Variations on Sweep Algorithms:

efficient computation of extended viewsheds and class intervals

Marc van Kreveld

Dept. of Computer Science, Utrecht University
The Netherlands

E-mail: marc@cs.ruu.nl

Abstract

Two novel applications of the plane sweep paradigm are demonstrated, namely, for the computation of extended viewsheds on gridded DEMs and for class interval selection on TIN-based DEMs. In both cases, the efficiency of the plane sweep algorithm is significantly better than a straightforward approach. The algorithms are presented by first giving the plane sweep method as a general approach that requires some ingredients to make it work well. Adaptations to minimize additional storage use are also presented.

1 Introduction

One of the most important paradigms in the design of geometric algorithms is that of sweeping. In a plane sweep algorithm, an imaginary horizontal line traverses the plane from top to bottom, during which some property of the data is computed (a more clear description is given later). Plane sweep algorithms have been used for a variety of geometric problems like map overlay, Voronoi diagrams and hidden surface removal. The plane sweep approach is described in most textbooks on computational geometry [34, 37, 40]. Implementation of plane sweep algorithms usually is straightforward.

In computational geometry, the plane sweep approach has become a standard technique in the design of efficient algorithms. On the other hand, in GIS literature the plane sweep technique is known—in particular for map overlay [4, 28, 32]—but not yet a standard technique. Certainly, its use hasn't been recognized to its full extent. The purpose of this paper is to give two new applications of the plane sweep method, showing its importance and versatility once more. Both applications address a problem in the important GIS capability of geographical analysis.

The first plane sweep algorithm that will be described solves various problems in viewshed analysis, such as the computation of extended viewsheds [18, 19] and visibility indices

[23, 44]. Given a gridded DEM and a specific pixel on it, we're interested in information like the number of pixels that are visible from the specific pixel (the visibility index), the vertical distance to visibility for the non-visible pixels, and the vertical distance to the local or global horizons. A straightforward algorithm would do these computations on an $n \times n$ grid in $O(n^3)$ time. Our variant plane sweep algorithm requires only $O(n^2 \log n)$ time, saving an order of magnitude. The algorithm is based on a half-line rotating around the specified pixel. Rotating sweep algorithms have been described before, but not for viewshed computations, nor for elevation grids. The additional storage required by the algorithm is only $O(n)$. There have been many other papers dealing with various aspects of viewshed analysis [20, 21, 17, 30, 31].

The second application of plane sweep involves the computation of class intervals on DEMs. These are needed for displaying isarithmic maps with appropriate isolines. We now assume that the elevation model is a triangulated irregular network, or TIN. The idea applies to grids as well. Since an elevation model represents some surface, the distribution of the elevation values of that surface is modelled best by a density function (or, frequency distribution). Once the density function is computed, various class interval selection systems may be used, like percentile (quantile) classes, natural breaks, and bounded within-class variance. It is important that classes are chosen based on the whole elevation model, not the underlying data set of point samples, so that percentile classes ensure equal representation of each class on the isarithmic map. For a good discussion on class interval selection, see Evans [14]. Several textbooks also describe the choice of class intervals, sometimes called setting contour levels or indicator thresholds [7, 27, 46, 47].

The algorithm to compute the density function sweeps a horizontal plane vertically through the TIN. For a TIN with n triangles the algorithm requires $O(n \log n)$ time. The sweep algorithm is the first one—to our knowledge—to sweep a horizontal plane through a TIN. The extra storage required by the algorithm is $O(n)$ in the worst case, but in practice it can be reduced considerably.

Section 2 of this paper explains the basic ideas and applications of the plane sweep method. In Section 3 the viewshed algorithm is presented, in Section 4 the approach toward the selection of class intervals is given, and in Section 5 we show that storage requirements of both algorithms can be reduced. Conclusions are given in Section 6.

2 Sweep algorithms

Imagine a set S of objects in the plane about which something must be computed. Let's say a question Q needs to be addressed. Sometimes the following intuitive approach works: Take a horizontal line above all objects and sweep it downward. During the sweep, make sure that all information relevant to answer Q becomes known when the sweep line passes the objects that give the information. When the sweep line lies below all objects, we have gathered all relevant information and we can output or construct the answer.

