

Greedy Cuts: An Advancing-Front Terrain Triangulation Algorithm

Cláudio T. Silva

180 Park Ave., Room D265

AT&T Labs-Research

Florham Park, NJ, 07932

csilva@research.att.com

Joseph S. B. Mitchell

Department of Applied Mathematics and Statistics

State University of New York at Stony Brook

Stony Brook, NY 11794-3600

jsbm@ams.sunysb.edu

Abstract

We propose advancing-front techniques for the problem of simplification of dense digitized terrain models. While most simplification algorithms have been based on either incremental refinement or decimation techniques, our *Greedy-Cuts* algorithms use a simple triangulation-growth procedure. They work by taking greedy cuts (“bites”) out of a simple closed polygon that bounds a connected component of the yet-to-be triangulated region. The method begins 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.

In this paper, we present both the basic Greedy-Cuts framework (which has been introduced in our earlier paper) and a new enhancement of the Greedy-Cuts method that improves the quality of the resulting triangulation. This improvement is made possible through the maintenance of *two* “fronts”, a real front and a virtual front, that bound between them a region of the terrain that has only a tentative triangulation. By allowing simple local operations (edge collapses and edge flips) in the tentative triangulation, we are able to avoid many of the artifacts of the basic Greedy-Cuts advancing-front technique, while not significantly affecting memory usage or running time.

Our implementation of Greedy-Cuts, as well as its multi-front enhancement is publicly available in the `GcTin` system. We give experimental evidence of the effectiveness of the multi-front enhancement to the Greedy-Cuts method and show that our method is competitive with current algorithms in terms of running time. One of the major advantages of our implementation is that it requires very little memory beyond that for the input height array.

1 Introduction

The problem of triangulating dense terrain models, while approximating them to within a user-specified error bound, is fundamental to several GIS applications, and has been studied actively since the early 1970's. Many algorithms have exploited the good quality properties of Delaunay triangulations; e.g., one standard “refinement” approach is based on building triangulations incrementally by inserting vertices into a coarse triangulation while maintaining the Delaunay property (e.g., see [8]). In contrast with refinement methods, which start with a coarse triangulation and refine it, decimation techniques start with a fine triangulation (the original data) and iteratively remove selected points, resulting in a coarsening of the triangulation [17, 18, 27]. We refer the readers to the recent survey of Garland and Heckbert [15] or of van Kreveld [31] for an extensive discussion of prior work.

There are several tradeoffs between incremental refinement and decimation techniques. If we seek a coarse approximation, incremental refinement methods tend to converge faster, since they start already with a very coarse triangulation. On the other hand, very accurate approximations, with a low error tolerance, are computed most directly using a decimation procedure, starting from the full resolution data. Memory consumption is an important issue in designing triangulation methods. Most of the current techniques need to keep fairly large data structures in memory, since they rely on having the triangulation in memory at intermediate stages of the algorithm.

Largely due to its low memory consumption, we have investigated “advancing-front” techniques for terrain triangulation. In short, advancing-front techniques are based on incrementally triangulating the terrain, one triangle at a time, at its final resolution, while advancing a “front” across the data. The front is a set of polygonal curves that represent the boundary between the already triangulated region and the yet-to-be triangulated region. The memory consumption is low, since we only have to store a representation of the front in memory; as triangles are created, they can be output (e.g., written to a file). Another potential advantage of advancing-front methods is that they lend themselves readily to being able to handle *structural fidelity* constraints (e.g., river, road, and fault line boundaries), by insisting that these edges appear as edges within the output triangulation, while still respecting the error bounds. The method also readily permits one to partition the data, possibly specifying a different error bound in different regions of the terrain; this

may be useful in applications requiring real-time triangulation.

This paper presents our experience in devising and implementing advancing-front techniques for terrain surfaces. We begin with a detailed discussion of our basic Greedy-Cuts algorithm, as was first reported in an earlier conference paper [29]. While we have found that the basic algorithm has favorable properties in terms of memory consumption and total number of triangles, it is much less effective at producing high *quality* triangulations from the point of view of angles that are close to zero or close to π . Thus, we have developed an enhanced version of Greedy-Cuts (as first reported in the conference version of this paper [28]) that addresses the main weakness of our earlier advancing front technique – triangle quality. The potential for low triangle quality is an intrinsic shortcoming of advancing front methods that do not permit backtracking in their decisions. The novel feature of our new algorithm is that we maintain two “fronts”, a real front, and a virtual “back” front, and the tentative triangulation of the region in between. By allowing edge collapses and flips in the region between the fronts, we are able to obtain much higher quality triangulations, while preserving the low-memory feature of the advancing-front method.

2 Background and Related Work

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 can contain an extremely high level of redundancy, which, in turn, can forbid 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 [23, 24]):

- *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 (although he does use a local edge-swapping heuristic in order to prefer quality triangulations). A new version of his code is publicly available, and we used it for comparison with our method. A detailed description of his algorithm and code is given in Section 5. Recently, substantial research has been conducted on creating hierarchical structures on top of TINs [7, 25], and on techniques to improve the quality of TIN meshes [26]. Scarlatos’ dissertation [23] 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 [17, 18] 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* [27] 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 [21], where a greedy set cover approach has been developed for approximating convex surfaces, and used recently by Varshney [32] 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) [4, 5], 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, 19, 21]), 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, 21, 32]) is the same greedy principle that is used to compute minimum-link paths in simple polygons. This problem is well studied in computational geometry [14, 20, 30] and can be used to find an optimal piecewise-linear approximation to a function of a single variable (see [11]). 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 [21] 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.

3 The Greedy-Cuts Triangulation Algorithm

This section describes the basic Greedy-Cuts algorithm in some detail; see also [29]. 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 (in our basic algorithm; the multi-front enhancement will include a limited amount of connectivity). 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 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 $S^{-\varepsilon}$ and completely below the surface $S^{+\varepsilon}$, where $S^{-\varepsilon}$ (resp., $S^{+\varepsilon}$) 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.

