An Efficient and Scalable Approach to Visualizing Sensor Networks

Dina Q. Goldin, Huayan Gao
Department of Computer Science, University of Connecticut
dqg@cse.uconn.edu, ghy@cse.uconn.edu

May 2004

Abstract

Sensor networks offer the potential to provide timely information in an untethered and unattended real-world domain. Dynamic (real-time) visualization of sensor network data is needed in order to fully harvest that potential, allowing the users to see the patterns and trends – both spatial and temporal – inherent in the sensor data. Such visualization is the goal of this paper.

Our approach to visualization is based on *sensor terrains*, triangulated irregular networks where the (x, y) coordinates of the vertices corresponds to sensor locations, and the z coordinate corresponds to sensor readings. We dynamically maintain such a sensor terrain for our sensor network. Furthermore, we dynamically maintain an isoline (contour) map over this sensor network. The user has the option of continuously viewing either the current shaded triangulation, or the current isoline map, or an overlay of both.

For large sensor networks, we assume that complete recomputation of either the sensor terrain or the isoline map at every epoch is impractical. If n is the number of sensors in the network, time complexity show be $O(\log n)$ to achieve real-time performance. To achieve this time complexity, our algorithms are based on efficient dynamic data structures that are continuously updated rather than recomputed. Specifically, we use a doubly-balanced interval tree, where both the tree and the edge sets of each node are balanced. Experimental results confirm both the efficiency and the scalability of our approach.

1 Introduction

Sensors have been receiving a lot of attention because of the advantage they offer in providing timely information in an untethered and unattended environment. *Sensor networks* can be deployed for monitoring tasks in a wild range of such environments, from monitoring buildings for heating and contaminant problems, to monitoring forests for fires, to monitoring underground streams for pollutants [EGH99, ASSC02b, ASSC02a, TM03].

Sensor network queries offer a database-inspired approach to sensor data integration and processing for the new generation of sensor network applications. For many applications, the ability to query sensor networks in an *ad hoc* fashion will be key to their usefulness. Rather than re-engineering the network for every task, as is commonly done now, *ad hoc* querying allows the same network to process any of a broad class of *queries*, by expressing these queries in a common query language.

The goal of *ad hoc* querying is to allow the user to interact with the sensor network in real time, as if it were a single embedded distributed agent, and to perform on-the-fly data integration to understand this agent's various observations of its environment. Besides real-time response, desirable properties for query mechanisms over sensor networks include *robustness* and *transparency* to the users. By *robustness* we mean the ability of the networked system to operate in the presence of failures, while preserving its efficiency to the extent possible given the available resources. *Transparency* is achieved by giving the user a single point of access to the network (the server), and by focusing on query primitives that are independent of the exact number and routing configuration of the sensors in the network. Sensor networks with these properties exemplify the pervasive computing paradigm that is expected to become central this century.

Dynamic (real-time) visualization of sensor network data is needed in order to fully harvest their potential. It allows the users to see the patterns and trends – both spatial and temporal – inherent in the sensor data being output by the system in response to the query. Such visualization is the goal of this paper.

Our approach to visualization is based on *sensor terrains*. They are *triangulated irregular networks* (TINS), where the (x, y) coordinates of the vertices corresponds to sensor locations, and the z coordinate corresponds to sensor readings. We dynamically maintain such a sensor terrain for our sensor network, continually *updating* rather than *recomputing* it after each change. Possible changes include new values for sensor readings, new sensors joining the network, or sensors leaving the network (by either physical removal or loss of power).

TINs are three dimensional. Efficient algorithms, especially when implemented in hardware, allow for fast shading of TINs. By combining shading with user-driven rotation and zooming, sensor terrains provide a very user-friendly way to visualize sensor networks.

Furthermore, we dynamically maintain an *isoline (contour) map* over this sensor terrain. Isolines consist of points of equal value; they are commonly used to map mountanous geography. Various approaches to computing dynamic isoline maps over sensor networks have been investigated before [Est03, HHMS03]. However, to our knowledge, no one has described an algorithm that uses sensor terrains as a basis for dynamic isoline maps. The isoline map can be displayed in isolation, or in conjunction with the underlying TIN, provided the user with a visualization that is both highly descriptive and very intuitive.

To achieve real-time performance, efficiency is crucial. If n is the number of sensors in the network, time complexity should be $O(\log n)$. To achieve this time complexity, our algorithms are based on efficient dynamic data structures that are continuously updated rather than recomputed. Specifically, we use a doubly-balanced interval tree, where both the tree and the edge sets of each node are balanced.

We have implemented the data structures and algorithms proposed in the paper. The user has the option of continuously viewing either the current shaded triangulation, or the current isoline map, or an overlay of both. Experimental results, and simulating a large network of randomly distributed sensors, confirm both the efficiency and the scalability of our approach.

