}$ in ccw order, and constructs $H_{1,l}$ in ccw order. To determine whether a point is above the horizon, it is compared with the last segment in the horizon; if the point is above the horizon, it is added to the horizon. Thus the traversal of a layer $l$ runs in $O(|L_l|+|H_{1,l-1}|+|H_{1,l}|)$ time. Over the entire grid, phase 2 runs in $O(\sum_l(|L_l| + |H_{1,l}|)) = O(n + H_{tot})$ time.

The IO-complexity of the second phase: The algorithm reads $E_k$ into memory, and writes $V_k$ to disk at the end. Over all the bands this takes $O(\mathsf{scan}(n))$ I/Os. If the horizon $H_{1,l}$ is small enough so that it fits in memory (for any $l$), then accessing the horizon does not use any IO. If the horizon does not fit in memory, we need to add the cost of traversing the horizon in ccw order, for every layer, $O(\mathsf{scan}(\sum_{l=1}^{\sqrt{n}} |H_{1,l}|)) = \mathsf{scan}(H_{tot})$ I/Os.

Finally, the third phase, CollectBands, takes as input the lists $V_k$ and the list of band boundaries and writes the visibility map. For $n \leq cM^2$, the list of band boundaries fits in memory. For any point $(i, j)$ the band containing it can be computed in $O(\lg n)$ time and no IO. The bands $V_k$ store the visibility values in the order in which they are encountered in a (row-column) traversal of the grid. Thus, once the index $k$ of the band that contains point $(i, j)$ is computed, the visibility value of this point is simply the next value in $V_k$. As with step 1, we distinguish two cases: if the number of bands is such that one block from each band fit in memory, then this step runs in $O(n)$ time and $O(\mathsf{sort}(n)) = O(\mathsf{scan}(n))$ I/Os. Otherwise, this step first performs a multi-level $M/B$-way merge of the bands into $O(M/B)$ super-bands so that one block from each can reside in main memory; in this case, the complexity of the step is $O(n \lg n)$ time and $O(\mathsf{sort}(n))$ I/Os.

Putting everything together, we have the following:

Theorem 1. *The algorithm* vis-iter *computes viewsheds in the layers model in $O(n \lg n + H_{tot})$ time and $O(\mathsf{sort}(n) + \mathsf{scan}(H_{tot}))$ I/Os, provided that $n \leq cM^2$ for a sufficiently small constant $c$.*

Furthermore, if $n = O(M^2/B)$ and the partial horizons $H_{1,l}$ are small enough to fit in memory for any $l$, the overall IO complexity becomes $O(\mathsf{scan}(n))$ I/Os. We note that when $n = \Omega(M^2)$ the algorithm can be adapted using standard techniques to run in the same bounds from Theorem 1, but we do not detail on this case because it has no relevance in practice.

### *Discussion.*

Phase 1 and 3 of the algorithm are very simple and perform a scanning pass over the grid and the bands, provided that $n \leq cM^2/B$: Phase 1 reads the input elevation grid sequentially and writes the elevation bands sequentially; Phase 3 reads the visibility bands sequentially and writes the visibility grid sequentially. We found this condition to be true in practice on our largest test grid ($28GB$) and with as little as $.5GB$ of RAM. With more realistic value of $M = 8GB$ (and $B = 16KB$), the condition is true for $n$ up to $10^{15}$ points. Thus, handling the sub-case $n \leq cM^2/B$ separately in the algorithm provides a simplification and a speed-up without restricting generality.

Phase 2, which scans partial horizons $H_{1,l}$ for every layer, runs in $O(n + H_{tot})$ time and $O(\mathsf{scan}(n + H_{tot}))$ I/Os. As we will prove below, in the worst case $|H_{1,l}| = \Theta(l^2)$, and the running time of the second phase could be as high as $O(H_{tot}) = O(\sum_{l=1}^{O(\sqrt{n})} \Theta(l^2)) = O(n\sqrt{n})$, with handling the horizon dominating the running time. The worst-case complexity is high but, on the other hand, if $H_{1,l}$ are small, they fit in memory and the algorithm is fast. In particular if $H_{tot}$ is $O(n)$, then phase 2 is linear. This seems to be the case on all terrains and all viewpoints that we tried and may be a feature of realistic terrains. In Section 4 we'll discuss our empirical findings in more detail.

