Automatic Generation of Triangular Irregular Networks using Greedy Cuts

Cláudio T. Silva* Joseph S. B. Mitchell‡ Arie E. Kaufman*

*Department of Computer Science
‡Department of Applied Mathematics & Statistics
State University of New York at Stony Brook
Stony Brook, NY 11794

Abstract

We propose a new approach to the automatic generation of triangular irregular networks from dense terrain models. We have developed and implemented an algorithm based on the greedy principle used to compute minimum-link paths in polygons. Our algorithm works by taking greedy cuts ("bites") out of a simple closed polygon that bounds the yet-to-be triangulated region. The algorithm starts with a large polygon, bounding the whole extent of the terrain to be triangulated, and works its way inward, performing at each step one of three basic operations: ear cutting, greedy biting, and edge splitting. We give experimental evidence that our method is competitive with current algorithms and has the potential to be faster and to generate many fewer triangles. Also, it is able to keep the structural terrain fidelity at almost no extra cost in running time and it requires very little memory beyond that for the input height array.

1 Introduction

A *terrain* is the graph of a function of two variables. The function gives the *elevation* of each point in the domain. Terrain models are widely used in visualization and computer graphics applications; such as flight simulators, financial visualization tools, strategic military analyzers, geographic information systems, and video games. Thus, it is of the utmost importance that primitive operations can be performed in real-time. Several factors may affect the efficiency of algorithms that operate on terrain; the most important are probably the *size* of the input and its underlying data structure.

The most common source of digital terrain elevation data is the DEM (*Digital Elevation Model*), supplied by the U.S. Geological Survey. A DEM is basically a two-dimensional floating point height array. It contains an extremely high level of redundancy, which in turn usually forbids real-time applications from using it. Several alternative data structures have been proposed, including contour lines, quad-trees, and TINs (*Triangular Irregular Networks*). TINs stand out as being one of the most convenient to use for rendering and other geometric manipulation operations. A TIN is a set of contiguous non-overlapping triangles whose vertices are placed adaptively over the DEM domain [8]. The automatic generation of TIN models from DEM models is an important area of research and is the main topic of this article. Several factors are important in judging the quality of the TIN representation of a given DEM (list partially adapted from [19, 20]):

- *Numerical accuracy* – measured as maximum, mean, or standard deviation error;
- *Visual accuracy* – usually assessed by inspection and by number of "slivery" triangles;
- *Size of the model* – measured as the number of output triangles;
- *Algorithm complexity* – measured in terms of the time to generate the TIN and the memory requirement.

Fowler and Little [8] have introduced one of the first (and still very popular) methods to address the problem of automatic generation of TINs directly from DEMs. Their method is very simple. First, they classify the points by automatically choosing some "important" features of the terrain, such as ridges and peaks. They describe this phase of the algorithm as constructing the "structural fidelity" of the model; i.e., the TIN representation should have the same geographical features as the DEM. Then, they incrementally compute a triangulation of the points; in their case, they chose to use the Delaunay triangulation. At each step, a new point is added to the triangulation until no points are farther from the original surface than a certain predefined threshold. This phase

is designed to preserve the "statistical fidelity" (i.e, to make it fit the specified error bound).

Franklin [9] has proposed a similar approach back in 1973. It appears that his method had no notion of structural fidelity, and he did not use the Delaunay triangulation as the basis for his method. A new version of his code is publically available, and we used it for comparison with our method. A detailed description of his algorithm and code is given in Section 4. Recently, substantial research has been conducted on creating hierarchical structures on top of TINs [7, 21], and on techniques to improve the quality of TIN meshes [22]. Scarlatos' dissertation [19] is a good survey of terrain modeling and representation. A very recent approach to building hierarchical models of terrains is given by de Berg and Dobrindt [6], who apply a hierarchical refinement of the Delaunay triangulation to represent terrain TINs at many levels of detail. See also [13, 14] for an approach called the "drop heuristic" and its comparison with other methods. Common to all these previous methods is the necessity to have a complete starting triangulation that is either *refined* by adding new points, or *decimated* [23] by removing redundant points. These approaches require that the algorithm maintain in memory a complete triangulation representation of the input, extended with various pieces of global information (e.g., most deviant point per triangle). The need for *global* information impacts the running time and memory requirements of these algorithms.

Our work is based on an entirely different approach for the triangulation and simplification of the data. It is based on an idea in the method developed by Mitchell and Suri [17], where a greedy set cover approach has been developed for approximating convex surfaces, and used recently by Varshney [25] in heuristics for simplifying CAD models. We can consider the input DEM to be an instance of a TIN with very high resolution. In particular, each pixel of the DEM corresponds to four elevation data points, and we consider these to define two adjacent triangles of a surface. (A square pixel can be triangulated in one of two ways. We triangulate all pixels uniformly, with diagonals at 45-degrees.) Our goal is to simplify this input TIN surface to create a new TIN that has far fewer triangles, but is still within a specified error bound of the original surface. From an algorithmic point of view, terrain simplification is hard (NP-hard) [5, 4], but some polynomial-time algorithms are known for computing a nearly-optimal (i.e., nearly minimum-facet) approximating surface, guaranteed to be within a factor $O(\log n)$ of optimal (see [1, 3, 15, 17]), or within a constant factor of optimal, if the surface is convex (see [2]). Unfortunately, the polynomial-time bounds for these theoretically good approaches is rather high (at least cubic). In contrast, from the practical point of view, most of the previous computer graphics and geography research in the area is based on heuristics for generating triangulations that "fit" the original data, but have no guarantees, either in terms of the closeness to optimal or in terms of the worst-case running time.