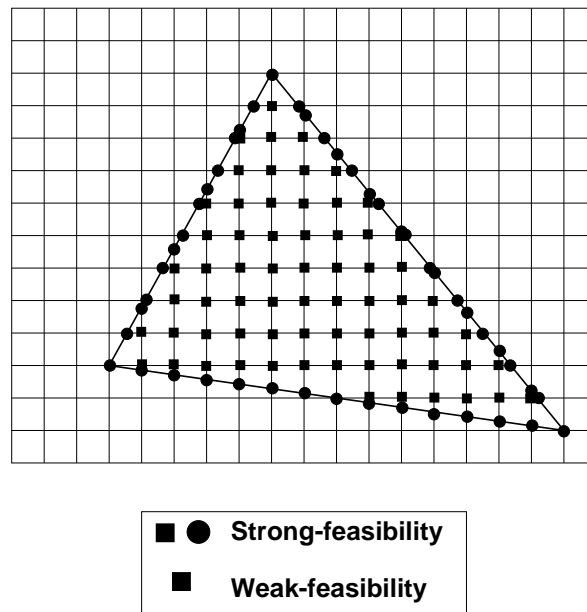


Figure 1: Weak and strong feasibility.

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-1}, v_{i+1}) 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. See Figure 3. 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-1}, v_{i+1}) is an ear of a simple k -gon [12], but the simple linear-time method is likely to be more practical (since k is typically small) and is what we currently have implemented. (A potentially faster implementation may be based on the use of simple hashing schemes, as is done in the ear-clipping-based triangulation code of Held [16].)

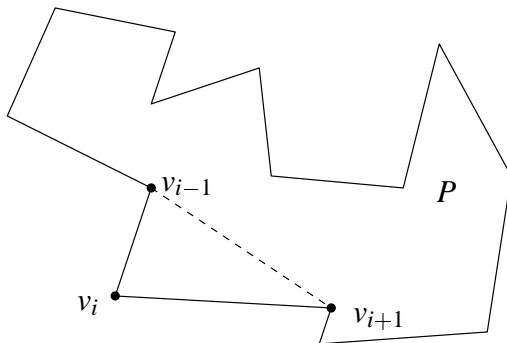


Figure 2: Illustration of ear cutting: (v_{i-1}, v_{i+1}) is a valid diagonal, so the ear with tip v_i can be clipped.

Each cut we perform lowers the complexity (number of edges) of polygon P by one, thereby 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 replace obtuse angles with 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 tends to happen 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, our currently implemented algorithm searches for such a point v among a select set of candidates, as follows. We search grid points that are approximately along a line that is perpendicular to (v_i, v_{i+1}) at the midpoint of (v_i, v_{i+1}) , using 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). See Figure 3.
- *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 .

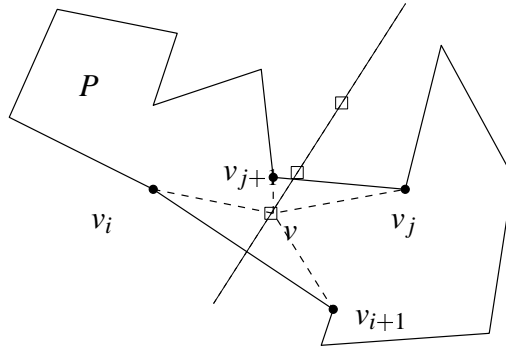


Figure 3: Illustration of greedy biting: The search for a valid bite tests the points denoted by hollow squares until the point v is found, resulting in triangle (v_i, v, v_{i+1}) . Since the nearby edge (v_j, v_{j+1}) defines a feasible triangle with v , we also output the triangle (v_j, v, v_{j+1}) and split P into two simple polygons.

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 [30], applied to the discrete data points between the offset curves obtained by shifting the terrain surface up/down by ϵ . See Figure 4.

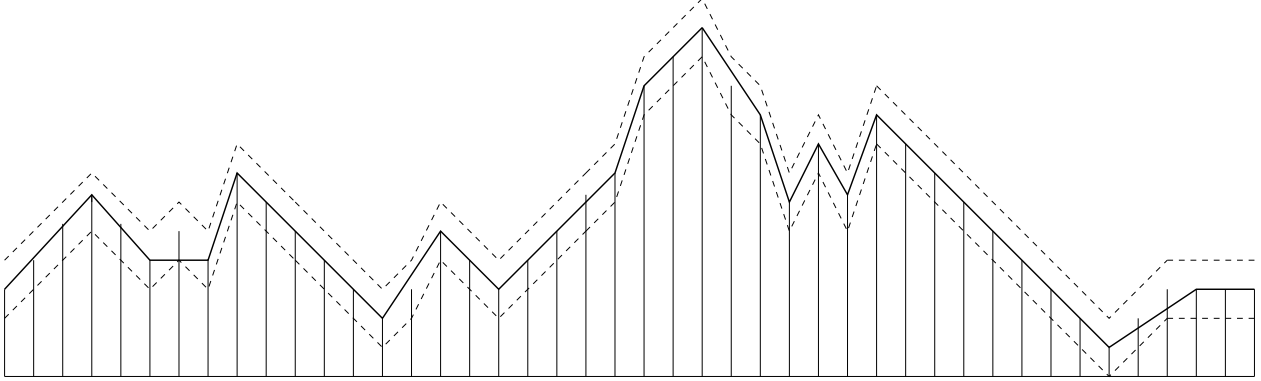


Figure 4: *The vertical solid lines denote the heights at points along one boundary segment of R . The dashed polygonal chains show the result of shifting the data points up/down, creating an envelope of ϵ -feasible curves. The solid line is calculated by the simple greedy method, at each step linking to the data point that extends the approximation as far as possible, while staying within the dashed envelope.*

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 .

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 Ears Theorem” [22].) 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 algorithm, 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

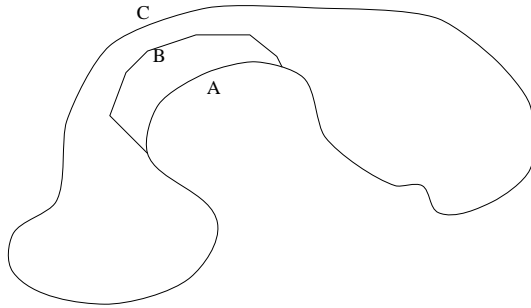
that rely on triangulation refinement must maintain some sort of topological data structure for the full set of triangles. Typically, one would expect that if the output size (number of triangles) is k , then the boundary of the polygons \mathcal{P} at any given instant will have roughly size \sqrt{k} .

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.