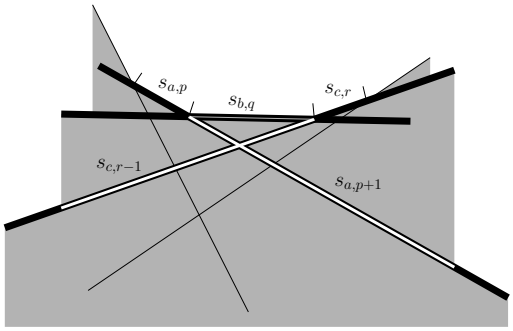



**Figure 3: Illustration of the proof of Lemma 1. The white subsegments take the charge for $s_{b,q}$; together they are at least as wide as $s_b$, since they stick out from under $s_b$ on both sides.**

### *Worst-case complexity of the horizon.*

Since the horizon is the upper envelope of the projections of grid line segments onto the view screen, its complexity is at most $O(n\alpha(n))$, where $n$ is the number of line segments [9, 12]. We will now show that we can prove a better bound for our setting. Let the *width* width($s$) of a line segment $s$ be the length of its projection on a horizontal line. We need the following lemma.

LEMMA 1. *If $S$ is a set of $n$ line segments in the plane, such that the widths of the line segments of $S$ do not differ in length by more than a factor $d$, then the upper envelope of $S$ has complexity $O(dn)$.*

PROOF. Let $s_1, ..., s_n$ be the segments of $S$. Each segment $s_i$ consists of a number of maximal subsegments such that the interior of each subsegment lies either entirely on or entirely below the upper envelope. Let the subsegments of $s_i$ be indexed by $s_{i,j}$, such that the subsegments of $s_i$ from left to right are indexed by consecutive values of $j$, and such that $s_{i,j}$ is part of the upper envelope if and only if $j$ is odd. Let $u_1, ..., u_m$ be the line segments of the upper envelope.

We consider two categories of line segments on the upper envelope: (i) segments that have at least one endpoint that is an endpoint of a segment of $S$; (ii) segments whose endpoints are no endpoints of segments in $S$.

Clearly, there can be only $O(n)$ segments of category (i), one segment to the left of each endpoint of a segment in $S$ and one segment to the right of each endpoint of a segment in $S$.

We analyze the number of segments of category (ii) with the following charging scheme. Given a segment $u_h = s_{b,q}$ of category (ii), let $s_{a,p}$ be the segment $u_{h-1}$ and let $s_{c,r}$ be the segment $u_{h+1}$. We charge $u_h$ to $s_{a,p+1}$ and $s_{c,r-1}$. Observe that with this scheme, each segment $s_{i,j}$ can only be charged twice, namely by the successor of $s_{i,j-1}$ on the upper envelope and by the predecessor of $s_{i,j+1}$ on the upper envelope. Since each segment $s_i$ has only one leftmost and only one rightmost subsegment, and each is charged at most twice (in fact, once), there are at most $O(n)$ segments of category (ii) that put charges on leftmost or rightmost subsegments. If neither $s_{a,p+1}$ is the rightmost subsegment of $s_a$ nor $s_{c,r-1}$ is the leftmost subsegment of $s_c$, then $s_a$ must appear on the upper envelope again somewhere to the right of the right end of $s_b$, and $s_c$ must appear on the upper envelope again somewhere to the left of the left end of $s_b$ (see Figure 3) Therefore width($s_{a,p+1}$) + width($s_{c,r-1}$) > width($s_b$) $\geq \frac{1}{dn}\sum_{i=1}^{n}$ width($s_i$). Since each subsegment is charged at most twice, the total length of subsegments charged is at most $2\sum_{i=1}^{n}$ width($s_i$). Thus there are less than $2dn$ segments of category (ii) that put charges on subsegments that are not leftmost or rightmost. □

Note that the widths of the projections of the edges of layer $l$ on the view screen vary between $1/(l+1)$ and $1/(4l-2)$. Therefore, the widths of the projections of the edges of the $l/2$ outermost layers in a square region of $l$ layers around $v$ differ by less than a factor 8. Thus, from Lemma 1 we get:

COROLLARY 1. *If $S$ consists of the $O(l^2)$ edges of the $l/2$ outermost layers in a square region of $l$ layers around $v$, then the horizon of $S$ has complexity $O(l^2)$.*

Furthermore:

LEMMA 2. *If $S$ and $T$ are two $x$-monotone polylines of $m$ and $n$ vertices, respectively, then the upper envelope of $S$ and $T$ has at most $2(m+n)$ vertices.*

PROOF. There are two types of vertices on the upper envelope: vertices of $S$ or $T$, and intersection points between edges of $S$ and $T$. Clearly, there are at most $m+n$ vertices of the first type. Between any pair of vertices of the second type, there must be a vertex of the first type. Thus there are at most $m+n-1$ vertices of the second type. □

THEOREM 2. *If $S$ consists of $l$ layers in a square region around $v$, then the horizon of $S$ has complexity $O(l^2)$ in the worst case.*

