Algorithms for Triangulated Terrains*

Marc van Kreveld

Dept. of Computer Science
Utrecht University
The Netherlands
marc@cs.ruu.nl

Abstract

Digital elevation models can represent many types of geographic data. One of the common digital elevation models is the triangulated irregular network (also called TIN, or polyhedral terrain, or triangulated terrain). We discuss ways to represent a TIN in a data structure, and give some of the basic algorithms that work on TINs. These include retrieving contour lines, computing perspective views, and constructing TINs from other digital elevation data. We also give a recent method to compress and decompress a TIN for storage and transmission purposes.

1 Introduction

Geographic Information Systems are large software packages that store and operate on geographic data. They are large because they usually include a full database system, and set of functions to operate on spatial data. It is the spatial (or geometric) component that distinguishes geographic information systems (or GIS for short) from standard databases.

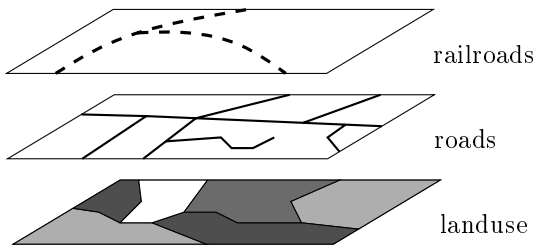

Figure 1: Layers of geographic data in a GIS.

Geographic data comes in many forms. Borders of countries and provences, locations of roads and hospitals, and pollution of the lakes and rivers are types of man-made geographic data. Natural geographic data includes elevation above sea level, annual percipitation, soil type, and much more. GIS store the different types of geographic data in different *map layers*, so there is a map layer with the major roads, one with the rivers, one with the current land use, and one with the elevation above sea level. GIS typically store from ten up to a few hundred map layers.

*Research is partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

It is rather useful to have types of data in different layers. A GIS user may wish to see a map on the screen with only cities and railroads, because this particular user plans to travel by train somewhere. Or a physical geographer may wish to see the *overlay* of soil type and amount wind erosion, to study how these two data sets are related.

For any specified location on the Earth or on a map, one can say that some value is associated to it in a particular theme. For instance, at $53°15'$ latitude and $6°$ longitude the particular land use is "agricultural", the elevation is 1 meter above sea level, and the annual percipitation is 790 mm. So the value can be a name, or a number, or something else. In the first case the data is called *nominal*, in the second case it is called *ratio*. (Traditionally, four scales of measurement were used: nominal, ordinal, interval, and ratio [27]. Geographic data can also be a direction or a vector, like wind.)

This paper deals with data on the ratio scale, which can be seen as a function from a 2-dimensional region to the reals. The domain can be referenced by geographic coordinates, for example, but we'll do as if we have a function from the xy-plane into the third, z-dimension. Elevation above sea level is the most obvious type of data that is modelled by such a function.

One of the problems when storing and computing on elevation data is that the amount of data can be enormous. Currently available for the US is elevation data for points at a regular spacing of 30 meters, which means a few billion points. In the future data sets of considerably smaller spacing will be collected, leading to even larger data sets. A consequence is that only *efficient* algorithms can be used to process the data. This paper surveys the common models to store elevation data, in particular the triangulated irregular network. We then discuss a couple of the basic algorithms that operate on this model, like determining contour lines, visualization by perspective views, and conversion from other elevation models. Then we concentrate on a recent result on efficient compression and decompression for the triangulated irregular network, which is important for background storage and for network transmission.

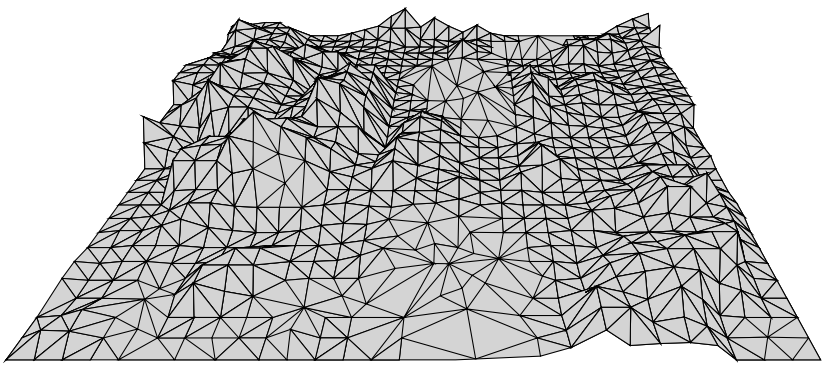

Figure 2: Perspective view of a triangulated irregular network.

This paper presents several algorithms for terrains. To analyse and express the efficiency of these algorithms we'll use big-Oh notation. For instance, for a triangulated irregular network determined by a set of n data points, the contour lines of some given elevation can be determined in $O(n)$, or linear time. It is important that the algorithms have running times like $O(n)$ or $O(n \log n)$, because quadratic time usually is too slow in practice for the amount of data involved.

2 Digital elevation models

In the computer the true geographic elevation (function) has to be approximated by some finite representation of it. This is called a *digital elevation model*. There are three common digital elevation models (or DEMs for short). They are the regular square grid, the contour lines, and the triangulated irregular network.

The regular square grid is a 2-dimensional array where each entry stores an elevation. An entry represents some region on the Earth of, say, 10 by 10 meters, and the stored value is the elevation of the center point of the region.

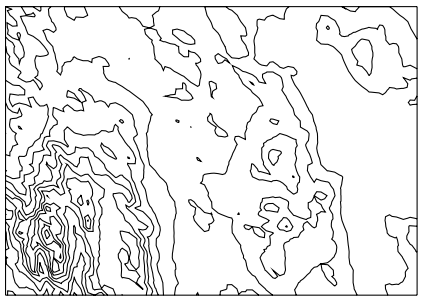

Figure 3: Contour lines of a terrain.