Outline. We describe sensor terrains in section 2, and discuss the algorithms for their computation and dynamic maintenance. In section 3, we give an algorithm for computing isoline maps over the sensor terrain, as well as their dynamic maintenance. In section 4 we present our implementation of sensor network visualization. Related work is discussed in section 5, and we conclude our paper in section 6.

2 Sensor Terrains

There are two main approaches to represent terrains in GIS. One is *Digital Elevation Models*(DEM), represending it as gridded data in some predefined intervals, which is volumn-based and regular.

DEMs are usually used in raster surface models. Due to the regularity of DEMs, they are not appropriate for sensor networks where the sensor positions are not very regular. The other is *Triangulated Irregular Networks*(TIN). The vertices of a TIN, sometimes called *sites*, are distributed irregularly and stored with their location (x, y) as well as their height value z as a vector data (x, y, z); TIN is usually used in vector data models. For a detailed survey of terrain algorithms, including TINs, see [vK97].

In this paper, we use a TIN to represent the state of a sensor network; the (x, y) coordinates of the TIN's vertices corresponds to sensor locations, and the z coordinates corresponds to current sensor readings. We refer to this representation as *sensor terrains*, a term first introduced for this purpose in [GSK+03]. Efficient dynamic algorithms for shading TINs provide a ready way to visualize sensor terrains.

We construct the sensor terrain with the following incremental algorithm. This is the typical algorithm for TIN construction, as in [GH95, GS85].

1. (Data Structure) Our data structure for sensor terrains consists of points, edges, triangles. Triangles are composed of edges; each edge has both end points. For each point, we have an edge list which stores all edges adjacent to the point. As a result, when one sensor changes its reading, we can get its adjacent edges quickly.

2. We begin with a square which is separated into two triangles, and then randomly add one sensor into the square, find the triangle enclosing this sensor, and split this triangle into three smaller triangles. The triangle is usually found by *random walking* citeex11, that is, locate a triangle randomly, then walk along the neighbors of the triangle towards the targeted triangle according to the location of the sensor point until we get the targeted triangle.

3. All triangles in the TIN must be *Delaunay triangles*. We use the *InCircle test* [GS85] to determine this; a triangle is Delaunay if the InCircle test is *not* TRUE for any of its edges. Given any edge (A, B) that is part of two trianges ABC and ABD, the InCircle test for (A, B) involves C and D as well. InCircle(A, B, C, D) is TRUE if one of the following holds:

   D is inside the circle incident on the triangle ABC, or
   A is outside the circle incident on the triangle DCB.

   This is illustrated in figure 1. In this case, the fix is to replace the triangles ABC and ABD by triangles ACD and BCD. Note that these new triangles must also be tested for being *Delaunay*.

4. Recursively execute step 2, untill all the sensors are added to the TIN.

During the construction, we maintain an edge list for each point so that we can track those edges in constant time whenever we change one sensor point randomly.

Since the construction of TIN is only dependent on the *location* of sensors, the topology of the TIN does not change with the change of the sensor readings. The only possibility for a TIN to change is when some sensor is added to the sensor network, or some sensors may leave the network, e.g. due to power loss. In the following, we describe the algorithms updating the TIN when a sensor is inserted or deleted.

Insertion. When a new sensor is added to the sensor network, we need to add the corresponding vertex to the sensor terrain. It would be the same algorithm as for building a new sensor terrain, since it is an incremental algorithm.

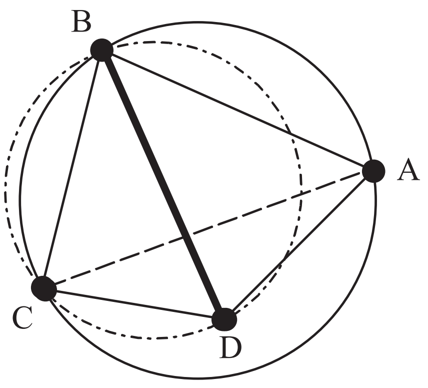

Figure 1: The InCircle Test

Deletion. When a sensor leaves the network, we need a local updating algorithm to maintain our dynamic TIN. Basically, this is the inverse of the incremental insertion algorithm, but in practice there are a variety of specific difficulties. [Hel90] first described a delete algorithm in detail, but unfortunately, it is false. The algorithm was corrected in [Dev99], and further improved in [MGD03]; it can be described briefly as follows.

1. Let P be the sensor point we are going to remove. First, identify the polygon enclosing P; its vertices are precisely those adjacent to P, found by considering the list of P's incident edges. Let this polygon be $v_0, v_1 \ldots v_k$, where the vertices are listed counterclockwise, and $v_k = v_0$.

2. Find the *ears* of the polygon. Any three continuous vertices of the polygon v_i, v_{i+1}, v_{i+2} are called an *ear* if: (a) the segment v_i, v_{i+2} is inside the polygon, and (b) there are no other points in the circles passing through (v_i, v_{i+1}, v_{i+2}), and through (v_i, v_{i+2}, P).