The principle that drives our method (and is related to that of [3, 17, 25]) is the same greedy principle that is used to compute minimum-link paths in simple polygons. This problem is well studied in computational geometry [12, 16, 24] and can be used to find an optimal piecewise-linear approximation to a function of a single variable (see [10]). Our problem is of one higher dimension. We use a *greedy-facet* approach, selecting large triangles (bites) by which to extend an approximating surface, based on their feasibility (i.e., they must lie within an ϵ-fattening of the original surface) and on their size (e.g., area of projection in the x-y plane). The use of greedy algorithms is known to give provably good approximation results in many combinatorial optimization problems, for example, the *set cover* problem is approximated within a log factor of optimal by a natural greedy algorithm, and this fact leads [17] to a provably good approximation algorithm for the convex case of our problem. We have not yet been able to prove that our algorithm has a guaranteed effectiveness with respect to optimal, but we are hopeful that interesting properties can be proved about its performance. Currently, our code only handles inputs in the form of elevation arrays, but in principle, there is no reason why it cannot be extended to arbitrary polyhedral terrains, or, for that matter, polyhedral surfaces in general. Extensions to higher dimensions also seem possible, that is, for simplifying piecewise-linear functions of three variables defined over tetrahedralizations of 3-space.

Instead of a top-down approach that starts with a feasible Delaunay triangulation and tries to generate finer and finer Delaunay triangulations by adding points to the already created triangulations, our algorithm works bottom-up. At each step a greedy cut is taken from an untriangulated polygon. The greedy cuts are an attempt to sample the data at the lowest possible resolution, thus minimizing the number of triangles in the output. A full description of our algorithm is given in the next section.

2 The Algorithm

This section gives a high-level description of our algorithm. The problem definition is as follows:

Given an input array, H, of heights $H(x, y)$, $0 \leq x < m$ and $0 \leq y < n$, whose data points are sampled from a regular grid on a rectangle R, and some $\epsilon > 0$ specifying an error tolerance. Find a triangulated surface (TIN) that represents a terrain on R, such that the TIN has a small number of triangles (T_i), and each data point given by the array $H(x, y)$ lies within vertical distance ϵ of the TIN.

The algorithm maintains a list of *untriangulated simple polygons*, $\mathcal{P}$, which represents the portion of

R over which no triangulated surface has yet been constructed. At each step, our goal is to select a maximum area triangle T within one of the polygons $P \in \mathcal{P}$, such that (1) the vertices $v_1 = (x_1, y_1)$, $v_2 = (x_2, y_2)$, and $v_3 = (x_3, y_3)$ of T are grid points (points (x, y) for which we have the altitude $H(x, y)$); (2) at least two of these vertices are vertices of P (i.e., T shares at least one edge with P); and (3) the triangle T corresponds to a triangle T' in space (with coordinates $(x_1, y_1, H(x_1, y_1))$, $(x_2, y_2, H(x_2, y_2))$, $(x_3, y_3, H(x_3, y_3))$) such that T' is "feasible" with respect to ϵ (see below for a precise definition). Because input data is sampled using a regular grid, the area of T is a good estimation of its combinatorial coverage (how many data points it covers). The ideal version of our algorithm searches all candidate triangles T and picks the best at each stage. However, for the sake of efficiency, the implemented version of our algorithm does not search all possible triangles T; instead, we do an approximate (limited) search for the best T, based on three basic operations, which will be described below.

Since each polygon $P \in \mathcal{P}$ corresponds to an independent subproblem, we can work on each separately. (There is no particular ordering in how we store the polygons $P \in \mathcal{P}$.) Thus, at each step of the algorithm, a *bite* (triangle) T is taken out of the polygon P at the head of the list $\mathcal{P}$, until P is reduced to a single feasible triangle, or it is divided into two new simple polygons, each of which is inserted into the list. The final result of our algorithm is the list of all triangles (bites), $\mathcal{T}$. There is no need to store in memory the list $\mathcal{T}$ of triangles as it is generated. Each triangle can be written out directly to a file. No triangle connectivity information is saved at this point. Each polygon $P \in \mathcal{P}$ is saved as a simple list of vertices, in counter-clockwise order. Thus, only very small and simple data structures are required.

We ought to define precisely what we mean by a triangle (in space) being "feasible" for input terrain H, with respect to a given ϵ. As already mentioned, we can consider the input DEM H to be an instance of a TIN (a polyhedral surface, S), even though no triangulation is explicitly given. Specifically, to fix that one of the many triangulations we consider to be the input surface, we consider point $(x, y, H(x, y))$ to have six neighbors, namely, those data points corresponding to $(x \pm 1, y \pm 1)$ (the standard four grid neighbors) and the diagonal points $(x+1, y+1)$ and $(x-1, y-1)$.

We say that a triangle T' (in space) satisfies *weak feasibility with respect to* ϵ if, for every grid point (x, y) that lies within the projection T of T' onto the (x, y)-plane, T' intersects the vertical segment joining $(x, y, H(x, y) - \epsilon)$ and $(x, y, H(x, y) + \epsilon)$. In other words, T' fits the function at the relevant internal grid points. Note that if T' has a very "skinny" or "small" projection (e.g., so that T contains no grid points at all), then it will certainly satisfy weak feasibility.