A more explicit but still general description follows; see also Figures 1 and 2. During the sweep, we nearly always have to maintain the subset of objects that is intersected by

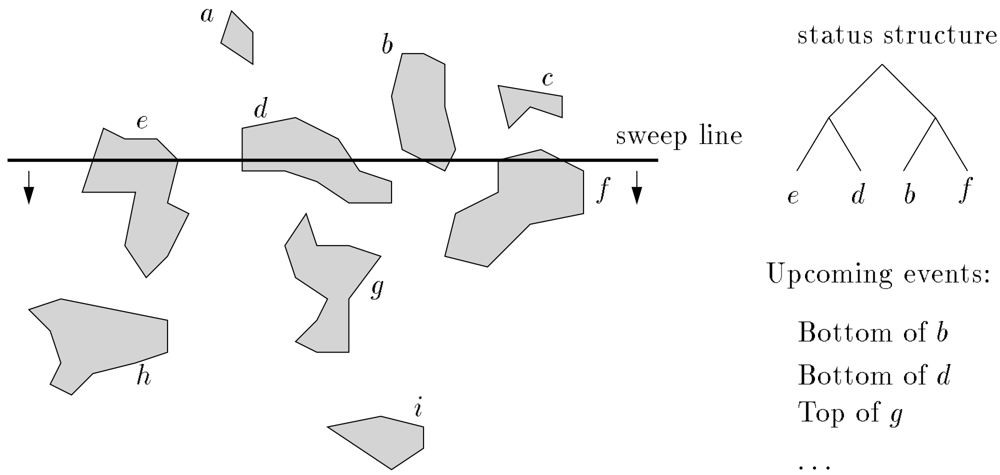

Figure 1: Sketch of the sweep algorithm.

the sweep line. Since this subset changes repeatedly, this subset must be maintained in a dynamic data structure. Often a balanced binary search tree suffices. This data structure is called the *status structure*.

We also have to know somehow when the status changes, and when we can find bits of information to answer the question Q. This happens at certain positions of the sweep line. These positions are called *events* and they are given by a y-coordinate since we assumed the sweep line to be horizontal. Since the sweep line may only move downward, we need to store these events sorted on y-coordinate. The data structure for this purpose is called the *event list*. Often it is simply a priority queue (or a balanced binary tree). The sweep line jumps from event to event by repeatedly removing the one with maximum y-coordinate, until all events have taken place.

There is one more ingredient to sweep line algorithms: the handling of events. This ingredient is the one that varies most from application to application. Two types of event are almost always present: when the sweep line starts intersecting an object of the set S, it must be inserted into the status structure, and when the sweep line has just passed the lowest point of an object of S, it must be removed again.

With plane sweep, a two-dimensional geometric problem is solved by using dynamic one-dimensional data structures. More extensive descriptions of plane sweep algorithms can be found in textbooks on computational geometry [34, 37, 40].

The idea can be generalized to dimension three, where a plane sweeps through space. The status structure now is a dynamic two-dimensional data structure and since these aren't that well-known, three-dimensional sweeping hasn't been applied as often as plane sweep.

Known applications of plane sweep and variants are computation of the Voronoi diagram [22], map overlay [4, 32], nearest objects [3, 25, 26], triangulation [29], hidden surface removal [33, 38], two- and three-dimensional point location [8, 41, 42], separation

```
1. Initialize the event list.
2. Initialize the status structure.
3. While the event list is not empty
4. Do Delete the event with maximum y-coordinate from the event list.
5.    If a new object is intersected by the sweep line
      then insert it in the status structure.
6.    If an object stops being intersected by the sweep line
      then delete it from the status structure.
7.    Address the question of interest using the status structure.
8.    If necessary, add new events to the event list.
9. Enddo
```

Figure 2: Structure of a sweep algorithm.

of point sets [15], rectangle intersection [11], shortest paths [36], median-of-squares statistics [13, 43], and many others that don't seem to have immediate applications in GIS. The Geometry Literature Database [24], a database with over 7000 papers related to computational geometry, lists about one hundred papers that employ plane sweep. In most of the two-dimensional applications listed above, the status structure is either a balanced binary tree or a segment tree. Both allow updates and queries to be performed in $O(\log n)$ if the data structure stores $O(n)$ objects. This often results in $O(n \log n)$ time algorithms, plus the time needed to report the answer. The more staightforward algorithms take quadratic time for these problems. For example, one can compute the Voronoi diagram of a set of n sites by computing all Voronoi cells (Thiessen polygons) separately, taking $O(n \log n)$ time per site. In total, this comes down to $O(n^2 \log n)$ time, whereas plane sweep only takes $O(n \log n)$ time.