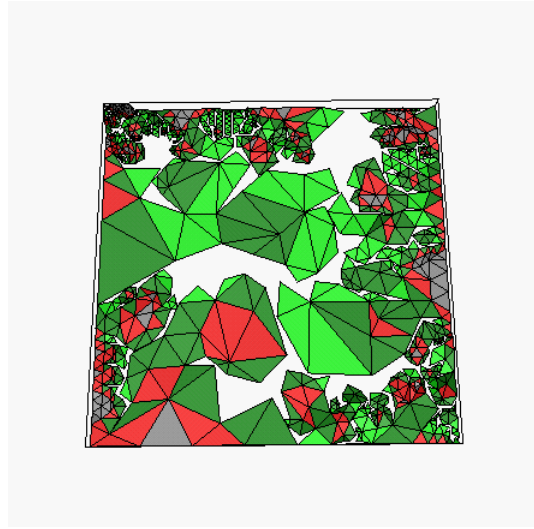
The output complexity for our algorithm is again hard to estimate from a theoretical point of view. The problem we are trying to solve approximately is known to be NP-hard, in general. Thus, the best we can hope for is that we may be able to prove a worst-case bound on the ratio of our output size (number 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 The *Multi-Front Greedy-Cuts* Triangulation

The main shortcoming of the basic Greedy-Cuts triangulation algorithm described in the previous section is that it is not possible to backtrack at all during the generation of triangles: Each triangle that is generated is committed, with no possibility to modify it later. This can lead to poor triangle quality, particularly in the case of “bottlenecks,” as illustrated in Figure 5. The bottleneck problem happens when one portion of the boundary interferes with the progress of the triangulation near another portion of the boundary. The bottleneck problem may be caused by two portions of the front coming close, while having substantially different sampling rates (resolutions) – one side is a polygonal curve having vertices placed much more closely along the curve than those on the other side. Then, it is difficult to complete the triangulation without doing many splits or creating very skinny triangles. The bottleneck problem can also arise if the sampling rates are comparable, but



(a)



(b)

Figure 5: Illustration of the bottleneck problem. (a) The “C” part of the boundary of the front, interferes with the “A” part as it attempts to grow. Instead of the triangulation being able to close neatly with few triangles being generated, the curve “A” will approach “C”, by closer and closer curves, such as “B”, generating a large number of small and badly shaped triangles. This can happen even if the terrain is perfectly flat in the neighborhood of the subcurves. (b) Snapshot (perspective projection) of a partial triangulation depicting the bottleneck problem in practice.

the portions of the front have gotten so close together in forming the bottleneck that no ear cuts or high-quality bites are possible.

In fact, any advancing front technique may suffer from the bottleneck problem, and related issues, since decisions that are made early in the triangulation process may force the algorithm into a difficult situation to resolve later. Incremental refinement and decimation methods avoid this issue by being “global” algorithms that are allowed to make changes anywhere within the triangulation.

Our new technique is based on a hybrid approach, which attempts to exploit the advantages of both the (local) advancing-front approach and the (global) refinement/decimation methods. We accomplish this by providing a simple and efficient *partial* backtracking mechanism for Greedy-Cuts, which allows the quality of the triangulation to be improved as the algorithm progresses,

giving a means of keeping a good triangulation throughout the execution. In order to keep the memory complexity low, we allow for only a limited amount of backtracking. In particular, we consider the terrain to be partitioned into three types of regions: regions with a *final* triangulation, regions with a *tentative* triangulation, and the yet-to-be triangulated regions. The region with a tentative triangulation is kept “small,” including only those triangles that are adjacent to vertices on the (true) front. The boundaries between the three types of regions are determined by *two* “fronts” – the usual front (which we will simply call the *front*), delineating the boundary between the triangulated region and the yet-to-be triangulated region, and a second front (which we will call the *back front*), delineating the boundary between the tentative triangulation and the final triangulation. The tentative triangulation lies in the region between the front and the back front; we can think of the back front as “lagging behind” the front, in the expansion of the region that we triangulate. Since the tentative triangulation that we maintain is very small (proportional to the complexity of the front), we are able to preserve the low memory overhead of the advancing-front technique.

Data Structures. Instead of explicitly keeping the two fronts and the tentative triangulation, we only keep a list of associated vertices and triangles. The `Triangle` and `Point` data structures are (roughly) as follows:

```
typedef struct point {
    Point2    position;
    int       refCount;
    int       nTriangle;
    Triangle *triList;
} Point;

typedef struct triangle {
    struct point *p[3];
    int         refCount;
} Triangle;
```

The vertices are instantiated only during greedy bites (which now include the edge split operation). When a triangle is created, it is not immediately written to the output; initially, it is

considered to be tentative (we say that it is *active*), and it is flushed to the output only when all of its corresponding vertices no longer belong to the outer front. Each triangle has a pointer to each of its corresponding vertices, and also an independent reference count (`refCount`). Also, each vertex has pointers to each triangle in its “use set” (also known as the “star” of the vertex), `triList`. When a vertex is “output” in the ear cutting phase (see below), each of the triangles in its use set is “dereferenced” (its `refCount` is decremented by one); similarly, when a triangle is dereferenced, each of its vertices is dereferenced (by decrementing `nTriangle`). When the reference count (`refCount`) of a triangle hits zero, its storage can be safely reclaimed and it can be written to a file. Since a vertex can be on more than one connected component of the front, a second reference count (`refCount`) is used to keep track of the number of active boundary components containing the vertex. A vertex is written to the output when its `refCount` hits zero.

Triangle Quality. In generating our triangles, instead of simply using a greedy selection, as in the previous section, we now enforce a quality criterion based on Gueziec’s notion of “compactness” [13]. Given a triangle with edge lengths l_0, l_1, l_2 , the compactness measure g is given by

$$g = \frac{4\sqrt{3}A}{l_0^2 + l_1^2 + l_2^2}, \quad (1)$$

where A is the (positive) area of the triangle. Note that $0 \leq g \leq 1$, and as g gets closer to 1, the triangle gets closer to an equilateral triangle. Basically, when generating new triangles, we give preference to ear cuts and greedy bites that result in a triangle whose compactness is close to 1 (within a user-specified tolerance).