We say that triangle T' (in space) satisfies *strong feasibility with respect to* ϵ if T' lies completely above the surface $S^{-\epsilon}$ and completely below the surface $S^{+\epsilon}$, where $S^{-\epsilon}$ (resp., $S^{+\epsilon}$) is the polyhedral surface (TIN) obtained by shifting S downwards (resp., upwards) by ϵ. Note that if T' satisfies strong feasibility, then it certainly satisfies weak feasibility (but the converse is clearly false). The notion of strong feasibility applies directly to approximating arbitrary input terrains (e.g., given by a TIN rather than a DEM).

In order to test weak feasibility of T', we only have to examine the elevations at grid points internal to the projected triangle T. Such internal grid points are identified using a standard scan conversion of T. In Figure 1, we indicate these grid points with small squares. Strong feasibility, however, requires that we also check the altitudes corresponding to those points (indicated with circles in Figure 1) that lie at the intersections of an edge of T with a grid edge.

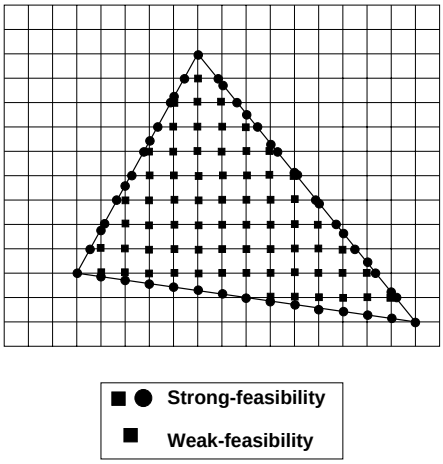


Figure 1: *Weak and strong feasibility.*

The algorithm works by performing three basic operations, one at a time: ear cutting, greedy biting, and edge splitting. Each operation is applied to a current active polygon. The next sections describe each of these operations in more detail.

Ear Cutting

This operation traverses a polygon $P \in \mathcal{P}$ looking for possible "ears" to cut. An *ear* of a simple polygon P is a triangle contained within P that shares two of its edges with P. We simply traverse the boundary of the polygon, "cutting off" any ear which we discover that corresponds to a *feasible* triangle (i.e., one that meets the feasibility criterion for ϵ). Given a vertex v_i, we check if the edge (v_i, v_{i+2}) is an internal diagonal to the polygon, that is, it is to the inside of the polygon and it does not intersect any other edge. This operation can easily be done in linear time by a simple traversal of the boundary of P. Using a dynamic triangulation of P, and performing "ray shooting queries", one can actually check in time $O(\log k)$ if (v_i, v_{i+2}) is

an ear of a simple k-gon [11], but the simple linear-time method is likely to be more practical (since k is typically small) and is what we currently have implemented.

Each cut we perform lowers the complexity (number of edges) of polygon P by one, thus taking the algorithm closer to completion. Ear cutting is essential for the algorithm to terminate. In general, it will be the final step in any run of the algorithm. Also, it has a tendency to turn obtuse angles into acute ones, which eventually leads to larger edges (hence triangles) in the triangulation. Ear cutting is the mechanism the algorithm uses to adapt itself to lower sampling rates (larger triangles).

Ear cutting fails when no more feasible ears exist. This happens when the size of the edges of P are too large, and the ears cover too much area in the polygon. In this case, there must be some way to make edges smaller, which leads to higher sampling rates. In order to adapt to more complicated terrains, we introduce two additional basic operations: greedy biting and edge splitting.

Greedy Biting

In this basic operation, we find a point v inside the polygon P and an edge, (v_i, v_{i+1}) of P, such that (v_i, v, v_{i+1}) forms a triangle, T, inside P that meets the feasibility criterion. We accomplish two things with this operation: (1) subdividing an edge of P in two (replacing (v_i, v_{i+1}) with (v_i, v) and (v, v_{i+1})), thereby achieving a higher "sampling rate"; and, (2) taking a bite out of the polygon P, thus progressing further in "eating away" all of P. The actual operation is a bit more complicated, as it needs to handle choices of v that may be a vertex of P and lead to P being split into two disjoint new simple polygons.

The greedy biting operation works as follows:

- *Bite.* For the polygon P, for each edge (v_i, v_{i+1}) search for a point $v \in P$ such that (v_i, v, v_{i+1}) corresponds to a feasible triangle. For efficiency, we search for such a point v in a neighborhood of (v_i, v_{i+1}). Currently, we limit the search to grid points along (close to) the vector perpendicular to (v_i, v_{i+1}) at the midpoint of (v_i, v_{i+1}). We use a binary search, starting at a point whose distance from (v_i, v_{i+1}) is roughly $|v_i v_{i+1}|$, then halving the distance at each step until a point is found (or we fail). (By trying other search strategies for v, we can likely improve the algorithm performance. This is being investigated.)
- *Split.* If the "Bite" step succeeds in finding a point v for which (v_i, v, v_{i+1}) corresponds to a feasible triangle, we will potentially split polygon P. We search for the closest edge (v_j, v_{j+1}) to v. If the triangle (v_j, v, v_{j+1}) also corresponds to a feasible triangle, we subdivide (split) the polygon P into two simple polygons, outputting both triangles ((v_i, v, v_{i+1}) and (v_j, v, v_{j+1})); otherwise, we simply output (v_i, v, v_{i+1}) without splitting P.

