Empirical Comparison of Data Structures for Line-Of-Sight Computation

Christoph Fünfzig
PRISM Lab, Arizona State University, Tempe, USA
Torsten Ullrich, Dieter W. Fellner
Graz University of Technology, Austria
Edward N. Bachelder
Systems Technology Inc, Hawthorne, California, USA

Abstract **– Line-of-sight (LOS) computation is important for interrogation of heightfield grids in the context of geo information and many simulation tasks like electromagnetic wave propagation and flight surveillance. Compared to searching the regular grid directly, more advanced data structures like a 2.5d kd-tree offer better performance. We describe the definition of a 2.5d kd-tree from the digital elevation model and its use for LOS computation on a point-reconstructed or bilinear-reconstructed terrain surface. For compact storage, we use a wavelet-like storage scheme which saves one half of the storage space without considerably compromising the runtime performance. We give an empirical comparison of both approaches on practical data sets which show the method of choice for CPU computation of LOS.**

Keywords **– line-of-sight computation, heightfield interrogation, kd-tree data structure, pyramid algorithms**

I. Introduction

Digital terrain models (DTM) can be represented in two forms, as a triangulated irregular network (TIN) and as a regular height grid (DEM). With the acquisition by airborne laser altimetry, especially the latter are available in good resolutions (30m/10m resolution for the USA [1] and 1km resolution for the earth [2] in publicly available datasets). Triangular irregular networks can be generated from DEMs by simplification for compact lossy storage and for efficient rendering by computer graphics. The deviation measure used for simplification is of special importance for the application [3]. Many simulation and planning problems for electromagnetic wave propagation are based on the ray tracing model like signal strength prediction [4], [5] and antenna placement [6]. In these models, the main paths of wave propagation are evaluated by rays (geometric optics), and special wave effects are added at the intersection points of the rays with the terrain surface. Of course, the line-of-sights from the antennas account for the main propagation paths. So efficient line-of-sight (LOS) computation is an important means for terrain interrogation.

Contact Email: c.fuenfzig@gmx.de

In this paper, we present efficient data structures for DEMs and their use for line-of-sight computation. We describe a 2.5d extension of the kd-tree, which has been invented for organizing point sets in arbitrary dimensions for nearest neighbor searching [7]. Section III describes the construction of the kd-tree in detail and how to traverse it for line-of-sight queries. The 2.5d extension allows to skip low height regions during line-of-sight computation for better data-dependent performance. Although the data structure requires storage space which is linear in the number of elevation values given in the original DEM, it has a number of inner nodes building the binary tree structure. Therefore, we devise a scheme storing the same information as the 2.5d kd-tree and merging two levels of the binary tree into a 4-ary recursive structure. The storage space is nearly the same as the original DEM and it is stored in a 2d array (Section IV). In Section V we describe optimizations and extensions of the kd-tree traversal for line-of-sight queries and for computing the minimum height having a line-of-sight. Section VI gives a detailed empirical comparison for different data set sizes and different numbers of ground stations. Finally, Section VII summarizes the results and concludes on the importance of the technique for line-of-sight computation.

II. Previous Work

The most straightforward approach traverses the projection of the ray on the domain plane and checks the ray heights against the heightfield heights at a number of points per cell [8]. Different terrain reconstruction is possible for non-integer grid positions: No interpolation (point reconstruction), double linear interpolation and bilinear interpolation. For an exact line-of-sight test, a specific intersection test is necessary for the reconstruction used. By checking a number p of discrete points per grid cell, large low-height regions cannot be exploited. The approach requires in the worst case $l \cdot p$ height evaluations regardless of the terrain traversed, where l is the ray length in cells.

Lately, also graphics processing units (GPU) have been used to determine line-of-sight on a terrain, see [9], [10]. These approaches render both the terrain surface and the ray line and test if all line fragments are above the terrain surface. If this is the case, it is a line-of-sight up to the image resolution used for rendering and for terrain reconstruction. A special hardware occlusion test is used for counting the number of visible fragments.

There is also work on slightly extended visibility problems like computing the horizon for each grid point and direction sector [11], [12]. Originally, the resulting horizon map was invented for self-shadowing of the terrain surface. In [13] the horizon map computation is optimized for a large number of points. Stewart presents an algorithm with runtime $O(sk(\log^2 k + s))$ for s horizon sectors per point and k points.