3. For each ear v_i, v_{i+1}, v_{i+2} perform InCircleTest on the quadrangle $(v_i, v_{i+1}, v_{i+2}, P)$ and swap the diagonal if it is true. Thus the polygon enclosing the point P will be changed correspondingly.

4. Repeat step 2 and step 3 until the polygon becomes a triangle; P will be located in this triangle. Now we can safely remove P.

In [MGD03], it is shown that the time complexity of deletion is $O(klogk)$ where k is the number of edges of the polygon enclosing P.

In figure 2 we can see the effect of this procedure when one sensor is removed from TIN.

TINs are three dimensional. Efficient algorithms, especially when implemented in hardware, allow for fast shading of TINs. By combining shading with user-driven rotation and zooming, sensor terrains provide a very user-friendly way to visualize sensor networks. Our implementation of the visualization of sensor terrain uses *OPENGL* [SWND03], a software interface to graphics hardware.

3 Dynamic Interval Tree

In this section, we describe how to extract isoline maps over sensor terrain from the dynamic sensor networks.

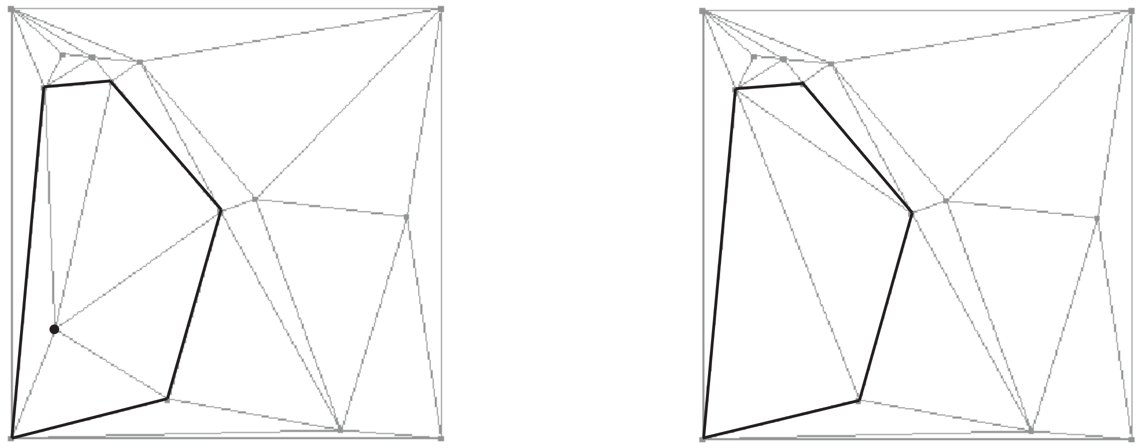

Figure 2: Deleting one sensor point from TIN: (a)before deletion; (b)after deletion.

3.1 Interval tree

One naive way to extract an isoline maps at a given height h is to traverse all the triangles in the sensor terrain, intersect each one with the plane $z = h$, and return all the resulting segments. An $O(n)$ time is needed for this brute-force approach, where n is the number of sensors.

We obtain a more efficient solution by using *interval trees* [Ede80]. These are special binary trees; besides the *split value*, each node also contains an *interval list*. For every edge in the TIN, it contains an interval corresponding to the edge's z-span. Given a set of such intervals, an interval tree is constructed as follows.

1. First, choose a split value s for the root. This value is determined by the first inserted interval. For example, if (a, b) is the first inserted interval, then the split value will be $(a + b)/2$.

2. Use s to partition the intervals into three subsets, I_{left}, I, I_{right}. Any interval (a, b) is in I if $a \leq s \leq b$; it is in I_{left} if $b < s$; and it is in I_{right} if $a > s$.

3. The intervals in I are stored at the root; they are organized into two sorted lists. One is *left list*, sorted in increasing order of a; the other is *right list*, sorted by decreasing order of b.

4. We recurse on I_{left} and I_{right}, creating a left and a right subtree, respectively.

Note that we use a *doubly-balanced AVL tree* for this algorithm: one is for the interval tree, the other is in the nodes of the tree, to store the edge lists. The AVL tree will enable us to provide quick updates to the tree (section 3).

Figure 3 gives an example of a TIN and the corresponding interval tree. The TIN is composed by 7 sensors and 19 edges. An interval tree is built from the TIN. More details can be found in [vK94] that uses an interval tree as the data structure to conveniently retrieve isoline segments. But they only considered static TIN, while we will give an algorithm for dynamic TIN in section 3.

An interval tree can also be constructed for *triangles*, rather than *edges*, of the TIN, since a z-span can just as easily be defined for triangles. The triangle-based interval tree allows for a more efficient algorithm to get the isolines from TIN than the naive one above. We can quickly find those triangles that intersect with the plane $z = h$, and avoid considering others. One such algorithm is given in [vK94].