Edge Splitting

It may happen that both ear clipping and greedy biting fail to find a feasible triangle. In this case, our algorithm attempts to split some edge of the polygon P. Checking each edge of P in succession, starting with the longest, we look for an edge to split (roughly) in half (or possibly in smaller pieces, if splitting in half fails). When we split edge (v_i, v_{i+1}) at a (grid) point v, we are actually creating a skinny (feasible) triangle, (v_i, v, v_{i+1}). Since the triangles created in this way are small or "slivery", we prefer not to perform this operation very often. Indeed, in practice this phase of the algorithm is seldomly needed.

Initialization

Each phase of our algorithm works to triangulate the interior of a simple polygon P, with feasible triangles. In order to generate the first such polygon, bounding the whole domain R, we apply a one-dimensional version of our algorithm in each of the four cross sections (defined by the vertical planes $x = 0, m$, $y = 0, n$) that correspond to the boundary of the region R. The algorithm can be considered to be a simplified version of the standard min-link path method of Suri [24], applied to the discrete data points between the offset curves obtained by shifting the terrain surface up/down by ϵ. See Figure 2.

Main Algorithm

The algorithm simply applies the above three operations, one at a time, giving priority (in order) to ear cutting, greedy biting, and then edge splitting. A complete description of our algorithm is outlined as follows:

Greedy Cuts Algorithm

(0) Initialize $\mathcal{P}$ to be a list of one element – the single polygon obtained by the initialization procedure above.

(1) While $\mathcal{P}$ is not empty, do

- **(a)** Let $P \in \mathcal{P}$.
- **(b)** If P is a single feasible triangle, output this triangle, and remove P from $\mathcal{P}$.
- **(c)** Else, while P is not fully triangulated,
  - **(i)** Perform ear cutting on P, until no feasible ears exist.
  - **(ii)** Perform greedy biting on P. If this results in a greedy bite that splits P, then remove P from $\mathcal{P}$, add the two new polygons to $\mathcal{P}$, and go to (1). Otherwise, if at least one greedy bite is found (for some edge of P), go to (1) (without splitting P).
  - **(iii)** Perform an edge split for P.

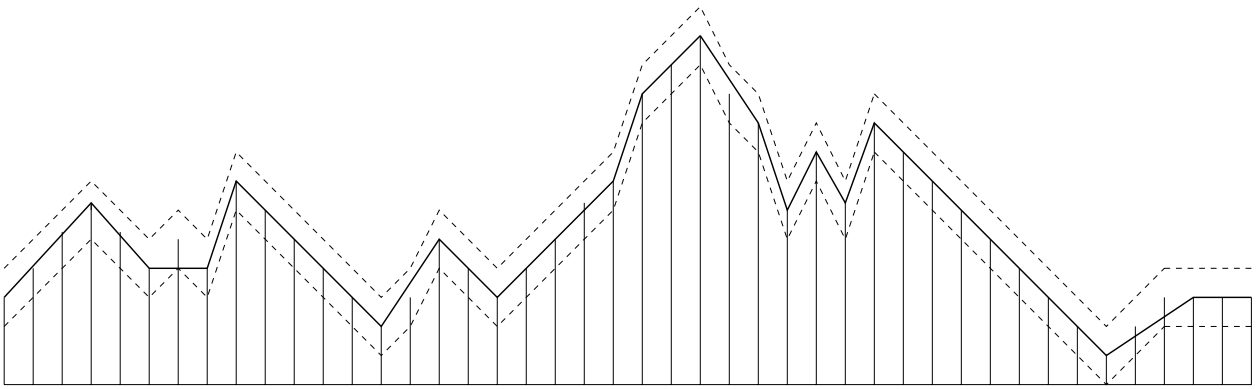


Figure 2: *The solid line is calculated by a greedy method. In linking data points, go as far as possible without exiting the strip defined by the dashed lines.*

3 Discussion

Terrain Sampling

One of the most interesting properties of our algorithm is the way it samples the dataset. It generates large triangles in places of relatively little change and small triangles in areas of more radical change. It is interesting to try to analyze how this happens, and here is where we can see the nice coupling of properties between the ear cutting phase and the others. If the terrain is largely uniform, ear cutting generally leads to longer and longer edges of P, until we encounter a region of high complexity, at which point edges are subdivided by greedy biting or edge splitting (a method of increasing the sampling resolution). Once we triangulate the high complexity region, ear cutting again makes the edges on the boundary larger and larger, i.e., making the triangles larger. Our algorithm therefore has a natural mechanism for attempting to minimize the number of triangles required. (Of course, as we have already said, our algorithm is not guaranteed to find a true minimum (an NP-hard problem).) The strategy of where/when to apply each of our three operations affects which regions get sampled at higher resolutions. Thus, we continue to experiment with further variants of our search strategy in hopes of obtaining better and smaller triangulations.