III. KD-Tree Ray Casting

A ray is defined by an origin point $\vec{o}$ and a direction vector $\vec{d}$, which implicitly gives a search order on the data domain traversed by the ray. So if the data domain is partitioned into several parts then these parts can be searched according to the ray. *Line-of-sight computation* is a problem that can be solved in this way. Additionally, the parameter interval of the ray inside a domain part is available during traversal.

The simplest partition tree for a 2-dimensional domain is a binary tree with domain-orthogonal partitioning planes. The parameter interval $[0, \infty]$ is subdivided by the ray-plane intersection point at parameter

$$\begin{array}{ll} \textit{split} & = \vec{n} \cdot (\vec{p} - \vec{o}) / \vec{n} \cdot \vec{d}, \\ \text{with} & \vec{p} \text{ an arbitrary plane point,} \\ & \vec{n} \text{ the plane normal,} \end{array}$$

into the near interval $[0, \textit{split}]$ and the far interval $[\textit{split}, \infty]$.

For axis-orthogonal partitioning planes (see Figure 1) the ray-plane intersection computation $\textit{split} = (p_x - o_x)/d_x$ resp. $\textit{split} = (p_y - o_y)/d_y$ is simple and therefore especially fast to compute. The resulting partitioning tree with alternating x- and y-orthogonal planes is called *kd-tree* of dimension 2. The traversal of the kd-tree can be adapted for ray segments restricted to a parameter interval $[t_{\text{near}}, t_{\text{far}}]$. The traversal visits the near node iff $[0, \textit{split}] \cap [t_{\text{near}}, t_{\text{far}}] = [t_{\text{near}}, \textit{split}]$ is not empty and it visits the far node iff $[\textit{split}, \infty] \cap [t_{\text{near}}, t_{\text{far}}] = [\textit{split}, t_{\text{far}}]$ is not empty. In this way the interesting parameter interval of the

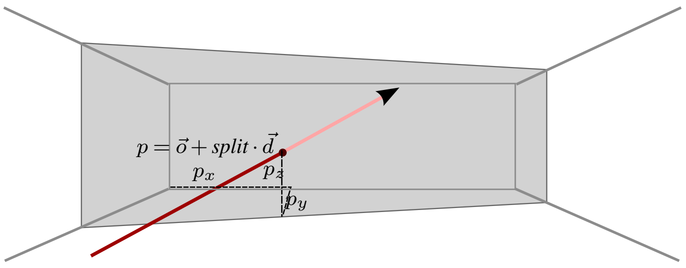


Fig. 1. A partitioning plane subdivides the ray into a near segment and a far segment at the ray-plane intersection point.

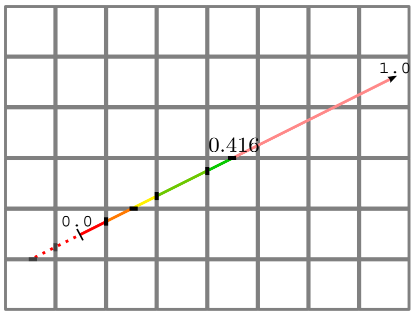


Fig. 2. Ray partitioning by a 2-dimensional kd-tree. The ray segments resulting from plane intersections are shown in different colors.

ray is always available for the current node. The special cases, where the ray is parallel to the partitioning plane ($\textit{split} = \infty$) or pointing away from the partitioning plane ($\textit{split} < 0$), are easy to handle.

We described so far the traversal of a 2-dimensional domain (the heightfield plane) by a kd-tree of dimension 2. For a 2.5-dimensional heightfield, which is a real height function on the 2-dimensional domain, the approach can be extended. If for each kd-tree node the maximum height is available then it can be used to prune subtrees of the kd-tree from traversal. For a ray with parameter interval $[t_{\text{near}}, t_{\text{far}}]$ and height $z_{\text{near}} = o_z + t_{\text{near}} d_z$ at entry and height $z_{\text{far}} = o_z + t_{\text{far}} d_z$ at exit, it can intersect a node with maximum height h_{max} only if $z_{\text{near}} < h_{\text{max}}$ or $z_{\text{far}} < h_{\text{max}}$. This simple extension requires only one additional real value per inner node of the kd-tree.

As the domain part represented by a kd-tree node shrinks with each subdivision, the leaf nodes represent single entries of the heightfield grid. From these discrete measurements, terrain surfaces can be reconstructed of various orders of continuity and of polynomial type. The question which one represents the terrain best for a special application has received considerable interest [14], [8].