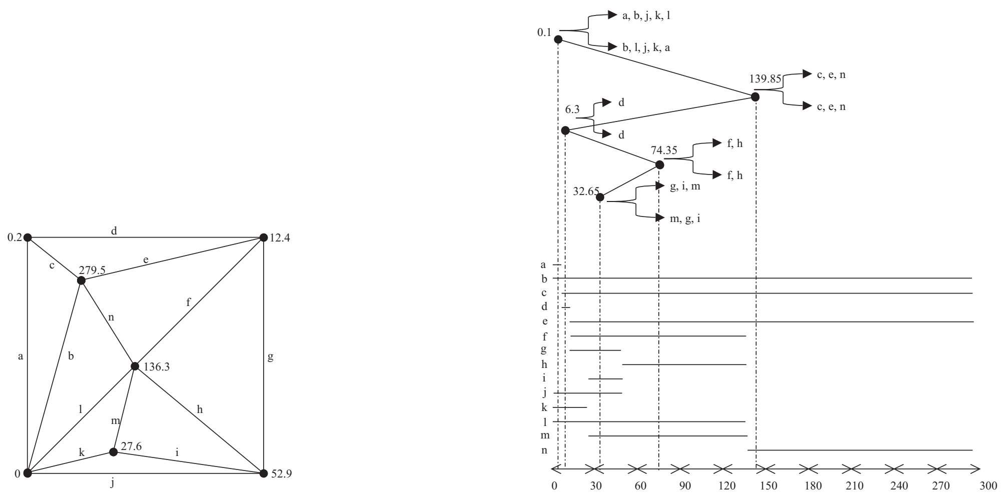

Figure 3: An example of a TIN and the corresponding interval tree.

In our paper, we found it more convenient to use edges to compute isolines from the TIN instead of triangles. Note that this kind of substitution does not affect the efficiency, because of the following fact: if there are n vertices in the TIN, the number of edges and the number of triangles are each $O(n)$ [Law77]. Let n_b denote the number of sensors on the boundary of the TIN, and n_i be the number of sensors in the interior so that the total number of sensors

$$n = n_b + n_i$$

The number of edges is

$$n_e = 2n_b + 3(n_i - 1) <= 3n$$

The number of triangles, let be n_t, would be

$$n_t = 2n - 6$$

when $n > 3$.

The algorithm for querying the isoline value v from the interval tree T is as follows. It is a recursive algorithm that begins with the root node. Let the split value be rs. If $v < rs$ then we will do the following two things. First search the left list of the root, and then search the left subtree of the root recursively. If $v > rs$ then we will do the similar two things. First search the right list of the root, and then search the right subtree of the root recursively. We stopp at the leaf node.

How do we query the matched edge list in the left list or right list of the interval node? Recalled that we stored the edge list as an AVL tree. Take the left list as an example, we only consider the left evaluation of the smaller point of the edge. Let it be the key k of the AVL node. We search the AVL tree recursively. If $v < k$, then we search the left subtree recursively. If $v > k$, which means that all the edges in the left subtree are the matched edges, output them and search the right subtree recursively. It is the same with querying the right list of the interval node.

3.2 Dynamic Interval Tree

In our setting, the sensor reading may change as time passes. Since our interval tree is built up on the edges of the TIN, and the z-span of the edge is dependent on the reading of the sensors adjacent to the edge, a change in sensor reading will necessitate a change to the interval tree.

We begin with a built TIN and an constructed interval tree. It is an AVL tree, storing the height for each of its nodes. In the following, we give a detailed description of the algorithm to update the tree after some sensor s changes its value from v_0 to v.

1. We use the TIN to find all edges incident with the sensor s. Since the TIN contains an incidence list L for each vertex, we can find these edges in constant time $O(1)$.

2. For every edge e in the list L, we need to update its position in the interval tree. Suppose that the original z-span for e is z_e, and the new one is z'_e.

3. Run a binary search from the root to find the node x which contains the interval z_e. This is done in $O(logn)$ time.

4. Delete z_e from both the right and the left lists of x. Since both of these lists are implemented as a standard AVL (Adelson-Velsky and Landis) tree, the performance is $O(logn)$.

5. Look for the node y that should contain z'_e, that is, its split value overlaps z'_e. First, check whether z'_e overlaps with the split value of x, in which case we need look no further. Otherwise, begin searching from the root of the tree, comparing z'_e with the split value of the node, until we find the node whose split value overlaps z'_e or reach a leaf. The time for this is in $O(logn)$.

6. If we found y, then insert z'_e into the right and the left lists of y. Both lists are implemented with AVL trees, and the size of each list is at most the total number of the edges in the TIN. So this insert should be in $O(logn)$.

7. If we have reached a leaf without finding y, we insert a new leaf into the interval tree to store the new interval. Its interval lists will contain just z'_e, and its split value will be the midpoint of z'_e.