Some variants of the standard sweep are versions where a line rotates about a point [9, 15, 35], and the three-dimensional version where a horizontal plane translates through space [39, 41]. There are also sweep algorithms where a "topological" line is used instead of a straight line [12].

Implementation of plane sweep algorithms is not difficult, certainly not if existing code for balanced search trees, sorting, and geometric primitives is used. A major advantage of sweeping over other methods like divide-and-conquer is that not all objects need be in main memory simultaneously. Objects are needed in the order of their y-coordinates, and only the objects that intersect the sweep line in its current position need be in main memory (in the status structure). There are simple ways to avoid storing the whole event list in main memory [1, 2, 28]. Divide-and-conquer algorithms usually require all objects to be in main memory at some moment, either during the first divide or during the last conquer step.

Of course, plane sweep is not the appropriate solution to all problems. For certain more difficult tasks, lack of a dynamic geometric data structure prohibits the use of efficient plane sweep. In other cases, plane sweep is just one of several candidates for a problem. Map overlay can also be solved using different techniques, like those based on R-trees [45, 5] or on geometric graph traversal [16].

Finally, there are cases where plane sweep is possible but not at all clever. To construct a minimum bounding box of a set of n objects one can use plane sweep and get an $O(n \log n)$ time solution. However, the straightforward inspection of all coordinate values of the objects is much simpler and takes only $O(n)$ time.

3 Viewshed computation

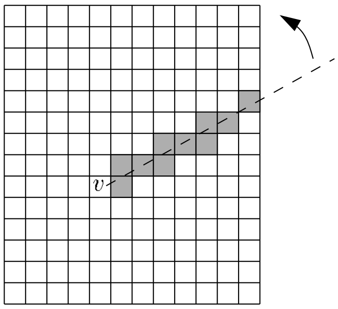

Figure 3: Rotating a half-line over the grid; the grey pixels are stored in order of intersection in the status structure.

This section presents efficient algorithms for viewshed computations on an $n \times n$ grid. We don't just consider standard viewsheds, but also local and global horizon offsets like described by Fisher [18, 19]. He notes the use of computing more than just a binary grid of visible/non-visible information, such as, what distance a grid pixel is below the local horizon that obscures it (the *local offset*). In other words, how high a stick must be placed so that it is visible from the viewpoint. Fisher studies the uncertainty aspects of viewshed computation rather than efficiency aspects. A standard algorithm for the visibility information would, however, require $O(n^3)$ time. We show that using sweep, $O(n^2 \log n)$ time suffices, improving the efficiency considerably. Our algorithm is more complex than the straightforward one, but still it is fairly simple to implement.

There are other methods to compute viewsheds in roughly quadratic time. These can perhaps be generalized to extended viewsheds as well, but they have other drawbacks. The ring growing method, described in several papers [23, 44], requires $O(n^2)$ time, but visibility between two pixels can be blocked by pixels that may be quite far from the line of sight. A highly undesirable situation. Another algorithm that requires quadratic time uses the line of sight from the viewpoint to all perimeter pixels, but not to others [23]. This method doesn't use center-of-pixel to center-of-pixel visibility except for the perimeter. Furthermore, it requires additional bookkeeping to determine for pixels which line(s) of sight determine the visibility. In some versions, a counter is needed with every pixel, thus requiring quadratic extra storage. Our algorithm, and also the partial blocking method of Teng and Davis [44], doesn't have these drawbacks.

Suppose a grid G with elevation data is given together with a viewpoint v, which is represented by the center of one of the pixels. The idea of our variant of plane sweep is to rotate a half-line about v a full turn of 2π radians, while computing the visibility or local offset of each pixel when the half-line passes over the pixel center. The status structure stores all pixels intersecting the sweep half-line in the leaves of a balanced binary search tree T, such that the leftmost leaf of T stores the pixel closest to the pixel with the viewpoint, see Figure 3. The tree T is augmented with one real number per leaf and internal node. With each leaf the gradient is stored of the line segment from the center of the pixel with v to the center of the pixel stored in the leaf (the gradient of the line-of-sight LoS). So if the pixel with v has elevation h_v, the pixel in a leaf l has elevation h_l, and the distance between the centers of the pixels is d, we store $\arctan((h_l - h_v)/d)$ in the leaf. In bottom up fashion we define the additional real number stored with internal nodes: it is the maximum of the real numbers stored with its two children. So for each node in T, we have stored the gradient of the "most obscuring" pixel in its subtree. One can perform insertions and deletions in $O(\log n)$ time in T, inclusive the updating of the additional real numbers. The method of augmenting data structures, that is, maintaining the additional information in the tree, is described in several textbooks on algorithms (e.g. [10]).