PROOF. Let $T(l)$ be the complexity of the horizon of the innermost $l$ layers around $S$. By Lemma 2, $T(l)$ is at most twice the complexity of the horizon of the innermost $l/2$ layers, plus twice the complexity of the remaining $l/2$ layers. By Corollary 1, the latter is $O(l^2)$, and therefore we have $T(l) \leq 2T(l/2) + O(l^2)$. This solves to $T(l) = O(l^2)$. □

## 2.2 A refined algorithm: VIS-DAC

This section describes our second algorithm for computing viewsheds in the layers model, VIS-DAC. VIS-DAC is a divide-and-conquer refinement of VIS-ITER and uses the same general steps: it splits the grid into bands, computes visibility one band at a time, and creates the visibility grid from the bands. The first phase (BUILDBANDS) and last phase (COLLECTBANDS) are the same as in VIS-ITER; the only phase that is different is computing visibility in a band, VISBANDS-DAC, which aims to improve the time to merge horizons in a band using divide-and-conquer.

Similar to VISBANDS-ITER, VISBANDS-DAC processes the bands one at a time: for each band $k$ it loads list $E_k$ in memory, creates a visibility list $V_k$ and initializes it as all visible. It then marks as invisible all points that are below $H_{prev}$, where $H_{prev}$ represents the horizon of the first $k-1$ bands (more on this below). The bulk of the work in VISBANDS-DAC is done by the recursive function DAC-BAND, which computes and returns the horizon $H$ of $E_k$, and updates $V_k$ with all the points that are invisible inside $E_k$. This is described in detail below. Finally, the horizon $H$ is merged with $H_{prev}$ setting it up for the next band.

In order to mark as invisible the points in band $k$ that are below $H_{prev}$ we first sort the points in the band by azimuth

angle and then scan them in this order while also scanning $H_{prev}$ (recall that $H_{prev}$ is stored in ccw order). Let $(a_1 = 0, h_1), (a_2, h_2)$ be the first two points in the horizon $H_{prev}$. For every point $p = (a, h)$ in $E_k$ with azimuth angle $a \in [a_1, a_2]$, we check whether its height $h$ is above or below the height of segment $(a_1, h_1)(a_2, h_2)$ in $H_{prev}$. When we encounter a point in $E_k$ with $a > a_2$, we proceed to the next point in $H_{prev}$ and repeat.

The recursive algorithm DAC-BAND takes as arguments an elevation band $E_k$, a visibility band $V_k$, and the indices $i$ and $j$ of two layers in this band ($w_{k-1} \leq i \leq j < w_k$). It computes visibility for the points in layers $i$ through $j$ (inclusive) in this band, and marks in $V_k$ the points that are determined to be invisible during this process. In this process it also computes and returns the horizon of layers $i$ through $j$ in this band. DAC-BAND uses divide-and-conquer in a straightforward way: first it computes a "middle" layer $m, i \leq m \leq j$ between $i$ and $j$ that splits the points in layers $i$ through $j$ approximately in half. Then it computes visibility and the horizon recursively on each side of $m$; marks as invisible all points in the second half that fall below the horizon of the first half; and finally, merges the two horizons on the two sides and returns the result.

**Algorithm** DAC-BAND($E_k, V_k, i, j$):
**if** $i == j$
  $h \leftarrow$ compute-layer-horizon(i)
  return $h$
**else**
  $m \leftarrow$ middle layer between $i$ and $j$
  $h_1 \leftarrow$ DAC-BAND($E_k, V_k, i, m$)
  $h_2 \leftarrow$ DAC-BAND($E_k, V_k, m+1, j$)
  mark invisible all points in $L_{m+1,j}$ that fall below $h_1$
  $h \leftarrow$ merge($h_1, h_2$)
  return $h$

*Efficiency analysis of* VIS-DAC .

The analysis of the first and last phase of VIS-DAC, BUILDBANDS and COLLECTBANDS, is the same as in Section 2.1. We now analyze VISBANDS-DAC. Recall that we can assume that $E_k$ and $V_k$ both fit in memory during this phase (see Section 2.1). The elevation and visibility of any point in a band can be accessed in $O(1)$ time, without any search structure and without any I/O. We denote $H^B_{1,i}$ the horizon of (the points in) the first $i$ bands; and by $H^B_{tot} = H^B_{1,1} + H^B_{1,2} + H^B_{1,3} + ... = \sum_{i=1}^{N_{bands}} H^B_{1,i}$.