We are using balanced (AVL) trees both for the interval tree, and for the interval lists within each node of the interval tree. To keep the trees balanced, all insertions and deletions are followed by a *rebalance* operation. There exists the rebalance algorithm for AVL trees in $O(logn)$ time [Wei97]. An alternative method is *relaxed AVL tree* [LSSW97]. Instead of rebalancing the tree every time, we relax the restriction and accumulate a greater difference is heights before we need to adjust the height of the AVL tree.

Figure 4 illustrates an update to the interval tree when the reading of sensor s moves from 136.3 to 170.4. Note all edges incident on s need to be checked. In this figure, we need to update the position of intervals f, h, l, m, n in the interval tree. This tree is highly unbalanced; it is a snapshot after the insertion but before rebalancing.

Figure 5 shows the result after adjustment from the interval tree in figure 4 that was an highly unbalanced tree. When we try to insert n into the interval tree, we noticed that we need to insert a new node to store the interval n. And after adjustment, there are no intervals in the node whose split value is 74.35, and this will decrease the height of the interval tree by 1, so we delete this node. The time complexity of this algorithm is $O(logn)$.

Note that our algorithm, while sometimes adding new nodes (with singleton interval lists), does not delete old nodes when their interval lists become empty. We determined experimentally that there was no benefit in doing so. Since the sensor readings move up and down (rather than monotonically increasing or decreasing), the empty node is very likely to be used up at some point, and it turns out that keeping it around for this eventuality is more time efficient than deleting it right away.

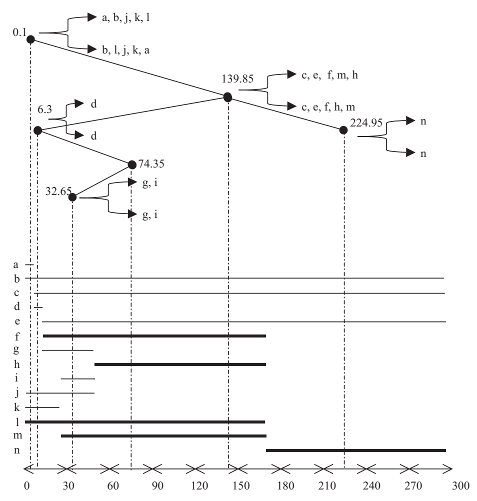

Figure 4: A dynamic TIN and the corresponding interval tree, before rebalancing.

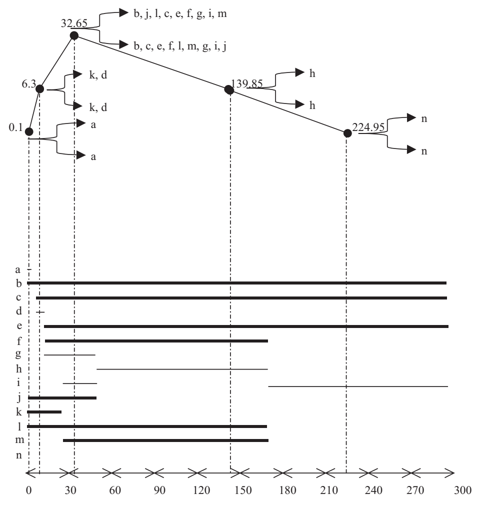

Figure 5: A dynamic TIN and the corresponding interval tree, after rebalancing.

To complete the performance analysis of the interval tree update algorithm, we need to know how many edges are incident on the given sensor s, since each of these k edges needs to be updated separately. Our approach was once again experimental. We measured the k, and the average of k is never more than 6, and it does not grow as the number of sensors increases. Therefore, we assume that k is $O(1)$. For each edge, we took its z-span interval from its old position and found an appropriate new position to insert the interval, rebalancing when needed. Each of these operations is in $O(logn)$ time. Therefore, the overall algorithm is $O(logn)$.

4 Performance

4.1 Assumptions

Our visualization algorithm relies on continuously updating the TIN and the corresponding interval tree. The updates are triggered by changes to sensor values, insertions (new sensors), or deletions (sensor loss). We use the *average-case* approach to complexity analysis, assuming that changes to sensor values happen much more often than either insertions or deletions.

Furthermore, we assume that not all changes to sensor values trigger an update. Due to the inherent "graininess" of any visualization medium, there is always a threshold δ so that changes smaller than δ can be ignored. This is accomplished by keeping both the old reading (i.e., the last one to cause an update) and the current reading, for each sensor. Whenever the difference between them is greater than δ, a new update is triggered.

4.2 Experiments

In our simulation, we used GNU C++ with OPENGL library under Linux platform to render the sensor terrain and isoline maps. We implemented the isolines extraction from sensor terrains using the algorithm we described above. In our experiment, we first generated terrain according to the sensors' reading using the algorithm in section 2. Then we constructed our interval tree using the algorithm in section 3.