The event list consists of all angles at which the sweep half-line starts to intersect a pixel, at which the sweep half-line stops intersecting a pixel, and at which the sweep half-line goes through a pixel center. So for each pixel exactly three event angles are stored, and in total $O(n^2)$ events will take place. If two angles are the same we let the event of the pixel closer to the viewpoint have preference in the event order.

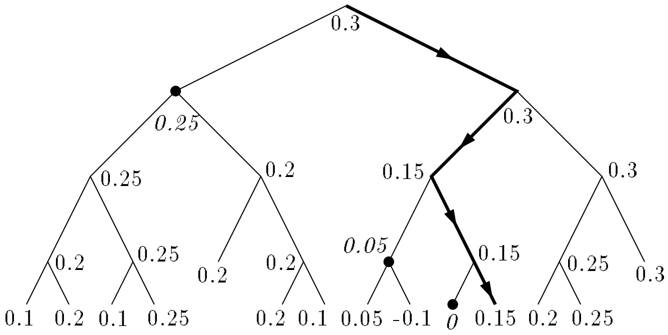

Figure 4: A search path in T.

The sweep algorithm is as follows. Initialize the event list by sorting all the event angles we referred to. Assume for instance we start with a rightward directed half-line (angle 0 with the positive x-axis) and we rotate counterclockwise. Repeatedly take an event—the first one—from the event list and depending on the type of event, insert a pixel in T, delete a pixel from T, or search in T. The latter is done when the event is caused by the center of a pixel, and the algorithm must establish whether it is visible (and if not, its local offset). The pixel is visible if and only if the LoS to it has greater gradient than of all pixels closer to the viewpoint, since only these may obstruct visibility. We follow the search path to the pixel whose center the sweep half-line has reached, and we consider the nodes in T nearest to the root that are left children of the nodes where the search path turned right. In Figure 4, a search path and the corresponding left children are indicated. Next we determine the maximum of the real numbers stored with these left children (0.25 in the figure). This maximum is the maximum gradient of the LoS to a pixel closer to the viewpoint. By comparing the maximum gradient to the gradient of the pixel itself we establish whether the pixel is visible. If not, we use the gradients and the distance of the pixel to the viewpoint to determine the local offset.

Observe that at most $2n$ pixels are stored at any time in T. Since T is balanced, its depth is $O(\log n)$. A search follows one path to a leaf, and the left children of this path. There can only be $O(\log n)$ left children on a path of length $O(\log n)$, and all further computations are trivial. So inserting, deleting, and searching only take $O(\log n)$ time per operation on an $n \times n$ grid. Since there are $O(n^2)$ events, and each event is handled in $O(\log n)$ time, we have given an $O(n^2 \log n)$ time algorithm. The initial sorting of events by angle also takes $O(n^2 \log n)$ time.

Note that in our model we have chosen to store the gradient to the center of the pixel in the leaves of the tree T. When the LoS passes over a pixel, it usually doesn't pass over the pixel center, which may influence visibility of the pixels beyond it slightly. It depends on the model we assume to hold for the grid entries. We omit this issue here; adaptations to the algorithm can be made to incorporate different grid interpretations.

The algorithm that was given determined visibility of every pixel from the viewpoint, and therefore it can be used to compute the visibility index as well. We simply record how many pixels were visible from v. It is easy to adapt the algorithm for global offsets; we won't discuss this any further. Similarly, one could compute the number of local horizons in each direction, or the number of visible local horizons, with a similar sweep. It comes down to augmenting the status structure T in a different manner.