Maintaining Structural Fidelity

A primary objective in any algorithm that simplifies (compresses) data is to maintain as much of the important structure of the input as possible. Our algorithm generates a TIN that is close to the input DEM, according to the given tolerance ϵ. However, beyond the constraint of being ϵ-close, one may wish to place further restrictions on the structural fidelity; for example, one may wish to preserve a selected set of point features or of edge features, requiring that the surface approximation include these points and segments in the output TIN. In top-down algorithms, such requirements can be incorporated using constraints; for example, line segments can be preserved using constrained Delaunay triangulation (e.g., [6]). In our bottom-up algorithm, we can incorporate such constraints directly, at low cost, within the test for triangle feasibility: A triangle T' is not feasible if its projection, T, contains a point feature on its interior or boundary, except at a vertex, or intersects an edge feature, except if the edge is an edge of T. Further, our algorithm can maintain the structure of an edge or a ridge, at a *lower* resolution (within, say, ϵ) than the full resolution, by executing the (lower dimensional) initialization step in a vertical wall (plane) through each constraint edge.

Termination

It is important to consider whether or not our algorithm ever terminates. Could it ever get "stuck" and fail to generate any further triangles, even though the list of untriangulated regions, $\mathcal{P}$, is not empty? The answer is "no" for the case of the weak feasibility condition, assuming that greedy biting is done by searching over all possible bites. As a proof, consider a polygon $P \in \mathcal{P}$. If P has no grid points, then any ear of P is feasible. (Any simple polygon with at least 4 vertices has at least two ears, by the "Two Ear Theorem" [18].) If P has grid points in its interior, then there must exist a triangulation of these points within P (since any polygonal domain can be triangulated). All triangles in this triangulation must obey weak feasibility. In particular, there must exist a triangle T that shares at least one of its edges with P. Such a triangle is either a (feasible) ear of P (found in ear cutting) or a potential bite (found in greedy biting, assuming that we do a full search). This proves termination.

In the strong feasibility case, however we get a different situation. Because of the discrete nature of the allowed output (i.e., triangles must use original data points, since we do not allow Steiner points), and the continuous nature of the strong feasibility condition (which joins data points to form a polyhedral surface constraint), there are (rare) instances in which the al-

gorithm, as implemented, can get stuck when using strong feasibility. In response to this, we have implemented a simple feature that will guarantee termination in all cases. If the algorithm cannot find a feasible triangle, then it relaxes the feasibility condition in ear cutting, and finds, instead, an ear that has the smallest deviation from the original DEM. (This same feature allows us to limit our search in greedy biting and still guarantee termination in the weak feasibility case.)

Complexity

We first remark that our algorithm uses very little internal memory. Other than the input data array, we keep track only of the list $\mathcal{P}$ of polygons, each of which is (typically) very small. Triangles that we generate do *not* need to be stored, but can be written out directly to disk. In contrast, methods that rely on triangulation refinement must maintain some sort of topological data structure for the full set of triangles. Typically, one would expect that if the output size (number of triangles) is k, then the boundary of the polgyons $\mathcal{P}$ at any given instant will have roughly size $\sqrt{k}$.

It is difficult to prove a bound on the expected run time of the algorithm. Clearly, the *worst-case* running time is polynomial in the input size, since each primitive test or computation can easily be performed, usually in worst-case linear time (linear, generally, in the size of $P \in \mathcal{P}$). However, our experimental evidence suggests that the algorithm runs in time roughly linear in the input size.

The output complexity for our algorithm is again hard to estimate from a theoretical point of view. The problem we are trying to solve approximately is known to be NP-hard, in general. Thus, the best we can hope for is that we may be able to prove a worst-case bound on the ratio of our output size (number of triangles) to the number of triangles in an optimal TIN. There is good theoretical basis (e.g., from greedy set cover heuristics) to suggest that our algorithm (or a close variant thereof) will never produce more than a small (e.g., logarithmic) factor more triangles than is possible for a given ϵ. Proving such a fact remains an open (theoretical) problem. Perhaps the best indication we have of the effectiveness of the algorithm is the experimental data we have, which suggests that our algorithm is obtaining substantially fewer (roughly 20-30 percent) triangles than the competing algorithm, for the same error tolerance ϵ.

4 Experimental Results

Our algorithm is relatively simple to implement. Our C implementation has only about 4,000 lines of code. The code uses several computational geometry primitives, many of which come from O'Rourke [18], including segment intersection testing, diagonal classification, point classification (point location with respect to a simple polygon). With these primitives in hand, and routines to handle simple polygon operations (e.g., splitting an edge of a polygon, inserting a vertex.), it is fairly easy to implement the algorithm described in Section 2. As with all geometric algorithms, care has to be taken with special (degenerate) cases that arise from collinearities.

In order to study its performance, we have conducted tests of our algorithm and compared it with Franklin's algorithm, which is a top-down approach. We compared the speed, average error bound (over all the triangles), and the complexity of the output (measured in the number of triangles). We ran both algorithms on the following types of input: real terrain datasets, artificially generated terrains arising from performing cuts to generate faults, and artificially generated terrains arising from lifting triangulations.