For our sensor data, we started with the actual data which describes the terrain around the University of Connecticut. As shown in figure 6, we simulated a network of 257 sensors deployed around UConn which were in the region $(0,0)-(9600,10115)$. The sensor readings were the local height at that coordinate, which ranged from 0 to 420 (feet). Figure 6 shows the shaded sensor terrain (a) before and (b) after a sensor in lower left quadrant changed its value, from 350 to 149.49. One can clearly see the difference in the shape of the two sensor terrain.

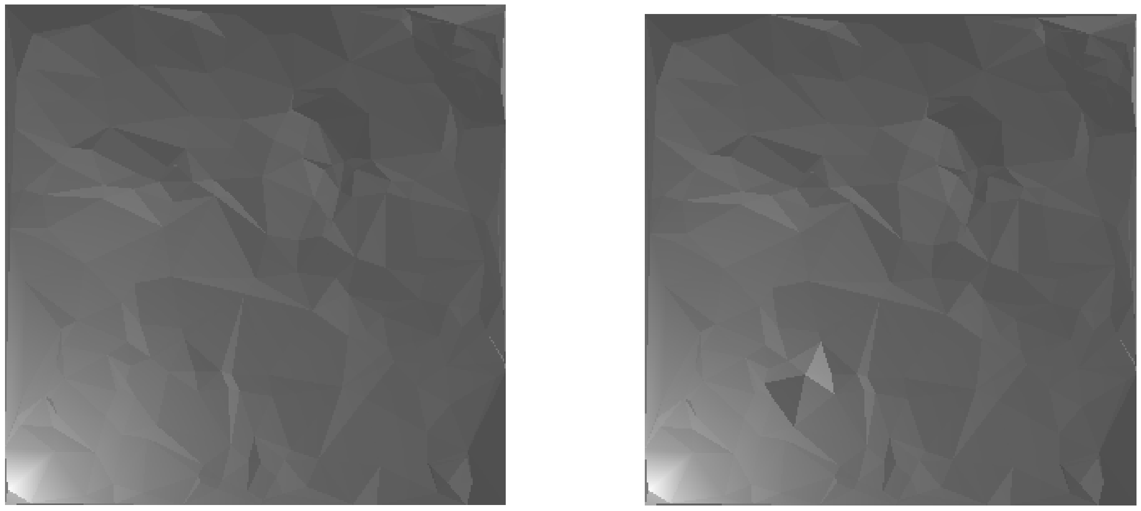

Figure 6: Sensor terrain of UConn. (a)before change; (b)after change.

After we generated the TIN and the isolines, we changed the reading of one sensor at a time. According to the sensors' reading, we updated the TIN and the interval trees. Figure 7 illustrates how the isolines can be affected by every change in sensor readings. It shows the TIN overlayed with the isolines; the thick line represents the isoline value 200, and the thin line represents the isoline value 300. When we change the reading of one sensor (colored as black) from 350 in (a) to 149.49 in (b), it is apparent how the isolines changed accordingly.

In addition, we also measured the performance of our update algorithms described in sections 2 and 3, plotting time performance against the number of sensors in the network, n. We measured n from 50 to 2500 in 50 unit intervals: $\{50,100,150,\ldots,2450,2500\}$. Given n sensors distributed

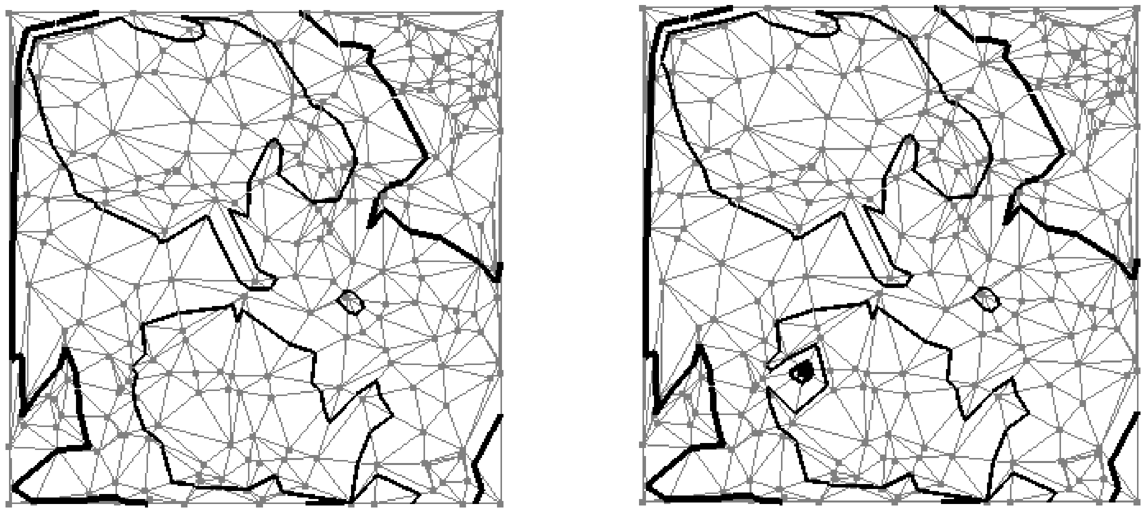

Figure 7: Isolines from TIN. (a)before change; (b)after change.

randomly in a region $(0,0) - (300,300)$, we chose one sensor at random and changed its reading, updating everything. We repeated this 100 times, getting the cumulative time for each n in microseconds. Figure 8 shows the plot we obtained from our experiments.