4 Density function computation and class interval selection

It is well-known that a finite population of interval data can be described by a histogram. For continuous interval data, the density function—or frequency distribution—is the corresponding descriptive statistic. It shows how frequent each elevation occurs in the data. The density function plays an important role in the choice of class intervals. This section studies the computation of the density function of a TIN that models a continuous univariate variable. Similar ideas can be used for some other elevation models, though. Note that it is more appropriate to compute class intervals based on the density function than on the finite set of sampled locations that has been the basis of the data. The presense of auto-correlation in the data will cause that class intervals based on the sample will not classify in a fair way, unless random sampling was used. By interpolating between the data points this problem is overcome (see e.g. [14, 27]). A triangulation is one example of an interpolation method.

We begin with a useful observation and a straightforward algorithm. Consider just one triangle Δ in 3-space with vertices u, v, w. Assume for simplicity that $h(u) > h(v) > h(w)$, where $h(..)$ denotes the elevation of a vertex. Then the density on the triangle for a given elevation t is $l \cdot \cos(\alpha)$, where l is the length of the intersection of the triangle Δ with the plane $z = t$, and α is the angle between the normal of Δ and any horizontal plane. The density is zero for all elevations t with $t > h(u)$ or $t < h(w)$. It is given by a function f_{uv} depending linearly on t if $h(u) > t > h(v)$, and it is given by a different function f_{vw} depending linearly on t if $h(v) > t > h(w)$. So we have $f_{uv}(t) = a \cdot t + b$, where a and b depend only on the coordinates of u, v, w and thus are fixed. The same holds for f_{vw}, but with different a and b.

For simplicity of exposition we assume that all vertices have different elevations. This restriction can be overcome without problems, but some care must be taken. Let $v_1, \ldots, v_n$ be the vertices of the TIN, and assume that they are sorted on decreasing elevation. This

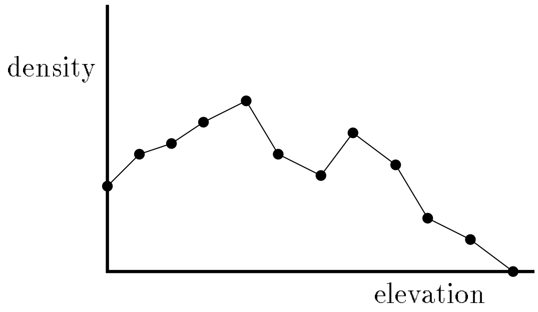

Figure 5: Density function of a TIN.

holds without loss of generality because we can simply relabel the vertices to enforce $h(v_1) > h(v_2) > \cdots > h(v_n)$. Consider the density for an elevation t, where $t \in (h(v_j),\, h(v_{j+1}))$. In such an open interval, the density is the sum of a set of linear functions, which is again a function linear in t. We denote the linear function that gives the density over the whole TIN in the interval $(h(v_j),\, h(v_{j+1}))$ by $F_j(t)$. So the linear functions $F_0,\, F_1, \ldots,\, F_n$ form the density function, where each function is only valid in its interval. By default we set $F_0(t) = 0$ and $F_n(t) = 0$ for the intervals $(h(v_1), \infty)$ and $(-\infty, h(v_n))$, because for these elevations the density is zero. We observe:

Proposition 1 *The density function based on a TIN with n vertices is a piecewise linear continuous function with at most $n+1$ pieces.*

The density function need not be continuous when there are vertices with the same elevation. The straighforward algorithm to construct the density function on the TIN is the following: Sort the vertices by elevation, and for each interval $(h(v_j),\, h(v_{j+1}))$, determine the set of linear functions contributing to it. Then add up these linear functions to get one linear piece F_j of the density function. Since we have $O(n)$ vertices, we have $O(n)$ intervals and for each we can easily determine in $O(n)$ time which linear functions contribute. The total time taken by this algorithm is $O(n^2)$.

The efficient computation of the density function is again based on the sweeping approach. We will exploit the fact that the linear function F_j can be obtained easily from the linear function F_{j-1} since the contributing f are for the larger part the same ones. We compute the summed linear functions F from top to bottom, which comes down to a sweep with a horizontal plane through the TIN. Throughout the sweep we maintain the density function of the current elevation. Using sweeping terminology, every vertex of the TIN gives rise to one event. The event list is a priority queue storing all these $O(n)$ events in order of decreasing elevation. With each event we store a pointer to the vertex of the TIN that will cause the event. The status structure is trivial: it is simply the summed linear function F for the current position of the sweep plane.