Ear Cutting. The ear cutting procedure has two new features:

- (1) We compute the compactness measure on each candidate ear triangle.
- (2) Before performing an ear cut, we first attempt to advance the front by an *edge collapse* operation on an edge of the front, in which one front vertex is moved on top of another front vertex, and the incident edges are adjusted accordingly. This edge collapse is considered to be feasible only if the resulting new triangles also meet the quality standard.

In Figure 6a, we illustrate a standard ear cut, which generates triangle (p, q, r) , while causing vertex q to be removed from the active list. Figure 6b illustrates the result of performing instead an edge

collapse on (q, r) . This edge collapse is possible, using our data structures, because we have not discarded any triangles that contain a vertex that still belongs to the active front boundary. Note that the edge collapse operation saves the creation of a new triangle. We will also see later that this new local edge collapse operation during an ear cut works nicely in concert with the new edge split operation.

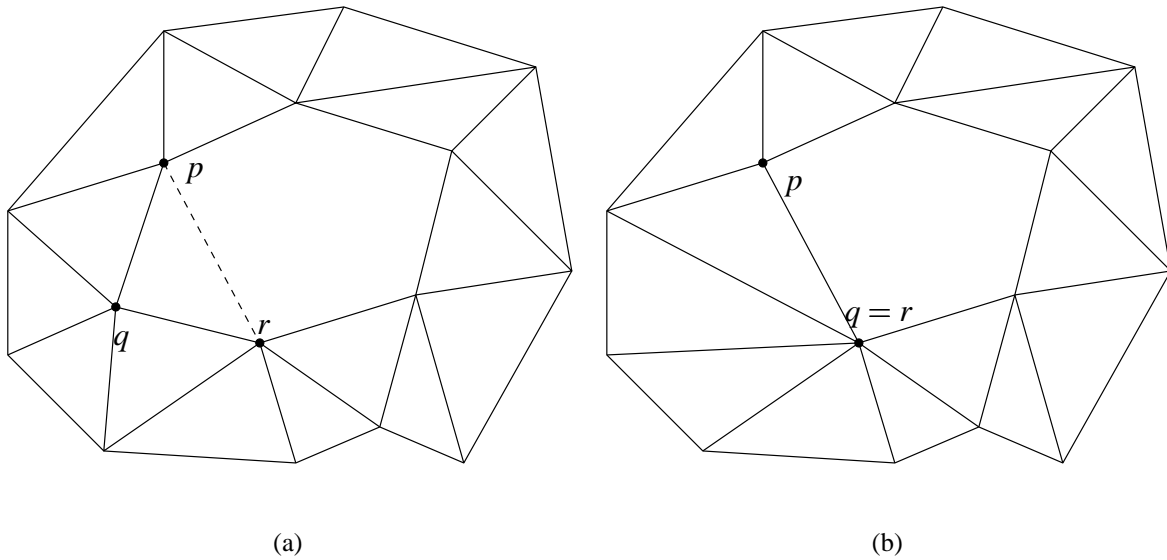


Figure 6: (a). A standard ear cut results in the addition of the edge (p, r) and the creation of the new triangle (p, q, r) . (b). An edge collapse, of edge (q, r) , results in the triangulation shown.

Greedy Biting and Edge Splitting. As with ear cutting, our new greedy biting procedure has some new features:

- (1) We compute the compactness measure on each candidate triangle.
- (2) We integrate now the edge splitting process into the greedy biting, as follows: If there is not a “good” bite (according to the quality measure) from a base edge e that is on the front, then we automatically perform an edge split on e . (The rationale is that if e is unsuitable for biting now, then it is “too long,” in a sense; since it will remain unsuitable as the algorithm progresses, we may as well do the split now.) An edge split involves creating a new vertex m near the middle of the edge $e = (p, r)$; however, now that we have available to us the triangles

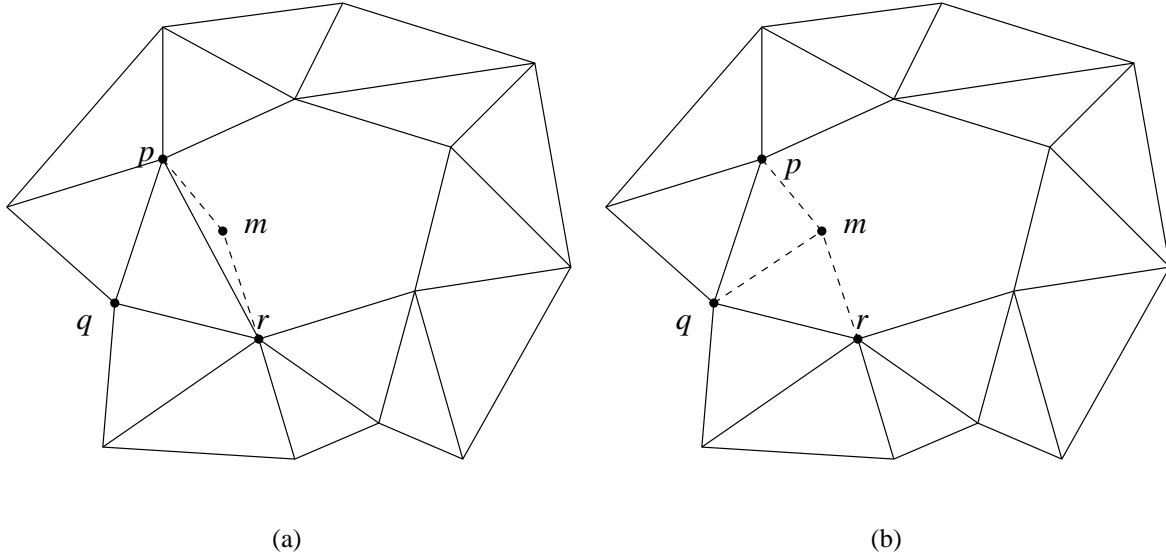


Figure 7: Edge splitting: If no feasible bite is possible from the base edge $e = (p, r)$, then e is split at a nearby point m . The prior Greedy-Cuts method creates a very skinny triangle (p, m, r) , as in (a), since there was not an option to change the existing triangle (p, q, r) . Now, we allow an edge swap to take place, removing (p, r) , and adding (q, m) instead, as in (b).

that are incident on the front, we are able to perform an *edge swap* in conjunction with the edge split, allowing us to avoid creating very skinny triangles in the process. See Figure 7. Note that the split only happens if triangles (p, q, m) and (q, m, r) are feasible.