Franklin's algorithm

Franklin's algorithm is described in [9], and is a nice and efficient example of a top-down triangulation method. Initially, the algorithm approximates the DEM by 2 triangles. Then, a general step of the algorithm involves finding the most deviant point in each already generated triangle and inserting this new point into the triangulation, splitting one triangle into three. Each time a point is inserted, the algorithm checks each quadrilateral that is formed by a pair of adjacent triangles, at least one of which is a new triangle (one of the three incident on the new point). A local condition on the quadrilateral determines whether or not to perform a diagonal swap. The original code works by performing a pre-determined number of splits. We have changed the code to make as many splits as necessary in order to meet a prespecified error bound ϵ. Franklin's implementation is done carefully, with emphasis on efficiency. For the sake of speed, it uses internal memory as much as possible.

Experimental Data

Our experiments were conducted on a Silicon Graphics ONYX, equipped with two 100Mhz R4400 processors and 64MB of RAM. Only one of the processors was used. The time to read the terrain datasets from the disk was not included in our runtimes. In Table 1, we show the results of running three algorithms on seven real terrain datasets. We ran Franklin's algorithm (f), and two versions of our algorithm — one using weak feasibility (w), and one using strong feasibility (s). The table shows the choice of ϵ, the running times, and the total number of triangles in the output TIN, for each of the seven terrains. The input terrains were all scaled to be 120-by-120 elevation arrays, for uniformity of testing.

In summary, *greedy cuts* with *weak-feasibility* beats Franklin's code in the number of output triangles in all instances. *Greedy cuts* with *strong-feasibility* loses in most cases, but it applies a stricter accuracy requirement than Franklin's algorithm (which uses weak feasibility). Franklin's optimized code is usually faster than our (relatively naive) implementation. We expect that with fine tuning and optimization, our algorithm

Table 1: *Running times (in sec) of three algorithms on seven real terrain data sets.* (f) *indicates Franklin's code;* (w) *and* (s) *indicate our algorithm with weak and strong feasibility, respectively. All terrains are 120×120 elevation arrays. The error bounds (ϵ) were chosen to keep the number of triangles* (Trgs.) *in the output approximately in the 1000 to 3000 range. Memory usage is the number of 8Kbyte pages allocated.*

Terrain	ϵ	Time	Trgs.	Memory
Buffalo	2.5 (f)	3.2	1994	6229
	2.5 (w)	8.12	1641	428
	2.5 (s)	21.86	2279	592
Denver	2.5 (f)	5.03	2688	8731
	2.5 (w)	17.38	2137	572
	2.5 (s)	27.57	2849	700
Eagle Pass	1.5 (f)	2.23	1564	4781
	1.5 (w)	4.24	1214	315
	1.5 (s)	8.1	1578	454
Grand Canyon	15 (f)	4.5	2822	8621
	15 (w)	12.87	2073	488
	15 (s)	37.96	3115	844
Jackson	0.5 (f)	2.44	1297	4084
	0.5 (w)	2.6	859	231
	0.5 (s)	3.62	1127	296
Moab	15 (f)	4.03	2561	8082
	15 (w)	10.27	1836	495
	15 (s)	21.09	2430	628
Seattle	5 (f)	5.28	2671	8365
	5 (w)	9.70	2011	486
	5 (s)	26.75	2763	672

will be able to run much faster. But perhaps more significant is the comparison of memory requirements. On average, Franklin's algorithm used more than an order of magnitude the memory our algorithms require.

Color plate 1 and Color plate 2 show rendering examples of real terrain rendered with both Franklin's and our algorithm. Our algorithm generates noticebly larger polygons.

5 Conclusions and Future Work

We have presented a new method to generate Triangular Irregular Networks (TINs) from dense terrain grids. Our algorithm differs from previous methods in its use of a bottom-up approach to terrain sampling. Its key features include:

- *Low Complexity Output TIN.* Our method generates very few triangles for a given ϵ. Indeed, a primary objective in using the greedy optimization step is the minimization of the number of triangles in the output.

- *Memory Efficiency.* It can be run on very large terrains, potentially even those whose grids cannot simultaneously fit in memory.

- *Maintenance of Structural Fidelity.* Our method is able to maintain with very little additional overhead any pre-specified set of features of the terrain, without the need for adding additional (Steiner) points.

- *Speed.* Our running times are comparable to the fastest available methods, and we can probably improve the performance dramatically with a careful refinement of our code.

Our experimental results so far have focussed on the quality of the output TIN. The running time can certainly be improved through more careful coding. Also, further experimentation with the heuristics, especially the greedy biting operation, should yield even better results with respect to the output size. On the theoretical side, we are also attempting to prove worst-case bounds on the performance of the approximation (e.g., that we obtain a number of triangles that is guaranteed to be within a small factor of optimal).

A straightforward modification of our code will permit the algorithm to work on arbitrary TIN terrain inputs, rather than just on DEM arrays. Conceptually, there are no changes needed to the algorithm. A somewhat less trivial modification will be to generalize the algorithm to approximate arbitrary (non-terrain) polyhedral surfaces and to find approximations to a minimum-facet separating surface (as done in [2, 3, 17], in the convex case).

Another straightforward extension of our method allows one to use it to build hierarchical representations of terrain. For example, we can simply start with an extremely crude terrain approximation (e.g., just two triangles), and then adjust ϵ to be smaller and smaller, making each corresponding TIN a refinement of the previous one, until we have the full resolution grid. An ideal such hierarchy would have logarithmic height, as the intermediate TINs have sizes 2, 4, 8, 16, etc.