- Marking as invisible the points in $E_k$ that are below $H_{prev}$ (here $H_{prev}$ represents $H^B_{1,k-1}$): this can be done by first sorting $E_k$ and then scanning $H^B_{1,k-1}$ and $E_k$ in sync. Over the entire grid, this takes $O(n \lg n + H^B_{tot})$ CPU and $O(\mathsf{scan}(n) + \mathsf{scan}(H^B_{tot}))$ I/Os.

- Merging horizons: After DAC-BAND is called in a band, the returned horizon is merged with $H_{prev}$. Two horizons can be merged in linear time and I/Os. Over the entire grid this takes $O(H^B_{tot})$ time and $O(\mathsf{scan}(H^B_{tot}))$ I/Os.

- DAC-BAND: This is a recursive function, with the running time given by the recurrence $T(k) = 2T(k/2) +$ `merge cost + update cost`, where $k$ is the number of points in the slice between layers $i$ and $j$ given as input. The base case computes the horizon of a layer $l$, which takes linear time wrt to the number of points in the layer. Summed over all the layers in the slice the base case takes $O(\sum_{l=i}^{j} |L_l|) = O(k)$ time and no I/O (band is in memory).

- The update time in DAC-BAND represents the time to mark as invisible all points in the second half that fall below the horizon $h_1$ of the first half. Recall that a band fits in memory and thus an input slice in `DacBand` fits in memory. If the band is sorted, the update can be done as above in $O(k + |h_1|) = O(k)$ time (by Theorem 2 we have $|h_1| = O(k)$).

- The merge time in DAC-BAND represents the time to merge the horizons $h_1$ and $h_2$ of the first and second half of the slice, respectively. This takes $O(|h_1| + |h_2|) = O(k)$ time.

- Putting it all together in the recurrence relation we get $T(k) = 2T(k/2) + O(k)$, which solves to $O(k \lg k)$ time. Summed over all bands in the grid DAC-BAND runs in $O(n \lg n)$ time and $O(\mathsf{scan}(n))$ I/Os.

Overall we have the following:

THEOREM 3. *The algorithm* VIS-DAC *computes viewsheds in the layers model in $O(n \lg n + H^B_{tot})$ time and $O(\mathsf{sort}(n) + \mathsf{scan}(H^B_{tot}))$ I/Os, provided that $n \leq cM^2$, for a sufficiently small constant $c$.*

**Discussion:** The worst case complexity of $H^B_{tot}$ is $\sum_{i=1}^{N_{bands}} H^B_{1,i} = O(N_{bands} \cdot n) = O(n^2/M)$; This is an improvement over $O(n\sqrt{n})$ ( provided that $n \leq cM^2$).

Consider a band that extends from layer $L_i$ to layer $L_j$ and contains $k$ points. The algorithm DAC-BAND runs in $O(k \lg k)$ time, while the iterative algorithm VISBANDS-ITER scans iteratively through all cumulative horizons of the layers in the band $H_{1,i}, H_{1,i+1}, ...$ and so on and runs in $O(k + |H_{1,i}| + |H_{1,i+1}| + ... + |H_{1,j}|)$. When the horizons are small, VIS-ITER runs in $O(k)$ time and is faster than VIS-DAC. The divide-and-conquer merging is not justified unless the horizons are large enough to benefit from it.

## 2.3 The gridlines model

The algorithms VIS-DAC and VIS-ITER described in Section 2.1 and 2.2 above compute viewsheds in the layers model. Let $X_i$ denote the line segments connecting points at distance $i-1$ with points at distance $i$ (Figure 4). The set $X_i$ represents the additional "obstacles" in the $i$th layer that could intersect the LOS in the gridlines model. With this notation the horizon of the $i$th layer in the gridlines model is $H(L_i) \cup H(X_i)$. The algorithms VIS-ITER and VIS-DAC can be extended to compute viewsheds in the gridlines model—the only difference is that they compute the horizon of a layer as $H(L_i) \cup H(X_i)$ instead of $H(L_i)$. Since $|X_i| = \Theta(|L_i|)$, the analysis and the bounds of the algorithms are the same in both models up to a constant factor.

Our algorithms VIS-ITER and VIS-DAC, when using the gridlines model, compute the same viewshed as `R3` [8]. VIS-DAC's upper bound of $O(n \lg n + n^2/M)$ is an improvement over `R3`'s bound of $O(n\sqrt{n})$, provided that $n \leq cM^2$.