We have concentrated on point reconstruction which represents the surface as a constant within its grid cell and on bilinear reconstruction which is a degree-two surface with linear x-parallel and y-parallel intersections. Figure 4 shows plots of both reconstructions for samplings of the function $f(x, y) = -8x^3 - 12x^2 + 3xy^2 + y^3 + 3y^2$. Both reconstructions can be efficiently incorporated into the kd-tree. The key to efficiency here is that the grid neighborhood required for the reconstruction is exactly available in a small subtree of the kd-tree. For point reconstruction, the neighborhood is small, $1 \cdot 1$. For bilinear

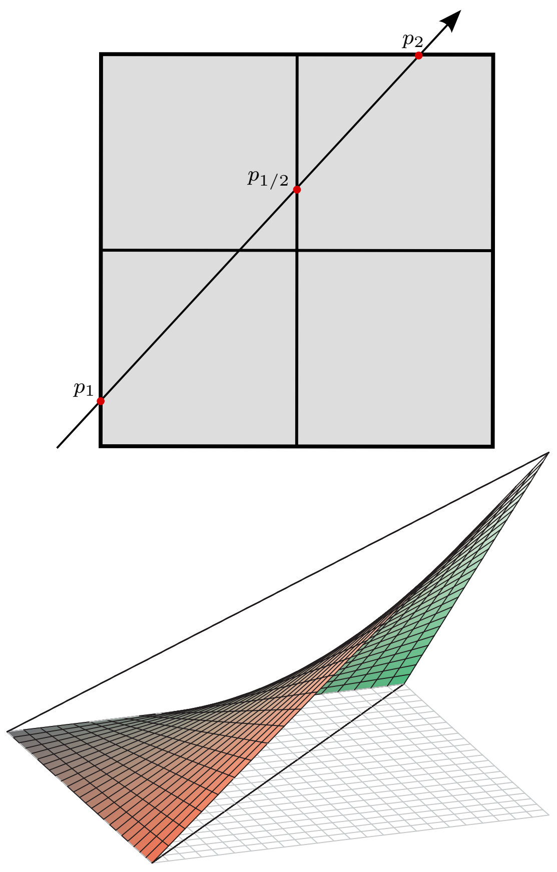


Fig. 3. RAY-INTERSECTION WITH A BILINEAR SURFACE TILE. NOTE, THE SURFACE IS A HYPERBOLIC PARABOLOID AND HAS LINEAR INTERSECTIONS ONLY WITH X-PARALLEL AND Y-PARALLEL PLANES SHOWN AS WIREFRAME.

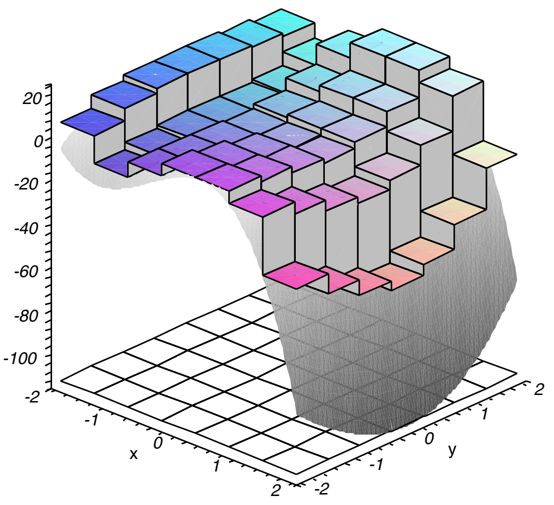


Fig. 4. SURFACE RECONSTRUCTION FROM GRID SAMPLES, HERE FOR THE FUNCTION $f(x,y) = -8x^3 - 12x^2 + 3xy^2 + y^3 + 3y^2$. POINT RECONSTRUCTION FROM MAXIMUM VALUES AND BILINEAR RECONSTRUCTION FROM FOUR CORNER VALUES.

reconstruction, it is larger, $2 \cdot 2$. This has some consequences for the tree construction described below.

For the intersection computation between the linear ray and the bilinear surface see Figure 3. Firstly, the intersection points p_1, $p_{1/2}$ and p_2 with the domain rectangle are calculated. The surface heights are added there by bilinear interpolation. From these three values the parabola in the ray plane is uniquely determined. By equating the ray segment and the parabola we can solve the intersection problem. For intersection testing, it is computationally faster and more robust not to compute all possible (two) solutions but to compute the parameter value of the parabola apex and test the ray height against the apex height just there.