Another extension that we are pursuing is to approximate functions (terrains) of three variables. Approximating such functions is very important in scientific visualization. One can apply our same paradigm to this problem, biting off tetrahedra that satisfy the ϵ-fitness criterion. The tricky issue in implementing this algorithm is in maintaining the regions $\mathcal{P}$ of *un*tetrahedralized domain, since this will be a polyhedral space, possibly of high genus.

Acknowledgements

A. Kaufman is partially supported by the National Science Foundation under grants CCR-9205047 and DCA 9303181 and by the Department of Energy under the PICS grant. J. Mitchell is partially supported by National Science Foundation grant CCR-9204585, and by grants from Boeing Computer Services and Hughes Aircraft. C. Silva acknowledges partial support from CNPq-Brazil under a PhD fellowship. We

thank Martin Held for supplying us terrain data and a program that decodes the DEM terrain datasets. We thank Pat Crossno, Juliana Freire, Paul Heckbert, and Amitabh Varshney for their comments on the paper. Special thanks to Wm. Randolph Franklin for making his triangulation code freely available on the Internet.

References

[1] P. K. Agarwal and S. Suri. Surface approximation and geometric partitions. In *Proc. Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 34–43, 1994.

[2] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite VC-dimension. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 293–302, 1994.

[3] K. L. Clarkson. Algorithms for polytope covering and approximation. In *Proc. 3rd Workshop Algorithms Data Struct.*, volume 709 of *Lecture Notes in Computer Science*, pages 246–252, 1993.

[4] G. Das and D. Joseph. Minimum vertex hulls for polyhedral domains. *Theoret. Comput. Sci.*, 103:107–135, 1992.

[5] Gautam Das and Michael T. Goodrich. On the complexity of approximating and illuminating three-dimensional convex polyhedra. In *Proc. 4th Workshop Algorithms Data Struct.*, Lecture Notes in Computer Science. To appear. Springer-Verlag, 1995.

[6] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C26–C27, v 1995.

[7] L. De Floriani. A pyramidal data structure for triangle-based surface representation. *IEEE Comput. Graph. Appl.*, 9:67–78, March 1989.

[8] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics*, 13(2):199–207, August 1979.

[9] W. R. Franklin. Triangulated irregular network to approximate digital terrain, Section 2.3, Research Interests. Technical report, Electrical, Computer, and Systems Engineering Dept., Rensselaer Polytechnic Institute, Troy, NY, 1994. Manuscript and code available on ftp://ftp.cs.rpi.edu/pub/franklin/.

[10] M. T. Goodrich. Efficient piecewise-linear function approximation using the uniform metric. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 322–331, 1994.

[11] M. T. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths via balanced geodesic triangulations. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 318–327, 1993.

[12] L. J. Guibas, J. E. Hershberger, J. S. B. Mitchell, and J. S. Snoeyink. Approximating polygons and subdivisions with minimum link paths. *Internat. J. Comput. Geom. Appl.*, 3(4):383–415, December 1993.

[13] J. Lee. A drop heuristic conversion method for extracting irregular network for digital elevation models. In *GIS/LIS '89 Proc.*, volume 1, pages 30–39. American Congress on Surveying and Mapping, Nov. 1989.

[14] J. Lee. Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models. *Intl. J. of Geographical Information Systems*, 5(3):267–285, July-Sept. 1991.

[15] J. S. B. Mitchell. Approximation algorithms for geometric separation problems. Technical report, Dept, of Applied Math, University at Stony Brook, Stony Brook, NY, July, 1993.

[16] J. S. B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.

[17] J. S. B. Mitchell and S. Suri. Separation and approximation of polyhedral surfaces. In *Proc. 3rd ACM-SIAM Sympos. Discrete Algorithms*, pages 296–306, 1992.

[18] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994. C code and errata available by anonymous ftp from grendel.csc.smith.edu (131.229.64.23), in the directory /pub/compgeom.

[19] L. Scarlatos. *Spatial data representations for rapid visualization and analysis.* Ph.D. thesis, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, 1993.

[20] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using terrain features. In *Proc. of the IEEE Conference on Visualization - Visualization '90*, pages 168–175. IEEE, 1990.