The results on the worst-case complexity of the horizon in the layer model extend to the gridlines model. The extension is not entirely straightforward, because the differences in width in the projection between non-layer edges are larger than between layer edges. The main idea of the

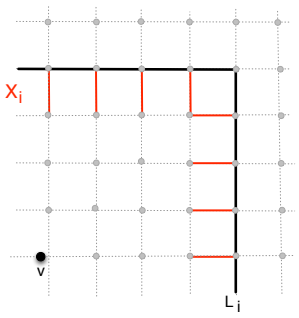



**Figure 4: The segments contributing to a layer's horizon in the gridlines model**

analysis is to focus on one "octant", for example the area northeast of the viewpoint. Let $S(l)$ be the complexity of the horizon of the edges with southwest endpoints in $[0, l] \times [0, l]$, let $U(l, y)$ be the complexity of the horizon of the edges with southwest endpoints in $(l/3, l] \times (y/3, y]$, and let $R(l, y)$ be the complexity of the horizon of the edges with southwest endpoints in $(l/3, l] \times [0, y]$. By Lemma 2 we have $S(l) \leq 2R(l, l) + 2S(\lfloor l/3 \rfloor)$ and $R(l, y) \leq 2U(l, y) + 2R(l, \lfloor y/3 \rfloor)$, with base cases $S(0, 0) = O(1)$ and $R(l, 0) = O(l)$. Since the widths of the $O(ly)$ non-layer edges contributing to $U(l, y)$ differ by at most a constant factor, and the widths of the $O(ly)$ layer edges contributing to $U(l, y)$ differ by at most a constant factor, we have, by Lemmas 1 and 2, $U(l, y) = O(ly)$. Thus, by the Master method, the recurrence of $R(l, y)$ solves to $R(l, y) = O(ly)$ for any fixed $l$, and in particular, $R(l, l) = O(l^2)$. Now, again by the Master method, we get $S(l) = O(l^2)$.

# 3. EVALUATING VIEWSHED ACCURACY

In order to evaluate the accuracy of a viewshed algorithm we need to compare its output with ground truth data. Unfortunately ground truth viewshed data is practically impossible to obtain. In the absence of ground truth data we analyze the accuracy of various viewshed algorithms by comparing them to a reference algorithm. While this is not a perfect indication of accuracy, it does give an idea of where an algorithm stands compared to another algorithm. In our experiments (see Section 4) we chose as reference the algorithm for computing viewsheds `r.los` in GRASS. The advantage of using `r.los` as a reference is that it is part of a well known GIS, it is widely used and tested by thousands of users worldwide, it is available free of charge and and its code is open source. There is no documentation on what model and algorithm it uses.

Let $A$ and $B$ denote two viewshed algorithms and let $V_A(v)$ and $V_B(v)$ be the viewsheds computed by these algorithms from an arbitrary viewpoint $v$ on some test grid. When comparing $V_A(v)$ and $V_B(v)$ we distinguish four type of points: (1) points that are labeled visible by both $A$ and $B$; (2) points that labeled invisible by both $A$ and $B$); (3) points that are labeled invisible by $A$ and visible by $B$—- referred as *false-visible*; and (4) points that are labeled visible by $A$ and invisible by $B$)—referred as *false-invisible*.

The set of points in a grid $G$ misidentified by $B$ when using $A$ as reference is the set of false-visible and false-invisible points. This is the error (difference) $E_{A,B}(v)$ at point $v$. The error over the entire grid is $E_{A,B} = \sum_{v \in G} E_{A,B}(v)$ Computing viewsheds from *all* the points in a grid may be unfeasible on grids of even modest sizes. For example, if a viewshed computation on a grid of 200,000 points takes on the order of $.5s$, computing viewsheds from all the 200,000 points in the grid will take >20 hours. We approximate the error using a sample $\chi$ of points in $G$:

$$E_{A,B}(\chi) = \sum_{v \in \chi} E_{A,B}(v)$$

This formulation of viewshed error was used by Ben Moshe et al. [3, 2], who pick the sample $\chi$ uniformly at random over $G$. Instead, we select $\chi$ from the set of points that lie on the channels and ridges of the terrain. It is generally agreed that these points represent important topographic features. These are the points where it is important for the viewshed to be accurate and where we want to know the error.

# 4. EXPERIMENTAL ANALYSIS

In this section we describe the results of preliminary experiments to assess the performance and accuracy of our algorithms. We have two algorithms—VIS-ITER and VIS-DAC, and two models—layers and gridlines. We denote the 4 combinations as follows: `iter-layers`, `dac-layers`, `iter-gridlines`, and `dac-gridlines`. Their implementation follows closely the description in Section 2.

**Platform.** The algorithms are implemented in C and compiled with g++ 4.1.2 with optimization level -O3. All experiments were run on HP 220 blade servers, with an Intel 2.83 GHz processor, 512MB of RAM and a 5400 rpm SATA hard drive. The hard disk is standard speed for a laptop.

The only IO-primitive used in our algorithms is scanning, and we are not using an IO-library. We store data on disk (for e.g. horizons and bands) in regular Unix files.