- (3) The polygon splitting that takes place in the previous section when a bite results in a new vertex close to an existing (opposite) front edge is now replaced by an edge collapse operation.

When generating a new triangle by greedy biting, our algorithm attempts to avoid creating a triangle (even a nicely shaped triangle) that leaves behind small angles in the front, as these will end up forcing small angles later in the triangulation process (e.g., by way of an ear cut). For example, in Figure 8, we may avoid using vertex a to create a triangle with base qr , even though the triangle (q, a, r) is almost equilateral, because the angle (p, q, a) might be too small; we may prefer creating triangle (q, b, r) in this situation.

Initial Boundary Smoothing. During the initialization phase of the basic Greedy-Cuts, we perform a curve fitting for the boundary of the terrain, finding a minimum-link approximation of the

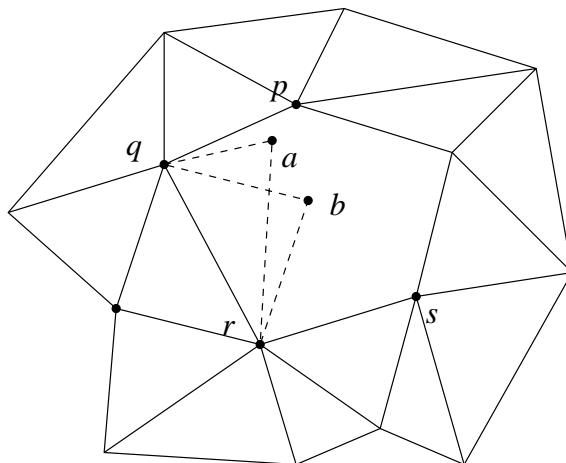


Figure 8: Angle considerations in selecting a new triangle. A quality measure is used in order to bias the algorithm in favor of nicely shaped triangles during triangulation.

boundary, subject to the error tolerance ϵ . (Recall Figure 4.) The resulting approximate boundary serves as an initial front, which is then advanced inwards. Unfortunately, this level of greediness has the undesirable effect of potentially oversimplifying the boundary, making it difficult later to utilize high-quality triangles to triangulate between the boundary and some nearby portion of higher terrain complexity. Thus, in our new algorithm we perform a “smoothing” operation on the simplified boundary. Specifically, we split edges as needed in order to have a bound on the ratio of the length of any one boundary edge and the length of its predecessor or successor edge along the boundary (i.e., so that $l_i/l_{i+1} \leq A$ and $l_{i+1}/l_i \leq A$, where l_i is the length of the i th edge in the approximating chain, and A is a parameter controlling the degree of smoothing). This procedure ensures a logarithmic scale on the size of the edges.

5 Experimental Results

Our Greedy-Cuts methods are 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 [22], 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 algorithms described here. 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 [9] is an incremental refinement method, with additional diagonal swapping heuristics included in order to generate hopefully good quality triangles. Initially, the algorithm approximates the DEM by 2 triangles. Then, a general step of the algorithm involves finding the most deviant point within each current 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 to improve the quality of the triangles. The original code works by performing a predetermined number of splits. We have modified slightly 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.

Delaunay-based Triangulation and Scape. We use the code written by Garland and Heckbert [10] which implements an extremely fast version of the algorithm originally proposed by Fowler and Little [8]. This provides us with another comparison point for the algorithms proposed in this paper.

Experimental Setup. Our experiments were conducted on a Silicon Graphics O2, equipped with one R5000 processor and 192MB of RAM.

In Table 1, we show the results of running on seven real terrain datasets each of the four

Terrain	ϵ	Franklin	GcTin	MF-GcTin	Scape
Buffalo	2.5	2K (1.1s)	2.2K (7.5s)	2.8K (5s), 2.8K (6s)	2K (0.2s)
Denver	2.5	2.7K (2s)	2.8K (9.6s)	3.4K (6.8s), 3K (5s)	2.6K (0.2s)
Eagle Pass	1.5	1.5K (1s)	1.6K (3s)	2.1K (3.2s), 1.7K (2s)	2K (0.2s)
Grand Canyon	15	2.8K (2.2s)	3.1K (13.7s)	3.7K (9.6s), 3.3K (7s)	2K (0.2s)
Jackson	0.5	1.4K (1.1s)	1.1K (1.3s)	1K (2.5s), 1.5K (1.8s)	3.1K (0.3s)
Moab	15	2.6K (1.8s)	2.4K (7.4s)	2.5K (5s), 2.6K (4.8s)	2.5K (0.2s)
Seattle	5	2.7K (2.8s)	2.7K (9.5s)	2.5K (4.8s), 3K (5s)	2.4K (0.2s)

Table 1: Experimental results of approximations, triangle counts, and running times: The triangle counts for the new algorithm are shown with strong and then weak feasibility.

algorithms: Franklin’s algorithm, the original GcTin, our new MF-GcTin algorithm, and Garland and Heckbert’s Scape. The table shows the choice of ϵ , and the total number of triangles in the output TIN, for each of the seven terrains. All the input terrains were 120-by-120 elevation arrays. See Figure 9(a)–(d) for screen shots of partial triangulations of the Denver terrain during the running of the new algorithm.

GcTin with *weak-feasibility* results in a lower triangle count than Franklin’s code, in all instances. Here, we are applying *strong-feasibility* with GcTin and both strong and weak feasibility with our new algorithm. In terms of triangle count, when using strong feasibility both GcTin and MF-GcTin are showing a slightly higher triangle count than Franklin’s, which essentially uses weak feasibility. However, note that the triangle counts of our new algorithm, MF-GcTin, under *weak* feasibility are substantially lower than those under strong feasibility, and compare very favorably to those of Franklin’s algorithm. Note, however, that with our new algorithm, making direct comparisons is somewhat complicated by the additional triangle quality parameters (bound on g). The results in Table 1 are based on using $g = 0.5$. Overall, we have been able to improve on both the number of triangles generated as well as their number, with respect to the original GcTin code.