A contour map consists of a collection of separate contour lines that each have some elevation. Each contour line can be stored as a sequence of control points through which the contour line is assumed to pass. A contour line is a closed curve, or it may have its endpoints on the boundary of the region for which the elevation function is defined. It can be represented by a polygon or polygonal line, or a spline curve.

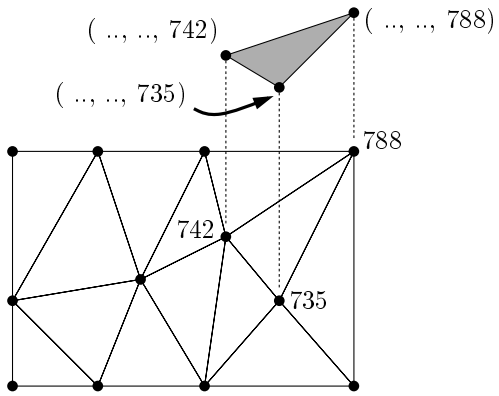

Figure 4: Triangulation with elevation values at vertices.

The triangulated irregular network (or TIN, or polyhedral terrain, or triangulated terrain) is a third way to represent elevation. A triangulation on some finite set S of points is a planar subdivision into triangles that is maximal, and such that only the points of S appear as vertices of the subdivision. When used as an elevation model, the vertices of the triangulation store an elevation value. The elevations on the edges and inside the triangles of the subdivision are obtained by linear interpolation. So if some point $q = (\bar{x}, \bar{y})$ lies inside a triangle with vertices $v_i = (x_i, y_i)$, $v_j = (x_j, y_j)$, and $v_k = (x_k, y_k)$ with elevations z_i, z_j, and z_k, then we consider

the unique plane that passes through the three points (x_i, y_i, z_i), (x_j, y_j, z_j), and (x_k, y_k, z_k). Then we determine the value $\bar{z}$ of q such that the point $(\bar{x}, \bar{y}, \bar{z})$ lies on the plane, which gives the interpolated value.

Note that there are many different triangulations possible of a given set of points. All must have the same number of edges and triangles, because triangulations are maximal planar subdivisions. By Euler's relation for planar graphs, the number of edges and triangles is linear in the number of points, the vertices that determine the subdivision. Different triangulations of a point set lead to different elevations at points on edges and inside triangles. For interpolation purposes, it seems natural to choose one that has small, well-shaped triangles. The standard choice is the so-called *Delaunay triangulation* that will be discussed later.

The contour line model isn't used as a way to store elevation data permanently. However, one of the ways to obtain elevation data is by digitizing the contour lines on contour maps, so one may have to deal with the contour model nevertheless. Often, the contour model is converted to the grid or TIN model before further processing.

One of the advantages of the TIN over the grid is that it is adaptive: more data points can be used in regions where there is much elevation change, and fewer points in regions where the elevation hardly changes. One of the disadvantages of the TIN when compared to the grid is that the algorithms usually are somewhat more complex.

3 Data structures for a TIN

This section gives two different ways to represent TINs. One is edge based and the other is triangle based. In both cases it will be possible to navigate on the TIN, going from one triangle to an adjacent one efficiently, or finding all triangles that are incident to a particular vertex. The two structures are simplified versions of data structures that can store arbitrary planar subdivsions such as the doubly-connected edge list, winged edge, or quad edge structure, commonly used in GIS, graphics, and computational geometry [6, 8, 14, 32].

In the triangle-based structure for a TIN, any triangle and vertex is represented by an object or record. A triangle object has six references, three to the adjacent triangles, and three to the incident vertices. Vertices are stored by objects that only have the x-, y-, and z-coordinates. Edges are not stored explicitly, but they can be determined from the structure if necessary. The CGAL-library of geometric primitives and algorithms provides this structure [2].

In the edge-based structure for a TIN, any edge is an object that has a dual purpose. It connects two vertices and it separates two triangles. In the structure any edge object has references to two vertex objects (of the vertices it connects) and to two triangle objects (of the triangles it separates; there may only be one). The triangle objects have references to the three edge objects that bound the triangle. Vertices again only store the x-, y-, and z-coordinates. There are no references between vertex objects and triangle objects.

4 Visualization and traversal of a TIN

The most common way to show elevation on maps is by contour lines at regular intervals. The map can be enhanced by *hill shading*, a technique where an imaginary light source is placed in 3-dimensional space, and parts of the terrain that don't receive much light are shaded. Another way to visualize a terrain is by a perspective view. The algorithms required for visualization are standard graph algorithms on the TIN structure in both cases.

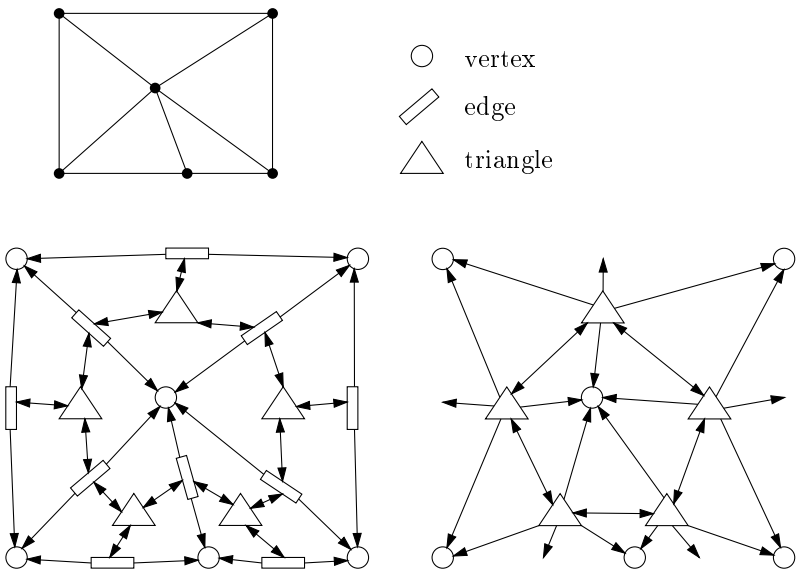

Figure 5: Left, the edge-based and right, the triangle-based structures of the triangulation shown at the top.