**Datasets.** The algorithms were tested on terrains up to over 7.6 billion elements, see Table **??**. The largest datasets are SRTM1 data, 30m resolution, see `http://www2.jpl.nasa.gov/srtm/cbanddataproducts.html/`. We note that SRTM datasets regions 1 through 5 are relatively close in size, having between 2 and 3 billion points each; while `region06` is 3 times larger at 7.6 billion points.

## 4.1 Running time of VIS-DAC and VIS-ITER

To assess the efficiency of our algorithms we ran experiments for each dataset choosing one viewpoint approximately in the middle of the terrain. This gives a good indication of the algorithm's performance, and is consistent with the experiments in our previous work [7]. We note that for all our algorithms the majority of the running time is spent handling the bands and we expect that the total running time will not vary significantly with the position of the viewpoint. Nevertheless, we believe that for a thorough analysis one should average the running time over a large number of viewpoints (as each viewshed takes on the order of hours, this will take a considerable amount of time).

For all datasets we found that $n \leq cM^2/B$ for a sufficiently small constant $c$, which means that BUILDBANDS and COLLECTBANDS run in a single pass over the data. Thus all algorithms perform in total 3 passes over the grid (one to split the grid into bands, one to compute the visibility bands, and one assemble the visibility bands into a grid). The total running time is split fairly evenly between the three phases.

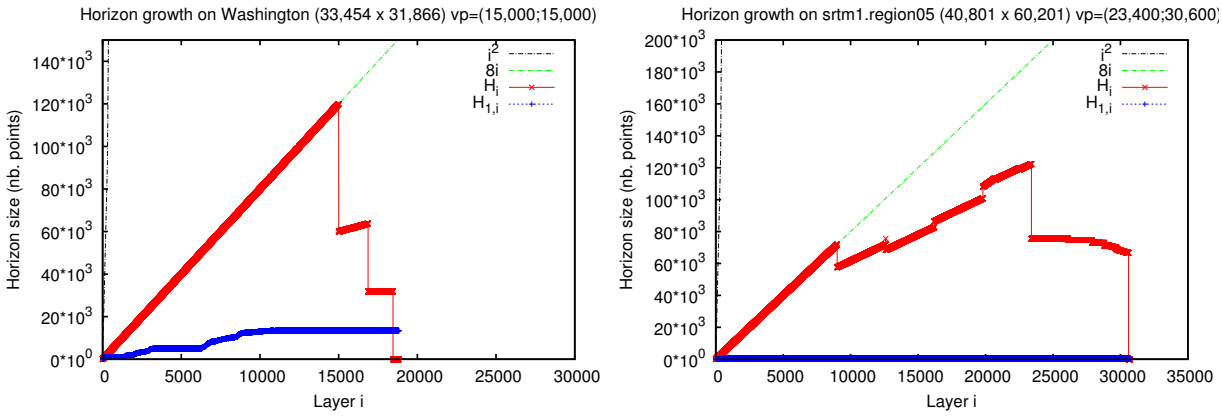



**Figure 5: Horizon growth for a viewshed computation on datasets `Washington` and `srtm1.region05`**

The actual visibility calculation runs at 100% CPU and represents <25% of the running time. More than 75% of the total time is spent reading or writing the bands.

In all our tests we found that the iterative algorithm VIS-ITER is consistently 10-20% faster than VIS-DAC; and both our new algorithms are faster than the algorithms in our previous work [7], while also having better accuracy (see Table **??**). To understand this we investigated the size of the horizons computed by VIS-DAC and VIS-ITER: $H_i$, the horizon of layer $i$; and $H_{1,i}$, the horizon of the points in the first $i$ layers. Note that the number of grid points on level $i$ is $8i$, and the total number of points on levels 1 through $i$ is $4i^2+4i+1 = \Theta(i^2)$. We know that $H_i = O(i) = O(\sqrt{n})$, and $H_{1,i} = O(i^2) = O(n)$ (Theorem 2). We recorded $|H_i|$ and $|H_{1,i}|$ for each layer $i$ during the execution of `iter-layers`. Figure 5 shows the results for two datasets; the results for the other datasets look similar.

We see that $|H_i|$ is very close to its theoretical bound of $8i = \Theta(i)$. As $i$ gets larger the $i$th layers starts to fit only partially in the grid, and this causes $|H_i|$ to drop and have steep variations. The main finding is that for all datasets $H_{1,i}$ stays very small, far below its theoretical upper bound of $\Theta(i^2)$. $H_{1,i}$ grows fast initially and then flattens out; For e.g. on `Washington` dataset (approx. 1 billion points), $|H_{1,i}|$ flattens at 13,452 points; and on `srtm1.region05` (approx. 2.5 billion points), $|H_{1,i}|$ flattens at 460 points. All SRTM datasets have the horizon $H_{1,O(\sqrt{n})}$ between 132 and 32,689.