The Scape code is by far the fastest. Often, it generates triangulations with very similar triangle counts as the other ones, although sometimes, such as for the Jackson terrain, it generates

Terrain	ϵ	Franklin		GcTin		MF-GcTin		Scape	
		μ	σ	μ	σ	μ	σ	μ	σ
Buffalo	2.5	0.74	0.22	0.64	0.29	0.76	0.20	0.77	0.17
Denver	2.5	0.73	0.23	0.64	0.29	0.75	0.21	0.77	0.17
Eagle Pass	1.5	0.72	0.22	0.60	0.30	0.75	0.21	0.75	0.18
Grand Canyon	15	0.71	0.24	0.61	0.30	0.77	0.20	0.75	0.18
Jackson	0.5	0.63	0.27	0.54	0.33	0.67	0.28	0.70	0.20
Moab	15	0.70	0.23	0.59	0.32	0.73	0.21	0.74	0.18
Seattle	5	0.71	0.24	0.63	0.30	0.76	0.20	0.75	0.18

Table 2: Summary of triangle quality measures. We report the median μ and standard deviation σ for the triangulations shown in Table 1.

more than 3 times as many triangles. We believe this is due to the Delaunay constraint which forces the triangulation to have strict (projected) shape. Note that for other meshes, the Delaunay triangulations yields much fewer triangles, such as for the Grand Canyon dataset. The triangulation quality of the Delaunay triangulation is quite good, and quite similar to MF-GcTin.

Memory Usage. The memory cost of MF-GcTin is about the same as GcTin, with only a very slight increase, since we are storing a small number of (tentative) triangles. By instrumenting `malloc()`, we were able to determine that Franklin’s code uses between 13 to 17 times as much memory as GcTin; thus, our techniques use an order of magnitude less memory than Franklin’s code. Scape uses quite a bit of memory; Garland and Heckbert [10] report that for generating an approximation with m (output) points of a mesh with n (input) points, their algorithm requires $3n + 292m$ bytes. That is, the memory cost greatly increases with the size (and accuracy) of the output. We should note that our algorithms do not have any significant memory cost related to the size of the output, thus making it possible to generate accurate triangulations of very large terrains with little memory overhead.

Triangle Quality Measures. Figs. 13–18 show histograms of Gueziec’s quality measure for six (of the seven) terrains. Table 2 summarizes the results. It is clear that the new algorithm gives a substantially better distribution of triangle quality than either Franklin’s algorithm or our basic `GcTin`. In particular, we can see that the number of good triangles (the major peak) is always higher in the (c) column. In fact, the new algorithm gives triangle quality comparable to the Delaunay method (`Scape`), while using far less memory and, on some datasets (e.g., Jackson terrain), using fewer triangles.

In Figure 10 we compare Franklin’s algorithm to ours on a challenging dataset consisting of a cliff. Franklin’s algorithm fails to triangulate properly. A similar experiment on a “bump” is shown in Figure 11. Here, neither technique performs very well, but both results are acceptable, and `MF-GcTin` generates a larger number of well-shaped triangles. Finally, in Figure 12, we compare the Denver terrain triangulated with `GcTin`, and our new multi-front extension, `MF-GcTin`.

6 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, 21], 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, e.g., for intermediate TINs having sizes 2, 4, 8, 16, etc.

Our methods apply also to huge datasets that have been cut into smaller blocks. Specifically, along the boundary of each block we perform the curve fitting for the corresponding “slice” of the terrain, and use this to initialize the front propagation into each of the (up to) two blocks that share the boundary. If we want to permit edge swaps across block boundaries, in the spirit of MF-GcTin, then we can maintain (at a small increase in the total storage) that portion of the triangulation that is incident on the points along the block boundaries, and then perform local operations to improve this portion of the triangulation, either at the beginning of the front propagation, or as a final postprocessing step.

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 *untetrahedralized* domain, since this will be a polyhedral space, possibly of high genus.

Preliminary input from our current users have been very favorable on the released (basic) version of GcTin. We plan to release a new version of the software, with the new MF-GcTin algorithm, and several improvements (and new features) based on their comments. We would very much like to know more about how effective GcTin can be in real GIS applications.

The web site (<http://www.ams.sunysb.edu/~csilva/gctin.tgz>) contains the current GcTin code and will be kept up to date with future releases.

Acknowledgements

We have used GeomView, from the Geometry Center at the University of Minnesota, for generating some of the pictures for this paper. We thank Martin Held for supplying us terrain data and a program that decodes the DEM terrain datasets. Special thanks to Wm. Randolph Franklin, Michael Garland, and Paul Heckbert for making their triangulation code freely available on the internet. J. Mitchell acknowledges the generous support of Hughes Research Labs, the National Science Foundation (CCR-9732220), NASA, Northrop-Grumman, Sandia National Labs, Seagull Technology, and Sun Microsystems.

References

- [1] P. K. Agarwal and S. Suri. Surface approximation and geometric partitions. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 24–33, 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 Comput. Sci.*, pages 246–252. Springer-Verlag, 1993.
- [4] G. Das and M. T. Goodrich. On the complexity of approximating and illuminating three-dimensional convex polyhedra. In *Proc. 4th Workshop Algorithms Data Struct.*, volume 955 of *Lecture Notes Comput. Sci.*, pages 74–85. Springer-Verlag, 1995.

- [5] G. Das and D. Joseph. Minimum vertex hulls for polyhedral domains. *Theoret. Comput. Sci.*, 103:107–135, 1992.
- [6] M. de Berg and K. Dobrindt. On levels of detail in terrains. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C26–C27, 1995.
- [7] L. De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Comput. Graph. Appl.*, 9(2):67–78, Mar. 1989.
- [8] R. J. Fowler and J. J. Little. Automatic extraction of irregular network digital terrain models. *Comput. Graph.*, 13(2):199–207, Aug. 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. Garland and P. Heckbert. Fast Polygonal Approximation of Terrains and Height Fields. Technical Report CMU-CS-95-181. Manuscript and code available on <http://graphics.cs.uiuc.edu/~garland/CMU>.
- [11] M. T. Goodrich. Efficient piecewise-linear function approximation using the uniform metric. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 322–331, 1994.
- [12] 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.
- [13] A. Gueziec. Surface simplification with variable tolerance. In *Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery*, pages 132–139, 1995.
- [14] 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, Dec. 1993.