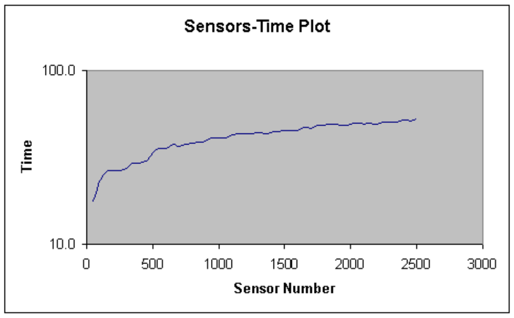

Figure 8: Time performance vs. number of sensors

The logarithmic trendline through this plot has the function $y = 9.5973(lnx) - 24.691$, and the value of .964 for R^2. This value of R^2 shows that there is a 96.4% reliability of the linear relationship between the plot and the trendline. From this picture, we can see that our algorithm is logarithmic and scalable. This confirms our analysis in section 3.

5 Related work

The visualization of sensor data involves much knowledge in computer graphics. A good review of rendering techniques such as transformations, shading, interpolation, texture mapping, ray tracing and so on and the mathematics theory behind can be found in [Wat99, Ang02]. One of the popular rendering libraries is OPENGL. OPENGL is said to be industry standard; it is stable, reliable, portable, extensible, scalable and easy to use. Documents about OPENGL are available from *http://www.opengl.org*. [SWND03] written by the OpenGL Architecture Review Board is the most authoritative one. In our simulation, we used GNU C++ with OPENGL under linux to render out sensor terrains and isolines map.

Interval tree first was propose by Edelsbrunner [Ede80] to efficiently retrieve intervals of real lines that contain a given query value. Cignoni et.al [CMM+97] uses interval tree as the data structure to extract isosurface. Chiang [CS97] describes how to extract isosurface from volumetric data using interval tree.

Van Kreveld [vK94] uses the interval tree as the data structure to extract isolines from TINs by associating each triangle with the intervals of the elevation it spans. There are several differences between that algorithm and ours. Their intervals are based on z-spans of triangles rather than edges. They do not use balanced trees as we do. Their algorithm is not dynamic (the update operations are not defined). Finally, their algorithm was never implemented.

To our knowledge, no one has described an algorithm to extract isolines efficiently and dynamically from sensor terrains as we have done. Related work has tried to extract isolines directly from sensor readings, in network. In [Est03], it is mentioned that sensor networks can be used to find contour and gradient. In [HHMS03], an isobar computation from sensor networks is performed as a form of aggregation; it assumes that every sensor is in a rectangular grid and merge these grids in-network. The communication between the sensors is based on a tree. The general process is like this way: each sensor got the isobar map from its children, combine its own information, and send the isobar map up to its parent. Finally, the root aggregate the isobar maps. So they need some polygon operations such as intersect and union. The time complexity is $O(nlogn)$, where n is the number of edges in the polygon. Whenever there are some sensors changing their reading, we need to compute the isobar again.

6 Conclusion and future work

In this work, we proposed to use interval trees in sensor networks to extract isolines. We proposed an updated interval tree algorithm when the sensor reading changes significantly. In our algorithm, we keep the time complexity $O(logn)$, where n is the number of the sensors in the TIN, to query or update the interval tree.

We have implemented 3D sensor data visualization, implemented, optimized and analyzed the isolines extraction algorithm from dynamic sensor networks. We implemented our algorithm in GNU C++ with OPENGL library under Linux and measure the time to change one sensor reading from the TIN and get the time-sensors plot. Finally, we discussed some related work.

Future work includes *multiresolution approaches* to the visualization for sensor networks. Suppose we have millions of sensors distributed in a wild area. We need a timely visualization of sensor terrains. A hierarchy of representations at various of levels of detail, which is called *multiresolution model*, can be used to achieved this goal. A survey about current multiresolution modeling techniques is given in [HG94]. [FP95] shows an algorithm to extract representations of terrain in variable resolution. A lot of literature, such [dBD95, CPS97], has multiresolution models, but none of it has considered the problem of dynamic visualization of TINs and isolines for sensor networks.

References

[Ang02] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach with OpenGL*. Pearson Addison-Wesley Publisher, July 6 2002.