Given a dataset, we refer to the horizon $H_{1,O(\sqrt{n})}$ as its *final* horizon. Figure 6(a) shows the size of the final horizon for each dataset as function of the number of *valid* points in the grid —- this excludes the points in the grid that are labeled as *nodata*, and which are used for e.g. to label the water/ocean; these points do not affect the size of the horizon, as chains of *nodata* points are compressed into a single horizon segment. We see that the final horizon: (1) has a lot of variation especially for the larger SRTM datasets, jumping from low to high values. This is likely due to the position of the viewpoint and possibly the topology of the terrain; (2) the horizon stays small, below $\sqrt{n}$ for all datasets, far below its worst-case bound of $O(n)$.

Figure 6(b) shows the cumulative sums, $\sum_{i=1}^{O(\sqrt{n})} |H_{1,i}|$ and $\sum_{i=1}^{O(\sqrt{n})} |H_i|$, for each dataset, as a function of the number of valid points in the grid; we recorded these sums because they come up in the analysis of VIS-ITER and can shed light on its performance. In Figure 6 we see that $\sum_{i=1}^{O(\sqrt{n})} |H_i|$ grows indeed linearly with the number of valid points in the grid. The sum $\sum_{i=1}^{O(\sqrt{n})} |H_{1,i}|$ has a lot of variability similar with the final horizon shown in Figure 6 (a), and for all datasets stays far from its worst-case upper bound of $O(n\sqrt{n})$. We note that Figure 5 and 6 are based on a single viewpoint and thus represent preliminary results; more experiments are needed that average over a large number of viewpoints, but we expect the results will carry over.

Comparing to the most recent work of Ferreira et al. [6]: Their algorithm, TILEDVS, also consists of three passes: convert the grid to Morton order, compute visibility using the `R2` algorithm, and convert the output grid from Morton order to row-major order. They report on the order of 5,000 seconds for `SRTM1.region06`, using a similar platform as ours and additional optimization like data compression. Assuming that this time includes all three passes, and modulo variations in setup, it is approx. 2.5 times faster than VIS-ITER. Recall that `R2` does not compute the "exact" viewshed, but an approximation. The advantage of TILEDVS is that its first step can be viewed as a preprocessing step common to all viewpoints, and thus TILEDVS computes the viewshed in only two passes over the grid.

## 4.2 Accuracy

A comparison of viewshed algorithms needs to look at both performance and accuracy. As described in Section 3, we assess the accuracy of a test algorithm compared to a reference algorithm by computing the number of false visible and false invisible points, $fv$ and $fi$, from a sample of viewpoints. For each trial viewpoint, $fv$ and $fi$ are reported as percentages of the size of the vieshed computed with the reference algorithm. We obtain the ridges and river network using the flow accumulation grid computed with the module r.terraflow in GRASS[2]. As reference algorithm we use the viewshed module r.los[3] in GRASS.

We implemented a tool that takes as input a terrain, its flow accumulation grid, a reference and a test algorithm, and a parameter that controls the number of trials. We ran it on two terrains of size approx. 500 by 500 points: a real terrain representing the watershed of a river in a mountain-

[2]See `http://grass.osgeo.org/grass64/manuals/r.terraflow.html`
[3]See `http://grass.osgeo.org/grass64/manuals/r.los.html`