4.1 Contour maps

To determine all contour lines of, say, 1000 meters, on a TIN representing a terrain, observe that any triangle contains at most one line segment that is part of the contour lines of 1000 meters. In fact, the contour lines of 1000 meters are nothing else than the cross-section of the terrain as a 3-dimensional surface, and the horizontal plane $z = 1000$. So, to determine the contour lines on a TIN it suffices to examine every triangle once and see if it contributes to the contour lines. Similarly, to compute hill shading for a TIN, one needs to determine the slope of each triangle and its aspect (the compass direction to which the triangle is facing, in the xy-projection). The slope and aspect determine how much a triangle is shaded, given a position of the light source. As for contour lines, it suffices to examine every triangle of the TIN once to compute hill shading for the whole terrain.

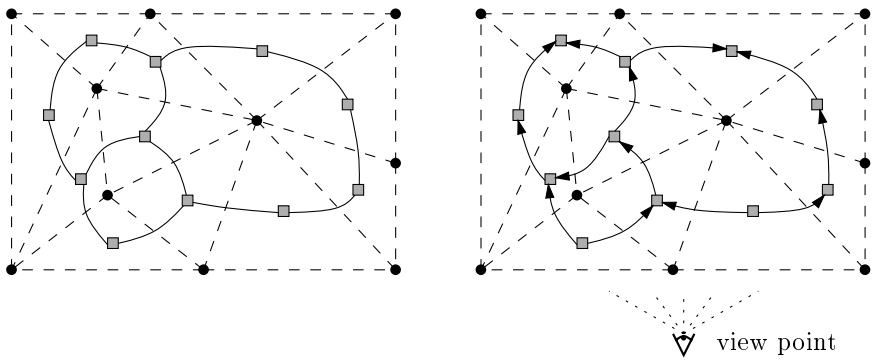

Figure 6: Left, dual graph of the TIN, with grey nodes and solid arcs. Right, dual directed acyclic graph for agiven view point.

A traversal that visits every triangle once is like a depth-first search in a graph dual to

the TIN. This graph has a node for every triangle, and two nodes are connected by an arc if the corresponding triangles share an edge in the triangulation. Both the triangle-based and the edge-based TIN structures implicitly store this graph, and depth-first search through all the triangles is easy if a mark bit is available in every object, to see if it has been visited before. So for a TIN with n vertices and, hence, $O(n)$ edges and triangles, one can compute all contour lines of a given elevation in $O(n)$ time by depth-first search. Similarly, one can compute hill shading in linear time.

4.2 Perspective views

Another way to visualize a TIN is by a perspective view of the terrain. We can produce such a view using the Painter's Algorithm, where all triangles are drawn from back to front, so that the ones more to the front erase the ones more to the back. What is the appropriate drawing order for the triangles of a TIN? Given the view point and a TIN, every two triangles that share an edge must be drawn in a specific order (unless the line supporting the edge happens to pass through the view point). This necessary condition on the drawing order happens to be sufficient as well: if for all edges of a TIN, the triangle 'behind' the edge is drawn before the triangle 'in front of' the edge, then the drawing order is correct. So the drawing order is a partial order as well, and it can be obtained by a topological sort of a directed graph. Again, this directed graph is implicitly present in either of the two structures for storing a TIN. It is the same dual graph as we used for depth-first search, but now the arcs have a direction. The direction of an arc can be determined by checking the coordinates of the endpoints of the dual edge of that arc, and the coordinates of the view point. So a TIN with n vertices can be drawn in perspective view in $O(n)$ time using the Painter's Algorithm.

5 Construction of a TIN

We'll now study algorithms for constructing a TIN from elevation data. First we consider the case that a set of data points with elevations is given, and the problem is to construct a triangulation on that set. Then we assume that the input is a large grid of regularly spaced data points, and the problem is to produce a TIN that approximates the grid to within a specified maximum error.

5.1 Delaunay triangulation on a point set

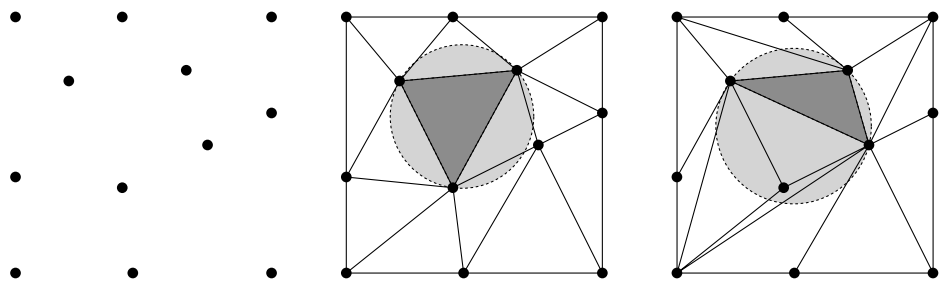

Figure 7: Left, a set of points in the plane. Middle, the Delaunay triangulation of the points, where all triangles have the empty circle property as shown for one triangle. Right, a triangulation of the same points with a triangle that doesn't have the empty circle property.

The most popular triangulation of a set of points without doubt is the Delaunay triangulation. For a set P of n points in the plane, three points $p, q, r \in P$ are the vertices of a triangle in the Delaunay triangulation of P if the circle through p, q, r doesn't contain any other points of P. If P doesn't contain four points that are co-circular, then the definition just given really defines a unique triangulation. It has the property that, among all triangulations of P, it is the one that maximizes the minimum angle of the triangles in the triangulation (in other words, no triangulation has a larger smallest angle). This property implies that the triangles generally will be well-shaped, which is important for the interpolation function it defines. More information on the Delaunay triangulation is in the book of Okabe, Boots, and Sugihara [19] and in textbooks on computational geometry [6, 20, 21].

Several algorithms are known that construct the Delaunay triangulation of a set on n points in $O(n \log n)$ time, and this is optimal. We sketch one that is simple and requires $O(n \log n)$ time expected, based on randomized incremental construction. The expectation in the running time is only dependent on the random choices made by the algorithm and is valid for any set of points, independent of the distribution.