[ASSC02a] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdai Cyirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, pages 102–114, August 2002.

[ASSC02b] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdai Cyirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38(4):393–422, 2002.

[CMM$^+$97] Paolo Cignoni, Paola Marino, Claudio Montani, Enrico Puppo, and Roberto Scopigno. Speeding up Isosurface Extraction Using Interval Tree. *IEEE Transcations on Visualization and Computer Graphics*, 3, April-June 1997.

[CPS97] Paolo Cignoni, Enrico Puppo, and Robert Scopigno. Representation and Visualization of Terrain Surfaces at Variable Resolution. *The Visual Computer, Springer International*, 13(5):199–217, 1997.

[CS97] Yi-Jen Chiang and Claudio T. Silva. I/O Optimal Isosurface Extraction. *IEEE Visualization '97*, pages 293–250, 1997.

[dBD95] Mark de Berg and Katrin Dobrindt. On Levels of Detail in Terrains. *Proceedings of the eleventh annual symposium on Computational Geometry*, pages 426–427, 1995.

[Dev99] Olivier Devillers. On Deletion in Delaunay Triangulations. *Proceedings of the Fifteenth Annual Symposium on Computational Geometry*, pages 181–188, June 1999.

[Ede80] Edelsbrunner. Dynamic Data Structure for Orthogonal Intersection Queries. *Tech. Rep. F59, Inst. Informationsverarb. Tech. Univ. Graz, Graz, Austria*, 1980.

[EGH99] Deborah Estrin, Ramesh Govindan, and John Heidemann. Next Century Challenges: Scalable Coordination in Sensor Networks. *Satis Kumar, Proceedings of MOBICOMM*, pages 263–270, 1999.

[Est03] Deborah Estrin. Embedded Networked Sensing for Environmental Monitoring: Applications and Challenges. *Invited talk, DIALM-POMC Joint Workshop on Foundations of Mobile Computing, San Diego, CA*, September 19 2003.

[FP95] Leila De Floriani and Enrico Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics (TOG)*, 14(4):363 – 411, October 1995.

[GH95] Michael Garland and Paul S. Heckbert. Fast Polygonal Approximations of Terrains and Height Fields. *Tech. Rep. CMU-CS-95-181, Carnegie Mellon University*, Sept 1995.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the Manipulation of General Subdivisions and the Computation of Vorono Diagrams. *ACM-Transactions on Graphics*, 4(2):74–123, 1985.

[GSK$^+$03] Dina Goldin, Mingjun Song, Ayferi Kutlu, Huayan Gao, and Hardik Dave. Georouting and Delta-gathering: Efficient Data Propagation Techniques for GeoSensor Networks. *GeoSensor Networks (GSN'03) Workshop, Portland, Maine*, Oct 2003.

[Hel90] Martin Heller. Triangulation Algorithms for Adaptive Terrain Modeling. *Proceedings of the 4th International Symposium of Spatial Data Handling*, pages 163–174, 1990.

[HG94] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc.

[HHMS03] Joseph M. Hellerstein, Wei Hong, Samuel Madden, and Kyle Stanek. Beyond Average: Towards Sophisticated Sensing with Queries. *2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, March 2003.

[Law77] Charles L. Lawson. Software for C^1 Surface Interpolation. *In John R. Rice, editior, Mathematical Software III, Academic Press, NY(Proc. of symp., Madison, WI, Mar.)*, pages 161–194, 1977.

[LSSW97] Kim S. Larsen, Eljas Soisalon-Soininen, and Peter Widmayer. Relaxed Balance through Standard Rotations. *AWorkshop on Algorithms and Data Structures*, 1997.

[MGD03] Mir Abolfazl Mostafavi, Christopher Gold, and Maciej Dakowicz. Delete and Insert Operations in Voronoi / Delaunay Methods and Applications. *Computers & Geosciences*, 29(4):523–530, May 2003.

[SWND03] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.4, Fourth Edition*. Addison-Wesley Publisher, November 14 2003.

[TM03] Malik Tubaishat and Sanjay Madria. Sensor networks: an overview. *IEEE Potentials*, 22(2):20–23, April-May 2003.

[vK94] Marc van Kreveld. Efficient Methods for Isoline Extraction from a Digital Elevation Model Based on Triangulated Irregular Networks. *In Sixth International Symposium on Spatial Data Handling Proc.*, pages 835–847, 1994.

[vK97] Marc van Kreveld. Digital Elevation Models and TIN Algorithms. *Algorithmic Foundations of Geographic Information Systems in Lecture Notes in Computer Science (tutorials),Springer-Verlag, Berlin*, 1340:37–78, 1997.

[Wat99] Alan H. Watt. *3D Computer Graphics*. Addison-Wesley Publisher, December 6 1999.

[Wei97] Mark Allen Weiss. *Data Structures and Algorithm Analysis in C*. Addison-Wesley Publisher, Jul 1997.