In principle, there are two approaches for tree construction, recursive top-down construction and iterative bottom-up construction [7]. Due to its simplicity, we use a recursive top-down construction, which chooses a split index along a split axis which alternates between x- and y-axis. For a perfectly balanced tree, the split index is chosen as the center. A data-dependent tree construction could take the height values in the current kd-tree node into account. One could choose the split index which creates two boxes of minimum sum of volumes or surface area. Due to the restriction of splitting planes to axis-aligned orientations, the separation of small and large heights is not so good, and the data-dependent construction is usually not worth it. For bilinear reconstruction, we process each value together with its left and lower neighbor ($2 \cdot 2$ neighborhood). This can be achieved simply by creating a one row and one column overlap when splitting the current node (*overlapped splitting*). Compared to the *disjoint splitting*, it generates four times as many inner nodes because of the overlap.

With disjoint splitting the height of the kd-tree for a heightfield of size $n \cdot m$ is $\log_2(n \cdot m)$, and the number of nodes is $2(n \cdot m)$. With overlapped splitting the height is $\log_2(n \cdot m) + 2$ and $5(n \cdot m)$ nodes. Notice that the $n \cdot m$ leaf nodes are reused in $3(n \cdot m)$ places for the overlap. The data per node consist of two pointers (4 bytes each on a 32-bit architecture), the real maximum height (8 bytes in double precision), and the real split value (8 bytes in double precision). So that the memory requirements sum up to 20 bytes per node (except for data alignment). For comparison, the given heightfield array consists of $n \cdot m$ height values with 8 bytes per entry.

IV. RAY CASTING WITH COMPRESSED GRID

The wavelet-like, compressed grid for a heightfield of size $n \times m$ has the same memory requirements as the square grid of side length $\max(\tilde{n}, \tilde{m})$ resp. $2 \cdot \max(\tilde{n}, \tilde{m})$ with overlapped splitting where $\tilde{x}$ denotes the smallest power of two greater than or equal to x.

The inefficiency for non-power-of-two and non-square formats can be eliminated, for example, by tiling techniques (as done in JPEG2000 [15]) or by a boolean sequence, which gives the orientation of the split, i.e., x- or y-orthogonal. In this way the wavelet-like compressed grid is a different storage scheme for the kd-tree of Section III. But if sufficient memory is available, the square, power-of-two format with its implicit, 4-ary splitting is beneficial for runtime efficiency.