In the randomized incremental construction algorithm for the Delaunay triangulation, all points of P are added one by one, and after each insertion, the Delaunay triangulation is restored to incorporate the new point. So we in fact compute a sequence $T_1, \ldots, T_n$ of triangulations, where triangulation T_i contains i points of P. To compute T_{i+1}, we choose one of the remaining $n - i$ points at random and insert it. So if $p_1, \ldots, p_n$ is a random permutation of P, we can simply insert the points in this order, and T_i is the Delaunay triangulation of $p_1, \ldots, p_i$.

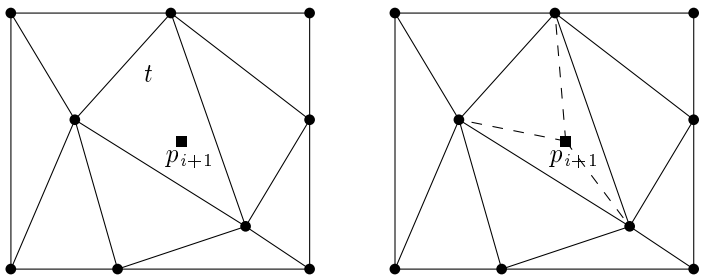

Figure 8: New point p_{i+1} in triangle t of T_i, and three of the new edges.

One insertion of a point requires locating the triangle of the triangulation that contains the point, and then the actual insertion. We'll skip the location part and assume that the new point p_{i+1} falls inside some triangle t of triangulation T_i. For certain, the Delaunay triangulation T_{i+1} will contain the three edges between p_{i+1} and the vertices of t. It is not certain, however, that the three triangles of which p_{i+1} is now a vertex really satisfy the Delaunay property: the circle through p_{i+1} and two of the vertices of t may contain other vertices of T_i. If this is the case, we'll *flip*: we destroy the edge between the two vertices of t and we create a new edge from p_{i+1} to repair the triangulation.

The correctness of such a flip depends on two facts that can be shown from the Delaunay property of T_i. Firstly, all edges that appear in T_{i+1} but not in T_i must have p_{i+1} as one of the endpoints. Secondly, if any triangle t' incident to p_{i+1} doesn't have the empty circle property, then at least the vertex opposite of the edge opposite of p_{i+1} in t' must be in this circle. The flip will connect p_{i+1} to this vertex.

Any flip destroys one triangle incident to p_{i+1} and another triangle, and creates two

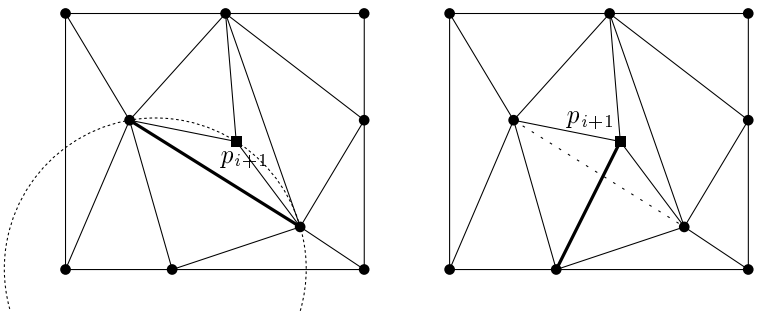

Figure 9: Flipping an edge because the empty circle property is violated.

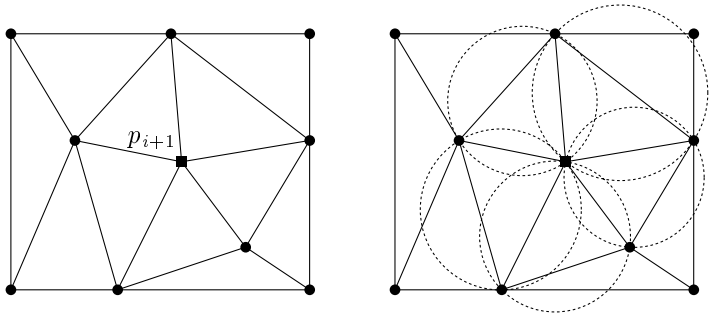

Figure 10: One more flip is needed to obtain T_{i+1}. Right, the empty circles of the triangles incident to the new point.

triangles incident to p_{i+1}. Any edge opposite to p_{i+1} of a triangle incident to p_{i+1} must be tested to see if a flip is necessary. If all such edges are tested and none have to be flipped to guarantee the empty circle property, then we have computed T_{i+1}, the Delaunay triangulation of $p_1, \ldots, p_{i+1}$.

After locating the triangle of T_i in which p_{i+1} lies, all the flipping and testing requires time linear in the number of new edges created, which is the degree of point p_{i+1} in the triangulation T_{i+1}.

One can show that the expected time for the flipping is constant, even though it is linear in the worst case. Consider the triangulation T_{i+1}. Since $p_1, \ldots, p_{i+1}$ is a random permutation of the set $\{p_1, \ldots, p_{i+1}\}$ of $i+1$ points, each one of those $i+1$ points is equally likely to have been inserted as the last one. The sums of the degrees of all points in T_{i+1} is $O(i+1)$, since T_{i+1} is a planar triangulation of $i+1$ points. So the average degree of a point is constant, which shows that the expected time for inserting p_{i+1} after locating the triangle t is also constant. This proof idea is called *backwards analysis* [6, 24]. A more complete description of randomized incremental construction of the Delaunay triangulation and its analysis can be found in [4, 6, 13].