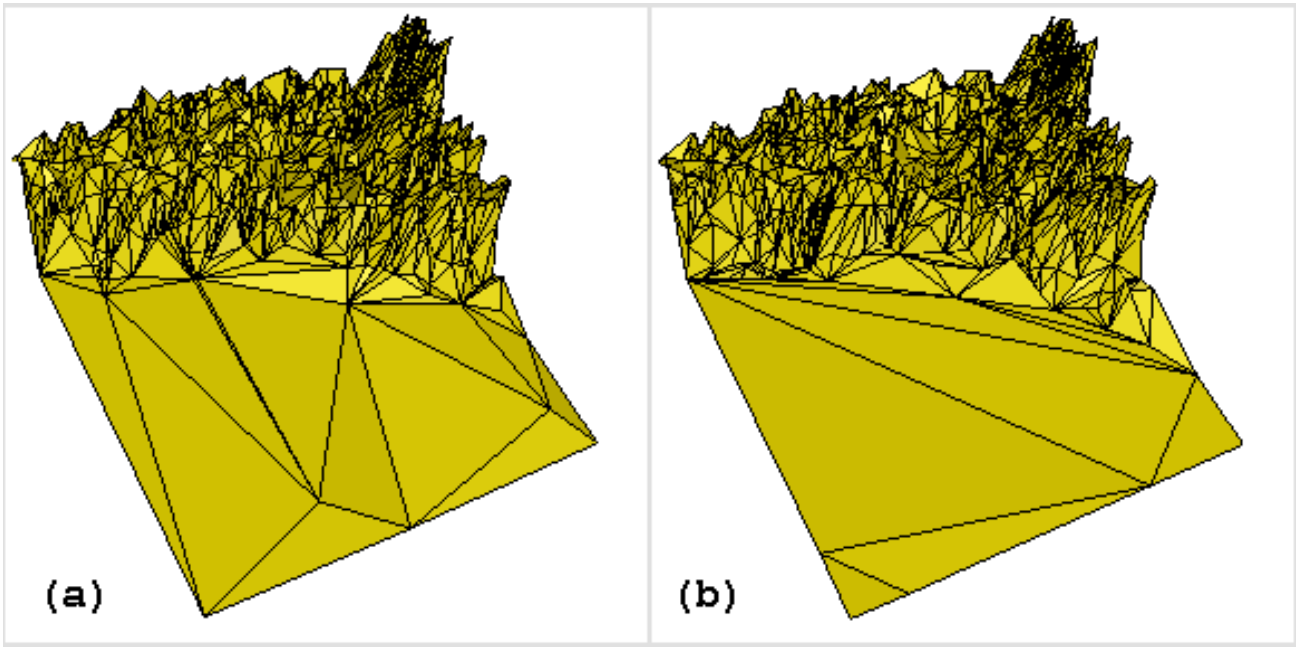
[21] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographics coherence. *CVGIP: Graph. Models Image Process.*, 54(2):147–161, March 1992.

[22] L. Scarlatos and T. Pavlidis. Optimizing triangulation by curvature equalization. In *Proc. of the IEEE Conference on Visualization - Visualization '92*, pages 333–339. IEEE, 1992.

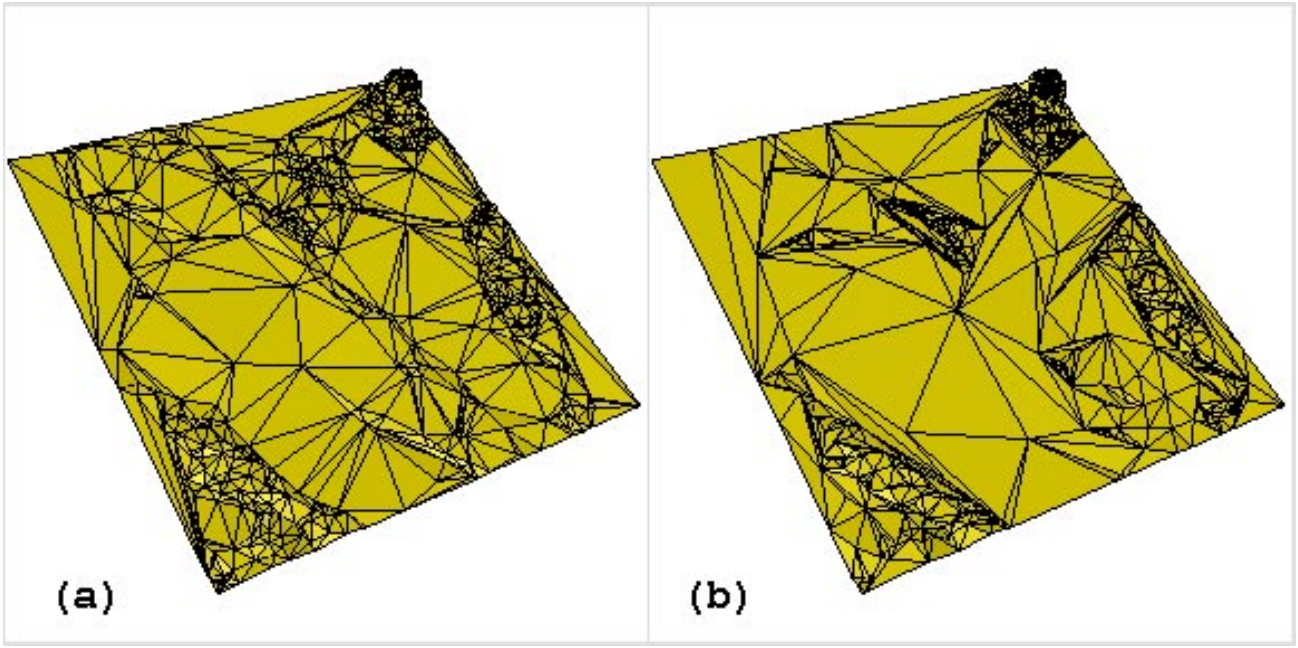
[23] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *SIGGRAPH '92*, volume 26, pages 65–70, July 1992.

[24] S. Suri. On some link distance problems in a simple polygon. *IEEE Trans. Robot. Autom.*, 6:108–113, 1990.

[25] A. Varshney. *Hierarchical Geometric Approximations.* Ph.D. thesis, Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175, 1994. TR-050-1994.



Color plate 1: *Buffalo terrain triangulated with (a) Franklin's algorithm, (b) our algorithm (strong-feasibility).*



Color plate 2: *Jackson terrain triangulated with (a) Franklin's algorithm, (b) our algorithm (strong-feasibility).*