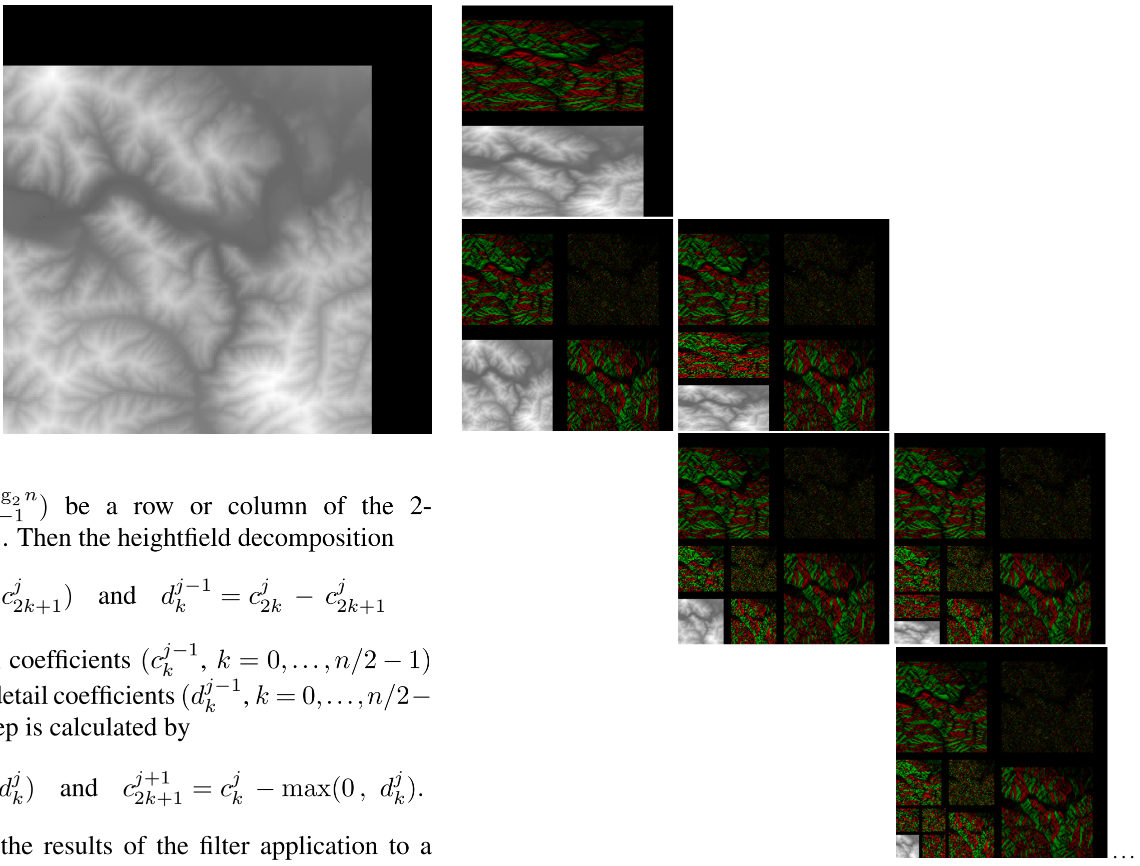


Fig. 5. Two-dimensional heightfield with non-standard decomposition. Green pixels mark positive coefficients d_k^j, and red pixels mark negative coefficients d_k^j.

Let $(c_0^{\log_2 n}, \ldots, c_{n-1}^{\log_2 n})$ be a row or column of the 2-dimensional signal [16]. Then the heightfield decomposition

$$c_k^{j-1} = \max(c_{2k}^j\,,\; c_{2k+1}^j) \quad \text{and} \quad d_k^{j-1} = c_{2k}^j \;-\; c_{2k+1}^j$$

provides the maximum coefficients $(c_k^{j-1},\, k = 0, \ldots, n/2-1)$ and the corresponding detail coefficients $(d_k^{j-1}, k = 0, \ldots, n/2-1)$. The composition step is calculated by

$$c_{2k}^{j+1} = c_k^j \;+ \min(0\,,\; d_k^j) \quad \text{and} \quad c_{2k+1}^{j+1} = c_k^j \;- \max(0\,,\; d_k^j).$$

Figure 5 illustrates the results of the filter application to a quadratic heightfield of size $220 \cdot 220$. In the non-standard decomposition used, the filter application alternates between columns and rows.

The analysis filter can be described in a short manner using operations in the max-plus algebra $_{\max}$ [17]. In this way the filter process is very similar to one of Haar wavelets. The algebra $_{\max}$ is only an idempotent semiring, and the detail filter is not linear in this algebra.

V. Optimizations and Extensions

For efficiency, we have employed some optimizations. On the one hand, we use standard techniques like a non-recursive traversal implementation with a stack, a reduction of memory footprint as much as possible and the avoidance of virtual methods in classes accessed at a high rate like `KDTreeNode`.

Further conceptual optimizations are also possible. A ray usually has more than one intersection with the heightfield. In the worst case, the query time is $O(l + log_2(n \cdot m))$ which is mainly determined by the traversal length l of the ray up to the first intersection. So it is reasonable to traverse from origin to destination forward if an intersection is nearby the origin, and backward if one is nearby the destination. This can be exploited if there are hints available, for example, from results of queries with an origin point or a destination point in the neighborhood (*spatial coherency*).

The kd-tree traversal can also be extended for computing the minimum height with a line-of-sight. By only changing the leaf node code. Instead of testing ray heights against heightfield heights, now the minimum height required for a line-of-sight is stored in the ray destination point. This can be updated with each leaf node traversed. Minimum height computation is also possible without a special kd-tree traversal by employing binary search. Starting with a height interval $[d_{\text{min}}, d_{\text{max}}]$ with d_{min} having no LOS and d_{max} having a LOS, always the interval center is tested for a line-of-sight. Then, the interval is updated accordingly, i.e., the lower interval bound is set to the center if there is no LOS and otherwise the upper interval bound is set to the center.

The approach can be used for composed domains which occur often in practice. Then the domain is tiled into rectangular grids of different resolutions. Besides handling the local tile coordinate systems in a global coordinate system, all to be done is to segment the ray at the tile borders and perform queries with the local rays until the first intersection is found.