5.2 Delaunay triangulation to approximate an elevation grid

We proceed with the construction of a TIN from a large grid of elevation data. The algorithm we'll describe selects a subset of the grid points, such that the Delaunay triangulation of this subset is a TIN that approximates the elevation at all grid points to within a prespecified error ϵ. In other words, at all grid points, the absolute difference of its z-coordinate and the

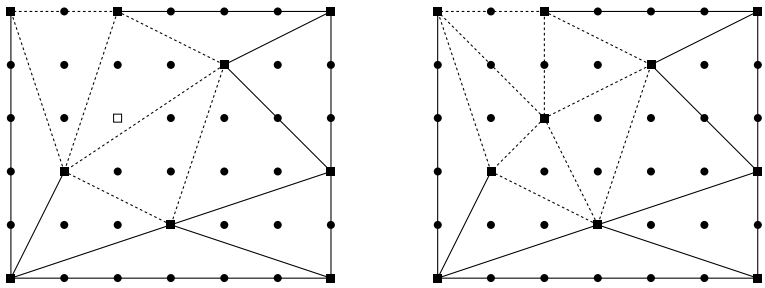

Figure 11: The right TIN shows the situation if the square grid point on the left is the one with maximum error.

interpolated z-coordinate of the TIN is at most ϵ. The approach is to start with a coarse TIN with only a few vertices, and keep adding more points from the grid to the TIN to obtain a TIN with smaller error. The algorithm was described before by Heller [16], Fjällström [9], and Heckbert and Garland [15].

1. Let P be the set of midpoints of grid cells, with their elevation value. Take the four corner points and remove them from P, and put them in a set S under construction.
2. Compute the Delaunay triangulation $\mathrm{DT}(S)$ of S.
3. Determine for all points in P in which triangle of $\mathrm{DT}(S)$ they fall. For points on edges we can choose either one. Store with each triangle of $\mathrm{DT}(S)$ a list of the points of P that lie in it.
4. If all points of P are approximated with error at most ϵ by the current TIN then the TIN is accepted and the algorithm stops. Otherwise, take the point with maximum approximation error, remove it from P and add it to S. Continue at step 2.

If we assume a simple and slow implementation of the algorithm, we observe that at most n times a Delaunay triangulation is computed. For each one, the points in P are distibuted among the triangles of $\mathrm{DT}(S)$. This requires for the whole algorithm $\Theta(n^3)$ tests of the type point in triangle, if a linear number of points is added to S.

A much faster implementation has a worst case performance of $O(n^2 \log n)$ time, and in typical situations even better: typically $O(n \log n)$ time. The algorithm resembles incremental construction of the Delaunay triangulation to some extent. But the TIN construction algorithm must also distribute the points of P and find the one with maximum approximation error. We'll show that these steps can be done efficiently.

Assume that $p \in P$ has been determined as the point with maximum error, bigger than ϵ, and p must be removed from P and added to S. Then we locate the triangle t of $\mathrm{DT}(S)$ that contains p, and we find the vertices that will become neighbors of p in $\mathrm{DT}(S \cup \{p\})$. This update step of the Delaunay triangulation is the same as in the incremental construction algorithm. To distribute the points of $P \backslash \{p\}$ over the triangles of $\mathrm{DT}(S \cup \{p\})$, we know that only the triangles of which p is a vertex in $\mathrm{DT}(S \cup \{p\})$ have changed. So for all triangles of $\mathrm{DT}(S)$ that don't exist in $\mathrm{DT}(S \cup \{p\})$, we collect the associated lists of points. These points are distibuted among the new triangles and stored in new lists.

The problem that remains is locating the point with maximum error. It is solved as follows. For each triangle of the TIN we determine the point of P inside it with maximum

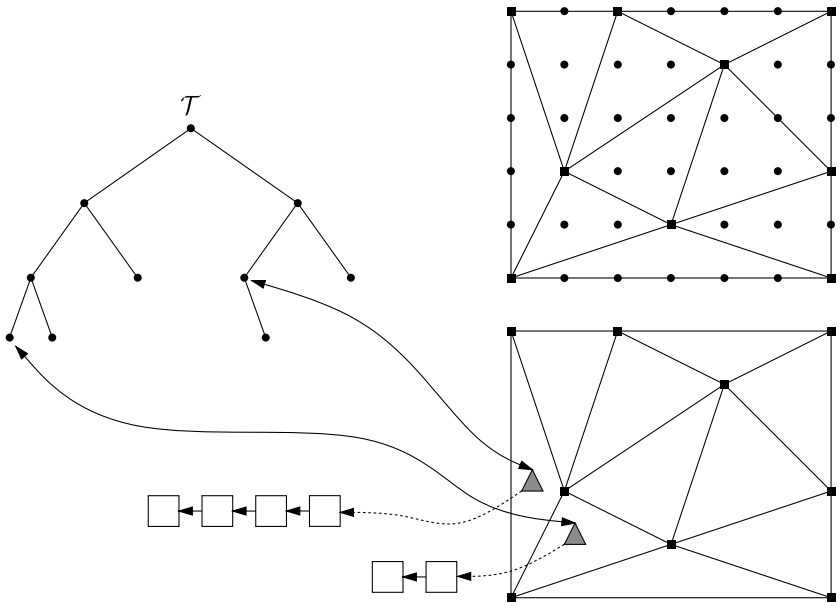

Figure 12: The situation for a TIN with vertices shown as small squares (top right), and the corresponding structure with a few of the pointers between triangle objects, list elements, and tree nodes.

error. These points are stored in the nodes of a balanced binary search tree $\mathcal{T}$ sorted on error. This allows us to locate the point p with maximum error efficiently; it is in the rightmost leaf of $\mathcal{T}$. Before p is moved from P to S, the Delaunay triangulation must be changed accordingly. To find the triangle in $\mathrm{DT}(S)$ that contains p we'll use a pointer from the node in $\mathcal{T}$ to the triangle record in the TIN structure; such pointers are shown as dashed lines with arrows in Figure 12. The triangle records are shown as grey triangles. After updating the TIN to be $\mathrm{DT}(S \cup \{p\})$ we move p from P to S.

Then we reorganize the lists that were stored with the triangles. When p was added to the Delaunay triangulation, some triangles were destroyed. The point of P inside each one that had maximum error is deleted from $\mathcal{T}$. The lists of points of the destroyed triangles contain p and the points that must be distributed among the new triangles, and stored in new lists. For each of the new lists we must find the point that has the maximum error in the corresponding triangle, and store it in $\mathcal{T}$.