The final ingredient to the sweep algorithm is handling the events. When considering how F_{j-1} should be changed to get F_j when the sweep plane passes the vertex with elevation

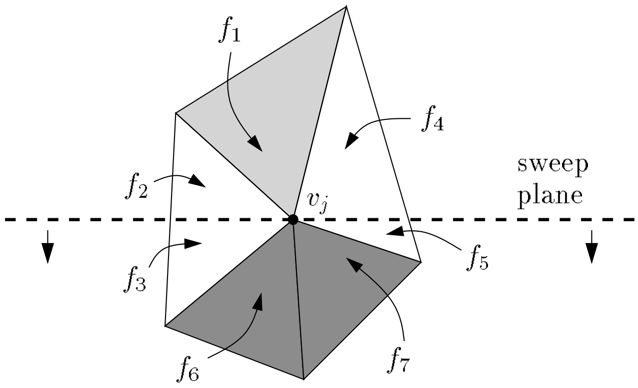

Figure 6: Passing a vertex with the sweep plane.

$h(v_j)$, we must examine how the density function changes. The vertex v_j is incident to some triangles, for which it can be the highest vertex, the lowest vertex or the vertex with middle elevation. We update F_{j-1} to get F_j according to the following rules:

- For all triangles for which v_j is the lowest vertex (lightly shaded in Figure 6), we subtract from F the appropriate linear function (f_1 in Figure 6).
- For all triangles for which v_j is the highest vertex (darkly shaded), we add to F the appropriate linear function (f_6 and f_7).
- For all triangles for which v_j is the middle vertex (white in the figure), we subtract the one linear function (f_2 and f_4) and add the other (f_3 and f_5).

We don't need to precompute or store the linear functions f on each triangle to update F; the f can be obtained from the coordinates of the vertices on the TIN when the event at vertex v_j is handled. We have fast access to vertex v_j in the TIN; recall that an extra pointer was stored in the event list.

We also evaluate the function F_j at the event. The sequence of evaluations gives the breakpoints of the (piecewise linear) density function. These breakpoints are computed from right to left in Figure 5 since the sweep goes from high to low elevations.

Considering the efficiency of the algorithm, the initial sorting of the events takes $O(n \log n)$ time for a TIN with n vertices. Extraction of an event takes $O(\log n)$ time; for all events this adds up to $O(n \log n)$ time. Updating the status structure at an event v_j requires time linear in the number of triangles incident to v_j. Summed over all vertices this is linear in n by Euler's formula. The evaluation to determine the breakpoints requires constant time per event. So in total the sweep algorithm requires $O(n \log n)$ time.

Once the density function is computed, the class intervals may be determined. Suppose as an example that the objective is to determine seven classes such that each class occupies an equivalent amount of area on an isoline map. We assume that the isoline map and the TIN have the same domain, otherwise we can clip the TIN with the domain of the isoline map before doing the sweep. The total area of the isoline map is the same as the total area under the density function and is denoted A. The area under the density function in the elevation interval $[a, b]$ is denoted $A(a, b)$. If $F(t)$ denotes the (piecewise linear) density function, then

$$A(a, b) = \int_a^b F(t)dt$$

The value of $A(a, b)$ is exactly the area for the class $[a, b]$ on the isoline map. We know the total area A and compute $A/7$, the desired area for each class. We then determine the lowest elevation such that $A/7$ of the area is below that elevation. This operation is easy by scanning over the known density function $F(t)$ from left to right and maintaining the area under $F(t)$ (this is also a kind of sweep). This gives the lowest class boundary. Continuing the scan gives all six boundaries of the seven classes in $O(n)$ time. In a similar way one can compute a non-fixed number of classes with the property that the within-class variance is less than or equal to a certain threshold, for each class. Finally, the density function can be used class interval selection by natural breaks in the data: They are the local minima of $F(t)$. We refer to Burrough [7] and Evans [14] for other classification schemes.

5 Reducing the storage requirements

In both of the given sweep algorithms, the event list requires far more storage than the status structure. For the viewshed computations, the status structure required $O(n)$ storage and the event list $O(n^2)$ storage, since we stored all upcoming events in the event list before the sweep started. Similarly, for the density function computation, the status structure required $O(1)$ storage and the event list $O(n)$ storage.