VI. Empirical Comparison

We have implemented the kd-tree data structure with line-of-sight computation and its compressed variant. In this section we give a systematic, empirical comparison of its performance in terms of data set sizes, search direction and sorting of ground stations by ground station height and by distance to the query origin. For this, we compute the minimum height with line-of-sight to one of a fixed number of ground stations for each point of the heightfield grid. Figure 6 shows the results with different

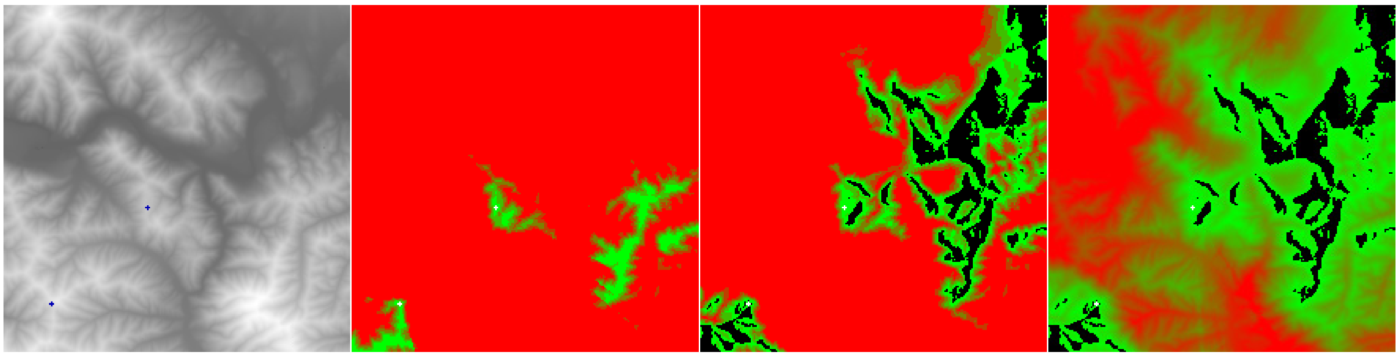


Fig. 6. Heightfield with line-of-sight information to two ground stations. From left to right: Point-reconstructed terrain, bilinear-reconstructed terrain, minimum height with line-of-sight. Color map shows the height above ground with full red at 243.9m to full green just above ground, and black areas have a line-of-sight at the ground height.

TABLE I.

Times for building the data structure in different resolutions.

	220*220	256*256	512*512	1024*1024	2048*2048
kd-tree, point	35.02ms	49.53ms	204.21ms	360.85ms	1441.14ms
kd-tree, bilinear	60.86ms	99.57ms	322.70ms	1218.63ms	5058.16ms
wavelet, point	7.34ms	7.47ms	131.20ms	553.43ms	2331.25ms
wavelet, bilinear	91.11ms	101.59ms	497.18ms	1962.52ms	8402.29ms

reconstructions of the terrain surface. *Point* reconstruction is the simplest and fastest method and also gives usually more conservative visibility results. *Bilinear* reconstruction uses a degree-two surface and has the property of being continuous, which is important for many simulation applications. With *bilinear-approximate* we denote the reconstruction which has the same border lines as the bilinear surface but connects also two arbitrary border points by a line. The resulting bundle of planes (containing the border point, the vertex point and one of the lines) is not a single surface as it does not have a unique tangential plane anymore.

Table II lists the computation times for different resolutions of the same heightfield data set. All timings were performed on a Windows XP system with Intel Pentium M 1.6GHz processor and 1.5GB RAM.[1] Point reconstruction is fastest as it needs to access only a single height value at the leaf level. For bilinear-approximate and bilinear reconstruction, we try to keep the four leaf nodes required for the reconstruction sequentially in memory. The resulting performance is roughly $3/4$ of that of point reconstruction. Especially notable is that the wavelet-like storage scheme is nearly as fast as the fully-stored kd-tree with inner nodes. For the more complex leaf code of bilinear reconstruction the difference is even smaller.

The tests concerning search direction in Table III show that the information where the intersection point lied in the last query at a different height or in a horizontal or vertical neighbor is most successful. Also this information is very easy to exploit. Note that it is not easy to predict if forward search or backward search has a shorter search length up to an intersection as it requires

[1] See the chart *Low End CPU's* for an assessment of this processor's speed http://www.cpubenchmark.net/low_end_cpus.html

TABLE II.

Timings of line-of-sight computation on the same heightfield in different resolutions (queries to two ground stations per heightfield cell).