If k is the number of neighbors of p in $\mathrm{DT}(S \cup \{p\})$, then $k-2$ triangles were destroyed and k new ones were made. Let m be the number of points in the triangles incident to p in $\mathrm{DT}(S \cup \{p\})$. Then the iteration that added p as a vertex of the TIN requires $O(k + \log n)$ time for updating the Delaunay triangulation, $O(km)$ time to redistribute the m points over the k triangles, and $O(k \log n)$ time to update the balanced binary tree $\mathcal{T}$. In the worst case, m and k are both linear in n, giving an worst case performance of $O(n^3)$. But redistribution of the points can also be done in $O(k+m \log m)$ time by sorting the m points by angle around p. Since all new triangles in the TIN are incident to p, we can distribute the m points over the k triangles by using the sorted order. The modification improves the worst case running time to $O(n^2 \log n)$.

One can expect that k is usually constant, and after a couple of iterations of the algorithm, m will probably be much smaller than n. The more iterations, the smaller m tends to be.

One can expect that the algorithm behaves more like the best case than like the worst case, for typical inputs. In the best case, k will be constant, and every list of points stored with a triangle reduces in length considerably each time it is involved in a redistribution. On the average, a new vertex has degree about six, which means that four triangles are destroyed. One can hope that the points of P in these four triangles are distributed more or less evenly over the six new triangles, implying that each of the six new triangles, on the average, only has $2/3$ of the points of P when compared to the average of the four triangles that were destroyed. So later iterations in the algorithm tend to go faster and faster, and m decreases from linear in n to a constant. Or the algorithm may stop sooner because the error criterion is met. If k is assumed to be a constant, we needn't use the modification to distribute the points, but simply spend $O(km) = O(m)$ time. Using an amortized analysis technique, one can show that the whole algorithm will take $O(n \log n)$ time under the (best case) assumptions given.

Emperical tests of the running time on real world input has shown that the typical running time seems to be closer to linear than to quadratic [9, 15, 16].

6 Compression and decompression of a TIN

In this section we explain a recent result for the compression and decompression of a TIN, assuming that the Delaunay triangulation is used for the data points. As we mentioned in the introduction, terrain data usually is huge in size, which means that a lot of storage is needed for the permanent storage of terrains, and a lot of bandwidth is needed for transmission over a network. We'll give a simple and efficient algorithm for compression and for decompression. The idea applies to some other geometric structures as well, like Voronoi diagrams, convex hulls, and vertical decompositions. It was introduced by Jack Snoeyink and the author of this paper [25, 26].

A structure like the Delaunay triangulation can be compressed by omitting all structural information (egdes and triangles), leaving only the vertices. This is true because the Delaunay triangulation is uniquely determined by its vertices. So to compress a Delaunay triangulation we need only store the vertices by the x-, y-, and z-coordinates. However, if we would do this, it'll take $O(n \log n)$ time to reconstruct the TIN, since it takes this much time to construct the Delaunay triangulation of n points from scratch. We'll show that if the vertices are stored in a particular order, then the reconstruction takes only $O(n)$ time, and it is simpler too. The algorithm to compute this particular order takes $O(n)$ time as well. Interestingly, the sorted order on x-coordinate doesn't work to (re)construct the Delaunay triangulation of a point set; the $\Omega(n \log n)$ lower bound remains valid [7].

Our first algorithm produces a permutation of the data; our second takes this permutation and reconstructs the Delaunay triangulation.

6.1 Compression

Let P be a set of n points in the plane, of which the Delaunay triangulation is given in some structure. For simplicity we'll assume that the point set lies in some rectangle of which the four vertices are also data points of the set, and that there are no other four co-circular points in P. To construct the permutation of P, we deconstruct its Delaunay triangulation in phases, by deleting groups of points in a specific way. More precisely (but still as an outline of the algorithm), in one phase we find a subset of the vertices that are an independent set in the Delaunay triangulation, see Figure 13. Then we delete this subset and restore the Delaunay triangulation for the remaining vertices, while remembering in which new triangles

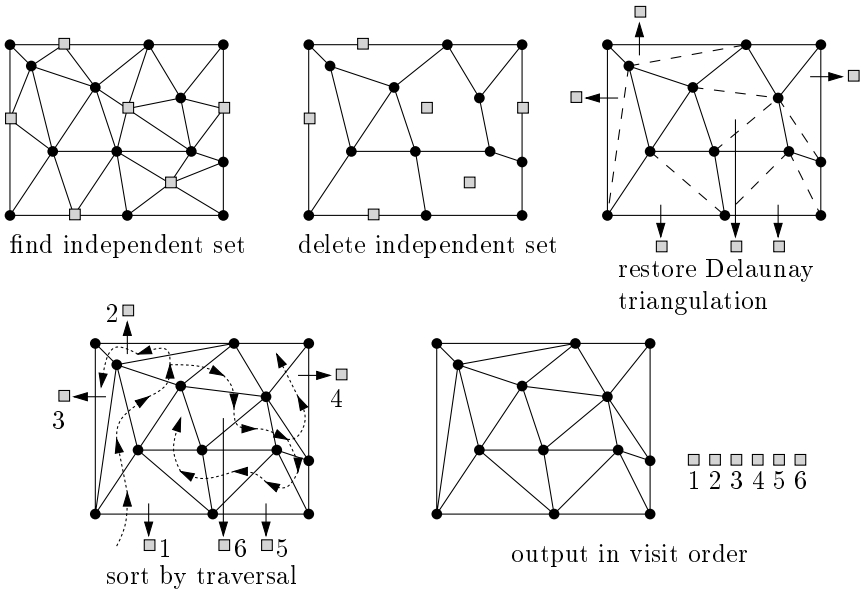

Figure 13: The steps for one phase in the compression algorithm.

the deleted points fall. Next the Delaunay triangulation is traversed, and on the way we collect the subset of points that were removed. So the order in which the triangles are visited determines the order of the subset. This concludes one phase, and we repeat the steps by finding a new independent set in the smaller Delaunay triangulation.