- [15] P. Heckbert and M. Garland. A Survey of Terrain Triangulation Algorithms. Technical report, Carnegie Mellon University, 1995.
- [16] M. Held. FIST-Fast Industrial Strength Triangulation of Polygons. *Algorithmica*, to appear 2000.
- [17] 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.
- [18] 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.
- [19] J. S. B. Mitchell. Approximation algorithms for geometric separation problems. Technical Report, Dept. Applied Mathematics and Statistics, SUNY Stony Brook, NY, July 1993.
- [20] J. S. B. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles in the plane. *Algorithmica*, 8:431–459, 1992.
- [21] 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.
- [22] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.
- [23] 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.
- [24] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using terrain features. In *Proceedings of the 1st 1990 IEEE Conference on Visualization, Visualization '90*, pages 168–175, IEEE Service Center, Piscataway, NJ, USA (IEEE cat n 90CH2914-0), 1990. IEEE.
- [25] L. Scarlatos and T. Pavlidis. Hierarchical triangulation using cartographics coherence. *CVGIP: Graph. Models Image Process.*, 54(2):147–161, Mar. 1992.

- [26] L. Scarlatos and T. Pavlidis. Optimizing triangulation by curvature equalization. In *Proceedings of the 3rd 1992 IEEE Conference on Visualization, Visualization '92*, pages 333–339. IEEE, 1992.
- [27] W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. *Comput. Graph.*, 26(2):65–70, 1992. Proc. SIGGRAPH '92.
- [28] C. Silva and J. Mitchell. Greedy Cuts: An Advancing-Front Terrain Triangulation Algorithm. In *Proc. 6th ACM International Symposium on Advances in Geographic Information Systems*, pages 137–144, 1998.
- [29] C. Silva, J. S. B. Mitchell, and A. E. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. In *Visualization 95*, pages 201–208, San Jose CA, 1995. IEEE Computer Society Press.
- [30] S. Suri. On some link distance problems in a simple polygon. *IEEE Trans. Robot. Autom.*, 6:108–113, 1990.
- [31] M. van Kreveld. Digital elevation models and TIN algorithms. In M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, editors, *Algorithmic Foundations of Geographic Information Systems*, number 1340 in Lecture Notes in Computer Science (tutorials), pages 37–78. Springer-Verlag, Berlin, 1997.
- [32] 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.

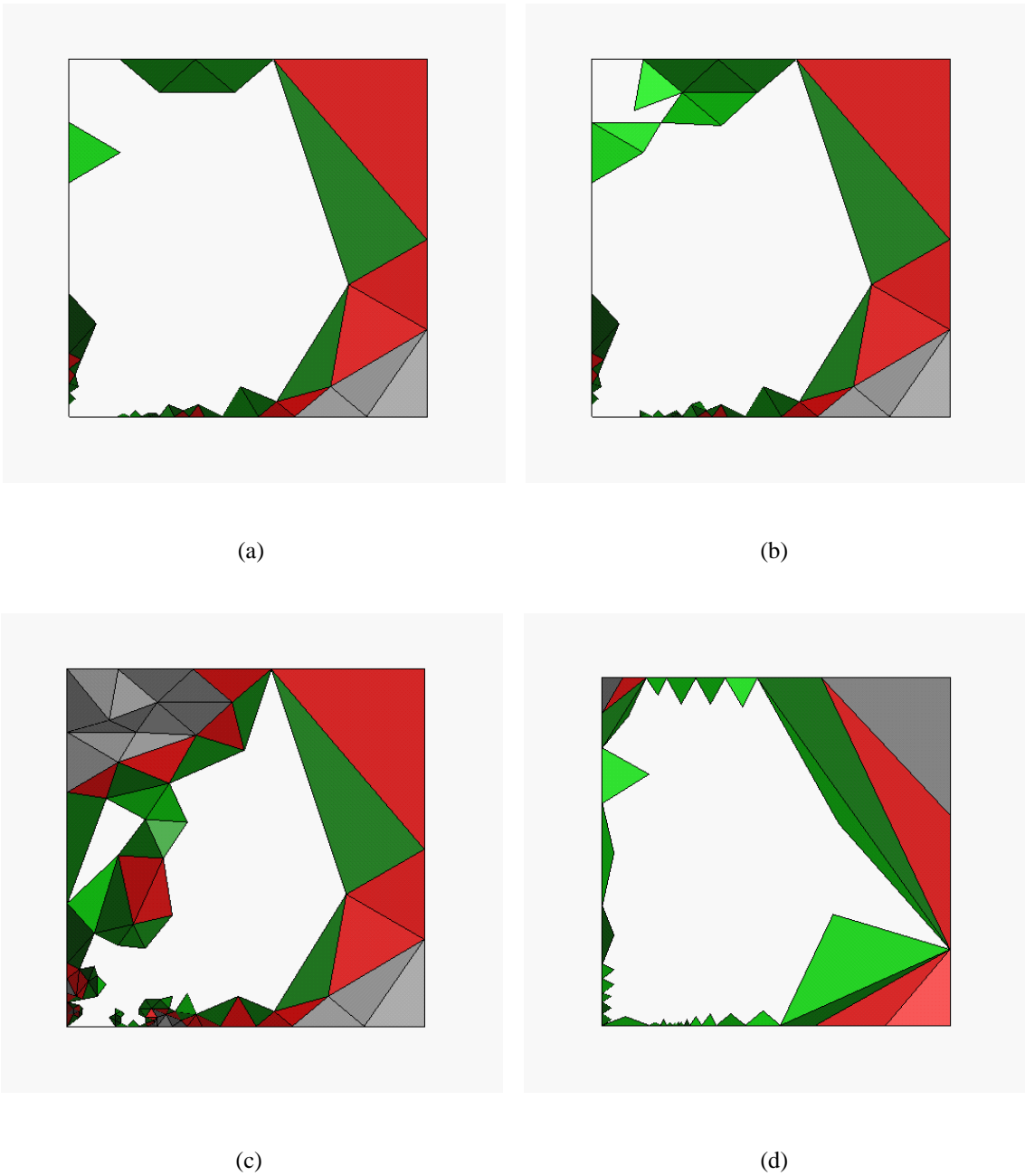
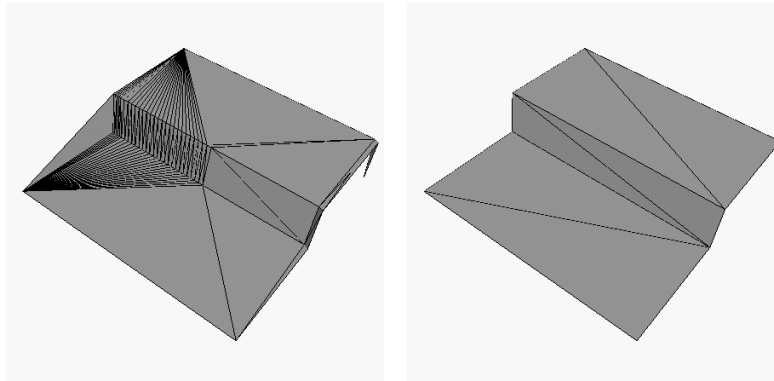