In both algorithms we can bring down the storgae requirements of the event list using the same idea. Observe that for a sweep algorithm to work well, we need not have all upcoming events in the event list initially. We only need to be sure that if some event is the next one to be handled (reached by the sweep), then it must be in the event list. The underlying idea is the same as Brown's improvement in storage of the well-known Bentley-Ottmann sweep algorithm [6].

Consider the viewshed algorithm of Section 3. Instead of storing all $O(n^2)$ events, we will only store the events caused by pixels that are not yet encountered, but which have a neighboring pixel that has been encountered. It is easy to see that there are only $O(n)$ of these pixels, and that every pixel is present in the event list before it causes an event. The algorithm has to be adapted slightly: when the event to be handled next is an insertion of a pixel to the staus structure, then we must also insert its neighbors in the event list (unless they were already present, or have already been passed by the sweep).

Consider the density function computation of Section 4. The storage needed by the algorithm is $O(n)$ for the event list, but this can be reduced using a simple observation. Every vertex except the local maxima (peaks) have a higher neighbor in the TIN. So we can initialize the event list with the local maxima only. When the event at a vertex v is handled, we insert all lower neighbors of v in the event list. This guarantees that every event is present in the event list when the sweep plane reaches it. The storage required by the algorithm is equal to the sum of the number of local maxima and the number of edges in the largest complexity cross-section.

6 Conclusions

This paper presented two new sweep algorithms that are useful in GIS applications. We showed that extended viewsheds can be computed in $O(n^2 \log n)$ time on an $n \times n$ grid of elevation data, and that the density function of an elevation model based on a TIN with n vertices can be determined in $O(n \log n)$ time. The density function is the basis of various class interval systems other than the equal-class-width classes. Both algorithms are simple to implement and require only little extra storage. In both cases we reduced the computation time by an order of magnitude, when compared to the more straightforward algorithms. This demonstrates the use and relevance of one of the most important geometric algorithms design methods, namely, that of plane sweep. This type of algorithm has much more promise in geographic information systems than is recognized to date.

References

[1] L. Arge. External-storage data structures for plane-sweep algorithms. Technical Report RS-94-16, BRICS, Aarhus Univ., Denmark, 1994.

[2] L. Arge. The Buffer Tree: A new technique for optimal I/O-algorithms. In *Proc. WADS*, number 955 in Lect. Notes in Comp. Science, pages 334–345, 1995.

[3] F. Bartling and K. Hinrichs. A plane-sweep algorithm for finding a closest pair among convex planar objects. In *Proc. 9th Sympos. Theoret. Aspects Comput. Sci.*, volume 577 of *Lecture Notes in Computer Science*, pages 221–232. Springer-Verlag, 1992.

[4] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28:643–647, 1979.

[5] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. ACM SIGMOD*, pages 237–246, 1993.

[6] K. Q. Brown. Comments on "Algorithms for reporting and counting geometric intersections". *IEEE Trans. Comput.*, C-30:147–148, 1981.

[7] P. A. Burrough. *Principles of Geographical Information Systems for Land Resourses Assessment.* Oxford University Press, New York, 1986.

[8] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7:202–220, 1986.

[9] R. Cole, M. Sharir, and C. K. Yap. On k-hulls and related problems. *SIAM J. Comput.*, 16:61–77, 1987.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[11] H. Edelsbrunner. A new approach to rectangle intersections, Part II. *Internat. J. Comput. Math.*, 13:221–229, 1983.

[12] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. *J. Comput. Syst. Sci.*, 38:165–194, 1989. Corrigendum in 42 (1991), 249–251.

[13] H. Edelsbrunner and D. L. Souvaine. Computing median-of-squares regression lines and guided topological sweep. *J. Amer. Statist. Assoc.*, 85:115–119, 1990.

[14] I. S. Evans. The selection of class intervals. *Trans. Inst. Br. Geogrs.*, 2:98–124, 1977.

[15] H. Everett, J.-M. Robert, and M. van Kreveld. An optimal algorithm for the ($\leq k$)-levels, with applications to separation and transversal problems. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 38–46, 1993.

[16] U. Finke and K. H. Hinrichs. The quad view data structure – a representation for planar subdivisions. In *Advances in Spatial Databases (proc. SSD'95)*, number 951 in Lect. Notes in Comp. Science, pages 29–46, 1995.

[17] P. F. Fisher. Algorithm and implementation uncertainty in viewshed analysis. *Int. J. of GIS*, 7:331–347, 1993.