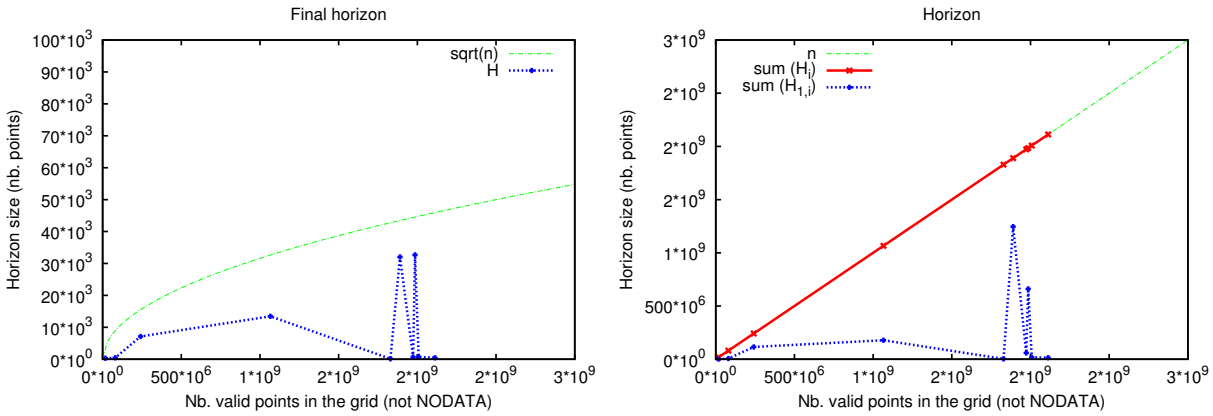



**Figure 6: (a) Size of final grid horizon $|H_{1,O(\sqrt{n})}|$ with dataset size. (b) $\sum_{i=1}^{O(\sqrt{n})} |H_{1,i}|$ and $\sum_{i=1}^{O(\sqrt{n})} |H_i|$ with dataset size.**

ous area, and an artificially generated terrain the shape of a hemisphere. The results on both terrains are similar.

Over 71 trials, our algorithm `iter-layers` has an average number of false visible points $fv = .2\%$ and an average number of false invisible points $fi = 4.2\%$. Our algorithm `iter-gridlines` has $fv = .1\%$ and $fi = 4.3\%$. The difference between `iter-layers` and `iter-gridlines` is practically insignificant; compared to `iter-layers`, `iter-gridlines` has $fv = 0$ and $fi = .2\%$. We conclude that the layers model is simpler, faster and computes practically the same viewshed as `iter-gridlines`.

We compared `r.los` with the algorithms `io-radial3` and `io-centrifugal` in our previous work [7]. Over the same number of trials, `io-radial3` has $fv = 53.3\%$ and $fi = 13.9\%$; and `io-centrifugal` has $fv = 7.6\%$ and $fi = 32.9\%$. We see that the viewshed generated by `io-radial3` and `io-centrifugal` are significantly different compared to those generated by `r.los`.

Finally, we compared `r.los` with our implementation of the `R2` algorithm used by Ferreira et al. in TiledVS [6]. `R2` was introduced as a speedup of `R3` that achieves good accuracy. Indeed, over 71 trials, `R2` has average $fv = 6.9\%$ and $fi = 7.2\%$. Given its fast runing time, `TiledVS` is useful in situations where repeated viewshed computations are performed and where an error level of 7% is acceptable.

We conclude that accuracy plays an important role in assesing viewshed algorithms, and that horizon-based algorithms emerge as a fast approach for computing viewsheds on grids due to the small size of horizons on grids. As avenues for further research we mention the problem of proving a sublinear bound for the expected complexity of the horizon on a grid of $n$ points; obtaining an output sensitive viewshed algorithm; and obtaining faster approximation algorithms that achieve similar accuracy while doing a single pass over the grid and possibly some preprocessing.

## 5. REFERENCES


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

[2] Boaz Ben-Moshe, Paz Carmi, and Matthew J. Katz. Approximating the visible region of a point on a terrain. *Geoinformatica*, 12(1):21–36, 2008.

[3] Boaz Ben-Moshe, Matthew J. Katz, and Igor Zaslavsky. Visibility preserving terrain simplification: an experimental study. *Computational Geometry*, 28(2-3):175–190, 2004.

[4] Richard Cole and Micha Sharir. Visibility problems for polyhedral terrains. *J. Symbolic Computation*, 7:11–30, 1989.

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

[6] Chaulio R. Ferreira, Salles V. G. Magalhães, Marcus Andrade, W. Randolph Franklin, and Andre M. Pompermayer. More efficient terrain viewshed computation on massive datasets using external memory. In *Proc. ACM SIGSPATIAL Symp. Geographic Information Systems (GIS 2012)*, pages 169–172, 2012.

[7] Jeremy Fishman, Herman Haverkort, and Laura Toma. Improved visibility computations on massive grid terrains. In *Proc. 17th ACM SIGSPATIAL Symp. Geographic Information Systems (GIS 2009)*, pages 121–130, 2009.

[8] W. Randolph Franklin and Clark Ray. Higher isn't necessarily better: visibility algorithms and experiments. In *Proc. 6th Symp. Spatial Data Handling (SDH 1994)*, pages 751–763, 1994.

[9] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6:151–177, 1986.

[10] David Izraelevitz. A fast algorithm for approximate viewshed computation. *Photogrammetric Engineering and Remote Sensing*, 69(7):767–774, July 2003.

[11] Marc van Kreveld. Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals. In *Proc. 7th Symposium of Spatial Data Handling (SDH 1996)*, pages 15–27, 1996.

[12] A. Wiernik and M. Sharir. Planar realization of nonlinear Davenport-Schinzel sequences by segments. *Discrete and Computational Geometry*, 3:15–47, 1988.