	220*220	256*256	512*512	1024*1024	2048*2048
kd-tree, point	0.00321ms (311463q/s)	0.00324ms (308214q/s)	0.00391ms (255696q/s)	0.00381ms (262440q/s)	0.00408ms (245386q/s)
kd-tree, bilinear-approx	0.00548ms (182468q/s)	0.00539ms (185671q/s)	0.00583ms (171488q/s)	0.00642ms (155867q/s)	0.00744ms (134424q/s)
kd-tree, bilinear	0.00600ms (178676q/s)	0.00557ms (179615q/s)	0.00558ms (179332q/s)	0.00697ms (143456q/s)	0.00759ms (131769q/s)
wavelet, point	0.00335ms (298808q/s)	0.00345ms (289764q/s)	0.00391ms (255636q/s)	0.00446ms (224383q/s)	0.00421ms (237468q/s)
wavelet, bilinear-approx	0.00553ms (180600q/s)	0.00568ms (175947q/s)	0.00597ms (167430q/s)	0.00698ms (143273q/s)	0.008090ms (123544q/s)
wavelet, bilinear	0.00546ms (183242q/s)	0.00581ms (172219q/s)	0.00605ms (165277q/s)	0.00713ms (140322q/s)	0.00845ms (118410q/s)

TABLE III.

Timings of different search directions: forward, backward or based on which one was faster in last query (heightfield of resolution 220*220 with two ground stations).

	forward	backward	last-fastest
kd-tree, point	0.00325ms (308084q/s)	0.00316ms (316837q/s)	0.00315ms (317766q/s)
kd-tree, bilinear	0.00612ms (163273q/s)	0.00556ms (179889q/s)	0.00563ms (177758q/s)
wavelet, point	0.00356ms (281024q/s)	0.00344ms (291051q/s)	0.00341ms (293648q/s)
wavelet, bilinear	0.00662ms (151095q/s)	0.00563ms (177482q/s)	0.00546ms (183242q/s)

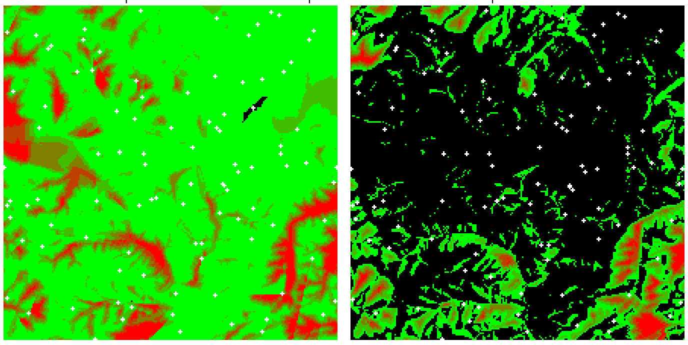
carrying out the search.

For querying a large number of ground stations, the question is what order of ground stations is fastest for finding a LOS to one of them. Table IV compares three different orders: random, sorted by decreasing height or sorted by increasing distance to the query origin. This experiment confirms the theoretical results. Sorting by increasing distance has the best worst-case runtime, and it is best if it is not dominated by the costs of sorting, which has to be done with each new query point. But here also coherency can help with regular sets of query points. The sorting by descreasing ground station height does not have

TABLE IV.

EFFECT OF DIFFERENT SORTING OF GROUND STATIONS: RANDOM, SORTED BY DECREASING HEIGHT OR SORTED BY INCREASING DISTANCE TO THE QUERY DESTINATION (HEIGHTFIELD OF RESOLUTION 220*220 WITH 100 GROUND STATIONS).

	random	sorted by height	sorted by distance
kd-tree, point	38.466s (8,953,563q)	48.676s (12,011,323q)	26.444s (6,005,989q)
kd-tree, bilinear	25.707s (4,468,750q)	29.942s (5,281,339q)	19.191s (3,382,248q)
wavelet, point	41.319s	52.053s	28.216s
wavelet, bilinear	27.061s	31.821s	20.000s



this cost per query point but its results heavily depend on the data set heights. In our experiments, it could not outperform the random order of ground stations. Notice that the point reconstruction performs 2–2.3 times more queries compared to the bilinear reconstruction to check that there is no LOS to any ground station at the current height.