6.1.1 Find an independent set S.

The algorithm starts with the Delaunay triangulation of a point set P. There are three conditions on the independent set we choose in the Delaunay triangulation, for efficiency or for simplicity of the algorithm. Firstly, we won't choose any of the four corner points in this step. Secondly, we'd like all vertices in the chosen independent set to have constant degree in the graph, say, each chosen vertex has at most ten neighbors. Thirdly, the independent set should be large, in any case at least some constant fraction of the whole set. By the four-color theorem, any planar graph on m vertices has an independent set of size $\lceil m/4 \rceil$. However, it takes quadratic time to find it [23]. Much simpler algorithms are known to compute an independent set of size $m/5$ [3]. However, the algorithms don't give only vertices with constant degree.

The following simple algorithm determines an independent set satisfying these conditions. The idea was introduced by Kirkpatrick [17], and used for efficient planar point location. Initially, all vertices are unmarked and not chosen. Traverse the vertices of the Delaunay triangulation in any order and choose a vertex if it has degree at most 10, it is not a corner, and it is unmarked. After choosing a vertex we mark the at most 10 neighbors, and then we continue the traversal. This simple algorithm runs in linear time.

One can show that at least $5/121$ of all non-corner vertices are chosen. For convenience of analysis we forget about not choosing corner vertices. Let m be the total number of vertices in the Delaunay triangulation. The total number of neighbors of all vertices is at less than $6m$ by Euler's relation. Therefore, there are less than $6m/11$ vertices with degree > 10, and more than $5m/11$ vertices with degree ≤ 10. So, an algorithm that chooses one vertex with degree

at most 10 and throws away the neighbors chooses at least 1 of 11 vertices of degree ≤ 10, and in total 1/11 times $5m/11$ vertices are chosen in the independent set S. The analysis given can be refined to get a larger constant than 5/121. A more clever linear time algorithm can guarantee choosing roughly 4/21 of the vertices, and in practice gives a fraction close to 1/3 [26].

6.1.2 Delete the points of S.

After computing the subset S of independent vertices we must delete each one, and restore the Delaunay triangulation. One can show that the only edges that were in $\mathrm{DT}(P)$ but not in $\mathrm{DT}(P \setminus S)$ are the ones incident to the vertices of S. We start by removing all these edges. Since S is an independent set and each vertex in S had degree ≤ 10, the graph we obtain has polygonal faces with at most 10 vertices. The faces can be triangulated separately, and in linear time we determine the Delaunay triangulation of $P \setminus S$. Then we determine for each of the deleted points of S in which new triangle of $\mathrm{DT}(P \setminus S)$ it occurs. This is easy, because a deleted point occurs in one of the at most 7 new triangles formed to triangulate the face it was in. So the total deletion time is linear.

6.1.3 Traverse to sort the deleted points.

The set of points S that has just been deleted will be placed in some order next. This order is determined by any traversal algorithm of the triangles of the Delaunay triangulation, like the depth-first search algorithm. When we visit a triangle that contains a deleted point, we collect it and add it to the sequence of points already collected.

6.1.4 End a phase and continue.

We add an end-of-phase marker to the collected sequence of points. Then we decide whether or not to restart by finding an independent set of the current, smaller Delaunay triangulation. We restart if more than five vertices are left. If only the four corner vertices and one more vertex remain, we simply store them in any order, and the algorithm terminates. For the final sequence of all n points, we start with the final five points, then an end-of-phase marker, and then the points chosen and deleted in the last phase, in the order in which they were collected. Then another end-of-phase marker, followed by the points of the second-last phase, and so on.

The steps given above for one phase take $O(m)$ time for a Delaunay triangulation with m vertices. But it is also true that all phases together take $O(n)$ time for an initial Delaunay triangulation with n vertices. Since the problem that remains after a phase has at most $116n/121$ vertices, the running time $T(n)$ satisfies the recurrence $T(n) \leq T(116n/121) + O(n)$ if $n > 5$, and $T(n) = O(1)$ if $n \leq 5$. This recurrence solves to $T(n) = O(n)$ time.

Some final remarks about the compression algorithm. To make the algorithm work as discribed we didn't choose the four corner vertices in the independent set. These wouldn't fall in a triangle of the new, smaller Delaunay triangulation when they are deleted. Therefore we leave the four corners in until the end. Secondly, the $O(\log n)$ markers that were placed in the sequence are used in the decompression algorithm. There are a few ways to avoid them altogether, and only use the order of the points [25, 26].

6.2 Decompression

Decompression is done when a computer gets the data from background storage, or receives the data over a network. The data arrives in a sequence, and the algorithm can start to reconstruct the Delaunay triangulation as soon as the first points arrive. According to the compression algorithm, these are the four corner points and one additional point. The following points are inserted in phases. All steps of the algorithm basically are the reverse of some step of compression, and therefore we only sketch it briefly.

The first five points are used to initialize the reconstruction, so we start by computing their Delaunay triangulation. Then one phase, until the next end-of-phase marker, can be inserted. In any phase, the next sequence of points between two end-of-phase markers is inserted into the current triangulation as follows. Traverse the current Delaunay triangulation using *the same traversal algorithm used to collect the points*, but the traversal now serves to locate the points and store them with the triangles containing them. During the traversal we need only test if the next triangle contains the first point of the sequence. We know that the first point in the sequence must be the first point that falls in a triangle, and all following points are in triangles visited later in the traversal. After all points between two end-of-phase markers are located in the triangles, we insert them in the Delaunay triangulation. This is the flipping algorithm described before in this paper. The addition to the Delaunay triangulation ends a phase, and the next sequence of points between end-of-phase markers can be added.