[18] P. F. Fisher. Stretching the viewshed. In *Proc. 6th Int. Symp. on Spatial Data Handling*, pages 725–738, 1994.

[19] P. F. Fisher. Reconsideration of the viewshed function in terrain modelling. *Geographical Systems*, 3:33–58, 1996.

[20] L. De Floriani, B. Falcidieno, C. Pienovi, D. Allen, and G. Nagy. A visibility-based model for terrain features. In *Proc. 2nd Int. Symp. on Spatial Data Handling*, pages 235–250, 1986.

[21] L. De Floriani, G. Nagy, and E. Puppo. Computing a line-of-sight network on a terrain model. In *Proc. 5th Int. Symp. on Spatial Data Handling*, pages 672–681, 1992.

[22] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[23] Wm Randolph Franklin and C. K. Ray. Higher isn't necessarily better: Visibility algorithms and experiments. In *Proc. 6th Int. Symp. on Spatial Data Handling*, pages 751–763, 1994.

[24] Geometry Literature Database. http://www.cs.ruu.nl/people/otfried/html/geombib.html.

[25] Thorsten Graf and Klaus Hinrichs. A plane-sweep algorithm for the all-nearest-neighbors problem for a set of convex planar objects. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 349–360, 1993.

[26] K. Hinrichs, J. Nievergelt, and P. Schorn. A sweep algorithm for the all-nearest-neighbours problem. In *Computational Geometry and its Applications*, volume 333 of *Lecture Notes in Computer Science*, pages 43–54. Springer-Verlag, 1988.

[27] E. H. Isaaks and R. M Srivastava. *An Introduction to Applied Geostatistics.* Oxford University Press, New York, 1989.

[28] H.-P. Kriegel, T. Brinkhoff, and R. Schneider. The combination of spatial access methods and computational geometry in geographic database systems. In *Advances in Spatial Databases (proc. SSD'91)*, number 525 in Lect. Notes in Comp. Science, pages 5–21, 1991.

[29] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[30] J. Lee. Analyses of visibility sites on topographic surfaces. *Int. J. of GIS*, 5:413–430, 1991.

[31] P. Magillo and L. De Floriani. Computing visibility maps on hierarchical terrain models. In *Proc. 2nd ACM Workshop on Advances in GIS*, page ??, 1994.

[32] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F40 of *NATO ASI*, pages 307–325. Springer-Verlag, Berlin, West Germany, 1988.

[33] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[34] K. Mehlhorn. *Multi-dimensional Searching and Computational Geometry*, volume 3 of *Data Structures and Algorithms.* Springer-Verlag, Heidelberg, West Germany, 1984.

[35] A. Mirante and N. Weingarten. The radial sweep algorithm for constructing triangulated irreguler networks. *IEEE Comput. Graph. Appl.*, pages 11–21, May 1982.

[36] J. S. B. Mitchell. L_1 shortest paths among polygonal obstacles in the plane. *Algorithmica*, 8:55–88, 1992.

[37] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, New York, 1994.

[38] M. Overmars and M. Sharir. A simple output-sensitive algorithm for hidden surface removal. *ACM Trans. Graph.*, 11:1–11, 1992.

[39] M. H. Overmars and C.-K. Yap. New upper bounds in Klee's measure problem. *SIAM J. Comput.*, 20:1034–1045, 1991.

[40] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[41] F. P. Preparata and R. Tamassia. Efficient point location in a convex spatial cell-complex. *SIAM J. Comput.*, 21:267–280, 1992.

[42] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[43] D. L. Souvaine and J. M. Steele. Time- and space- efficient algorithms for least median of squares regression. *J. Amer. Statist. Assoc.*, 82:794–801, 1987.

[44] Y. A. Teng and L. S. Davies. Visibility analysis on digital terrain models and its parallel implementation. Technical Report CAR-TR-625, Center for Automation Research, University of Maryland, 1992.

[45] P. van Oosterom. An R-tree based map-overlay algorithm. In *Proc. EGIS'94*, pages 318–327, 1994.

[46] David F. Watson. *Contouring: A Guide to the Analysis and Display of Spatial Data.* Pergamon, 1992.

[47] R. Webster and M. A. Oliver. *Statistical Methods in Soil and Land Resource Survey.* Oxford University Press, New York, 1990.