VII. CONCLUSIONS

We proposed a 2.5d extension of the kd-tree data structure for line-of-sight computation on terrains. This data structure is classic for nearest neighbor searching in large point sets and for speeding up ray intersection with 3d triangle soups. With our extension, it shows good performance also for line-of-sight queries on terrain surfaces, where large low height regions can be processed with much fewer ray-terrain height tests. We did an empirical comparison with data sets of different resolutions and different numbers of ground stations. To save for the additional storage of the inner nodes of the binary tree, we used a wavelet-like compressed storage scheme which merges two levels of the kd-tree (into a 4-ary tree) and allows for reconstruction on-the-fly during a line-of-sight query. Despite the additional reconstruction operations, the overall runtime is nearly as good as the runtime for the fully-stored kd-tree with inner and leaf nodes. As future work, we think about lossy compression and decompression of the heightfield array and an analysis of the resulting errors in line-of-sight computations.

REFERENCES

[1] US Geological Survey, "National Elevation Dataset (NED, Resolution 1arcsec/0.33arcsec)," 2001, http://ned.usgs.gov/.

[2] US Geological Survey, "GTOPO30 Dataset (Resolution 30arcsec)," 1996, http://edc.usgs.gov/products/elevation/gtopo30/gtopo30.html.

[3] Boaz Ben-Moshe, Matthew J. Katz, Joseph S. B. Mitchell, and Yuval Nir, "Visibility preserving terrain simplification: an experimental study," *Comput. Geom. Theory Appl.*, vol. 28, no. 2-3, pp. 175–190, 2004.

[4] Dieter W. Fellner and Norbert Schenk, "MRT - A tool for simulations in 3d geometric domains," in *Proceedings of ESM*, Jun 1997, pp. 185–188.

[5] Arne Schmitz and Martin Wenig, "The effect of the radio wave propagation model in mobile ad hoc networks," in *MSWiM '06: Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems*, New York, NY, USA, 2006, pp. 61–67, ACM Press.

[6] Marco Allegretti, Marco Colaneri, Riccardo Notarpietro, Marco Gabella, and Giovanni Perona, "Simulation in urban environment of a 3d ray tracing propagation model based on building database preprocessing," in *Proceedings of URSI General Assembly*, 2005.

[7] Jon Louis Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[8] Michael D. Proctor and William J. Gerber, "Line-of-sight attributes for a generalized application program interface," *JDMS*, vol. 1, no. 1, pp. 43–57, Apr. 2004.

[9] Brian Salomon, Naga Govindaraju, Avneesh Sud, Russell Gayle, Ming Lin, and Dinesh Manocha, "Accelerating line of sight computation using graphics processing units," in *Proc. of I/ITSEC 2005*, 2005.

[10] David Tuft, Brian Salomon, Sean Hanlon, and Dinesh Manocha, "Fast line-of-sight computations in complex environments," Tech. Rep. TR05-025, 2005.

[11] Nelson L. Max, "Horizon mapping: shadows for bump-mapped surfaces," *The Visual Computer*, vol. 4, no. 2, pp. 109–117, Mar 1988.

[12] Holly Rushmeier, Laurent Balmelli, and Fausto Bernardini, "Horizon map capture," *Computer Graphics Forum*, vol. 20, no. 3, 2001, ISSN 1067-7055.

[13] James A. Stewart, "Fast Horizon Computation at All Points of a Terrain with Visibility and Shading Applications," *IEEE Trans. on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 82–93, Jan. 1998.

[14] David Kidner, Mark Dorey, and Derek Smith, "What's the point? Interpolation and extrapolation with a regular grid DEM," in *Proc. of GeoComputation 1999*, 1999.

[15] C. Christopoulos, A. Skodras, and T. Ebrahimi, "The JPEG2000 Still Image Coding System: An Overview," *IEEE Transactions on Consumer Electronics*, vol. 46, no. 4, pp. 1103–1127, Nov. 2000.

[16] Eric J. Stollnitz, Tony D. DeRose, and David H. Salesin, "Wavelets for Computer Graphics: A Primer, Part 1 & 2," *IEEE Computer Graphics and Applications*, vol. 15, no. 3, pp. 76–84, 1995.

[17] M. Akian, G. Cohen, S. Gaubert, R. Nikhoukhah, and J.P. Quadrat, "Linear systems in (max,+) algebra," *Proceedings of the 29th Conference on Decision and Control, Honolulu, Hawaii*, pp. 151–156, Dec 1990.

[18] Christoph Fuenfzig, Torsten Ullrich, and Dieter W. Fellner, "Hierarchical Spherical Distance Fields for Collision Detection," *IEEE Computer Graphics and Applications*, vol. 26, no. 1, pp. 64–76, 2006.