Just like the compression algorithm, decompression takes $O(n)$ time for the Delaunay triangulation of n points. For a comparison, the randomized incremental construction algorithm for the Delaunay triangulation takes $O(\log n)$ time expected to locate the triangle that contains the next point to be added. Then it spends $O(k)$ time for flipping if the next point has degree k. And one can agree that k is constant in the expected case. The reconstruction algorithm of this section does the point location for one new point in $O(1)$ time in the amortized sense, that is, the location of all n points in $O(n)$ triangles takes at most $O(n)$ time together. Then $O(1)$ is used for adding a new point to the Delaunay triangulation, since we have by construction that the new point has constant degree.

7 Conclusions and further reading

This paper surveyed a couple of geometric algorithms that can be used when working with digital elevation data. These algorithms were developed in the research areas of computational geometry and GIS. Both areas also have strong connections with computer graphics.

Two simple algorithms for visualization were presented. More efficient methods are known to find contour lines of a terrain, by using preprocessing [10, 28]. De Berg has written a more extended survey on TIN visualization, including the use of levels of detail of terrains in visualization [5]. More generally, visualization in GIS is treated in a book edited by MacEachren and Taylor [18], see also [1, 22, 30].

Algorithms for the construction of TINs from digital elevation in another form has been studied extensively. This can be the triangulation between contour lines, grid to TIN conversion as in this paper, or producing a TIN from point data, with or without an interpolation method. Surveys on digital elevation models contain many references to such methods [29, 31].

Compression of digital elevation data hasn't been studied so much yet. For gridded data, Franklin gives a number of experimental results showing how well standard image compression techniques work for elevation data [11, 12].

More algorithms that operate on terrains and can be used in GIS have been described in a survey of the author of this paper [29].

References

[1] B.P. Buttenfield and W.A. Mackaness. Visualization. In D.J. Maguire, M.F. Goodchild, and D.W. Rhind, editors, *Geographical Information Systems – Principles and Applications*, volume 1, pages 427–443. Longman Scientific & Technical, 1991.

[2] CGAL home page. `http://www.cs.ruu.nl/CGAL/`.

[3] N. Chiba, T. Nishizeki, and N. Saito. A linear 5-coloring algorithm of planar graphs. *J. of Algorithms*, 2:317–327, 1981.

[4] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete Comput. Geom.*, 4:387–421, 1989.

[5] M. de Berg. Visualization of TINs. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, Lecture Notes in Comp. Science. Springer-Verlag, 1997. to appear.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[7] H. Djidjev and A. Lingas. On computing Voronoi diagrams for sorted point sets. *Internat. J. Comput. Geom. Appl.*, 5:327–337, 1995.

[8] D. P. Dobkin. Computational geometry and computer graphics. *Proc. IEEE*, 80(9):1400–1411, September 1992.

[9] P.-O. Fjällström. Polyhedral approximation of bivariate functions. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 187–190, 1991.

[10] L. De Floriani, D. Mirra, and E. Puppo. Extracting contour lines from a hierarchical surface model. In *Eurographics'93*, volume 12, pages 249–260, 1993.

[11] Wm Randolph Franklin. Compressing elevation data. In *Advances in Spatial Databases (SSD'95)*, number 951 in Lecture Notes in Computer Science, pages 385–404, Berlin, 1995. Springer-Verlag.

[12] Wm Randolph Franklin and A. Said. Lossy compression elevation data. In *Proc. 7th Int. Symp. on Spatial Data Handling*, pages 8B.29–8B.41, 1996.

[13] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[14] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.

[15] P. S. Heckbert and M. Garland. Fast polygonal approximation of terrains and height fields. Report CMU-CS-95-181, Carnegie Mellon University, 1995.

[16] M. Heller. Triangulation algorithms for adaptive terrain modeling. In *Proc. 4th Int. Symp. on Spatial Data Handling*, pages 163–174, 1990.

[17] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[18] A.M. MacEachren and D.R.F. Taylor, editors. *Visualization in Modern Cartography.* Elsevier Science Inc., New York, 1994.

[19] Atsuyuki Okabe, Barry Boots, and Kokichi Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* John Wiley & Sons, Chichester, UK, 1992.

[20] J. O'Rourke. *Computational Geometry in C.* Cambridge Univ. Press, NY, 1994.

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

[22] Relief: depicting a surface on a map. `http://acorn.educ.nottingham.ac.uk/ShellCent/maps/relief.html`.

[23] N. Robertson, D.P. Sanders, P. Seymour, and R. Thomas. Efficiently four-coloring planar graphs. In *Proc. 28th ACM Symp. Theor. Comp.*, pages 571–575, 1996.

[24] R. Seidel. Backwards analysis of randomized geometric algorithms. In J. Pach, editor, *New Trends in Discrete and Computational Geometry*, volume 10 of *Algorithms and Combinatorics*, pages 37–68. Springer-Verlag, 1993.

[25] J. Snoeyink and M. van Kreveld. Good orders for incremental (re)construction. In *Proc. 13th ACM Symp. Computational Geometry*, pages 400–402, 1997.

[26] J. Snoeyink and M. van Kreveld. Linear time reconstruction of the Delaunay triangulation with applications. In *Proc. 7th Europ. Symp. Algorithms*, Lecture Notes in Comp. Science. Springer-Verlag, 1997.

[27] S.S. Stevens. On the theory of scales of measurement. *Science*, 103:677–680, 1946.

[28] M. van Kreveld. Efficient methods for isoline extraction from a TIN. *Int. J. of GIS*, 10:523–540, 1996.

[29] M. van Kreveld. Digital elevation models and TIN algorithms. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of GIS*, Lecture Notes in Comp. Science. Springer-Verlag, 1997. to appear.

[30] Visualization techniques for landscape evaluation, literature review. `http://bamboo.mluri.sari.ac.uk/ jo/litrev/chapters.html`.

[31] R. Weibel and M. Heller. Digital terrain modelling. In D. J. Maguire, M. F. Goodchild, and D. W. Rhind, editors, *Geographical Information Systems – Principles and Applications*, pages 269–297. Longman, London, 1991.

[32] M.F. Worboys. *GIS: A Computing Perspective.* Taylor & Francis, London, 1995.