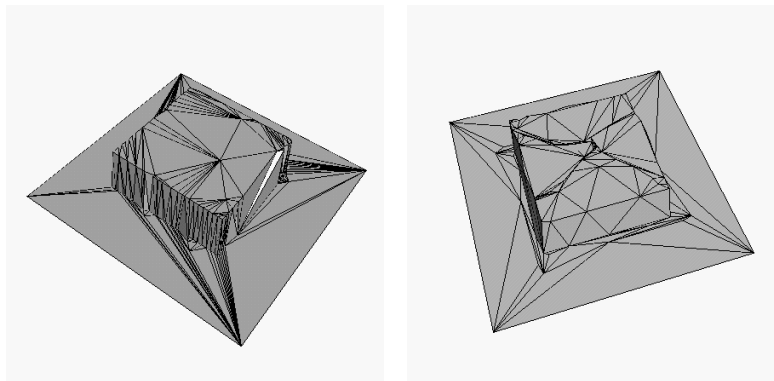
Figure 9: Screen shots during the triangulation of the Denver terrain, using the new algorithm. Colors correspond to the current state of a triangle: light green (3 vertices on the front), dark green (2 vertices on the front), red (1 vertex on the front), or gray (fully committed – no vertex on the front). (See color version on our web site.) (a). early stages of the algorithm; (b). at a split; (c). the triangulation around a split; (d). an early stage of the algorithm *without* quality measures imposed on triangles.



(a)

(b)

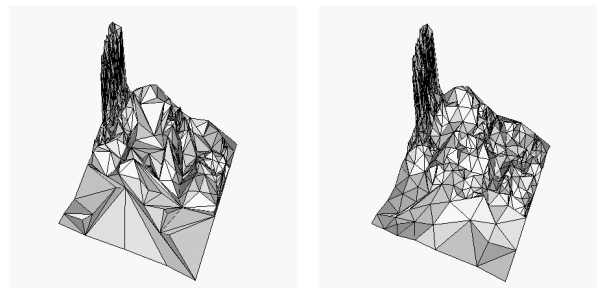
Figure 10: Quality comparison using a “Cliff” dataset. (a) Franklin’s algorithm. (b) MF-GcTin.



(a)

(b)

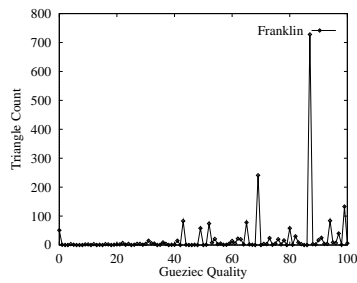
Figure 11: Quality comparison using a “Bump” dataset. (a) Franklin’s algorithm. (b) MF-GcTin.



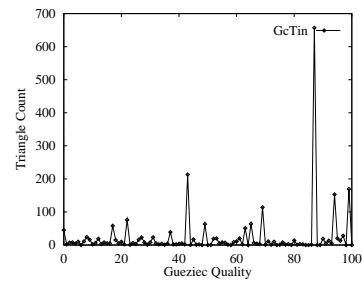
(a)

(b)

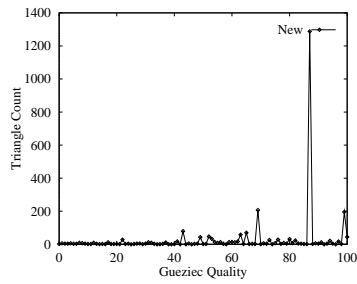
Figure 12: Quality comparison using the “Denver” dataset. (a) The basic GcTin algorithm. (b) The new multi-front technique, MF-GcTin.



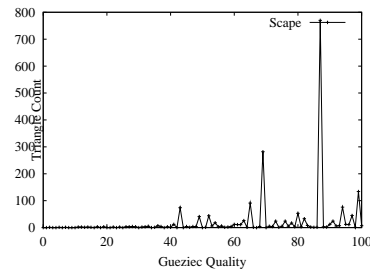
(a)



(b)

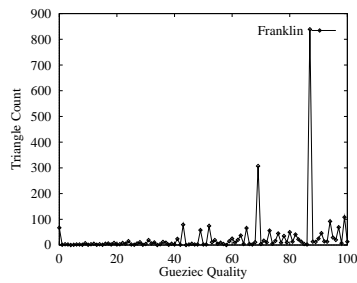


(c)

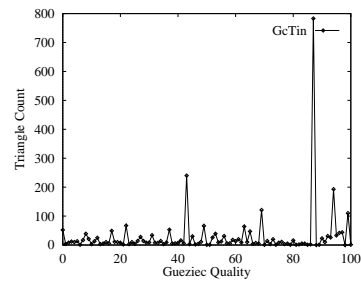


(d)

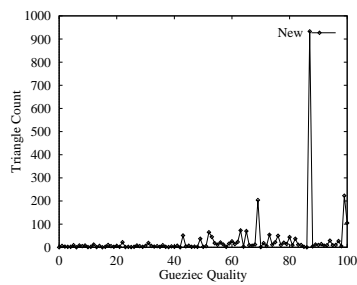
Figure 13: Histogram of Gueziec's quality measure for Buffalo terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new MF-GcTin algorithm in (c), and Scape in (d).



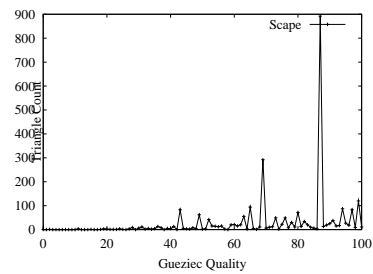
(a)



(b)



(c)



(d)

Figure 14: Histogram of Guezic's quality measure for Denver terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new MF-GcTin algorithm in (c) and Scape in (d).

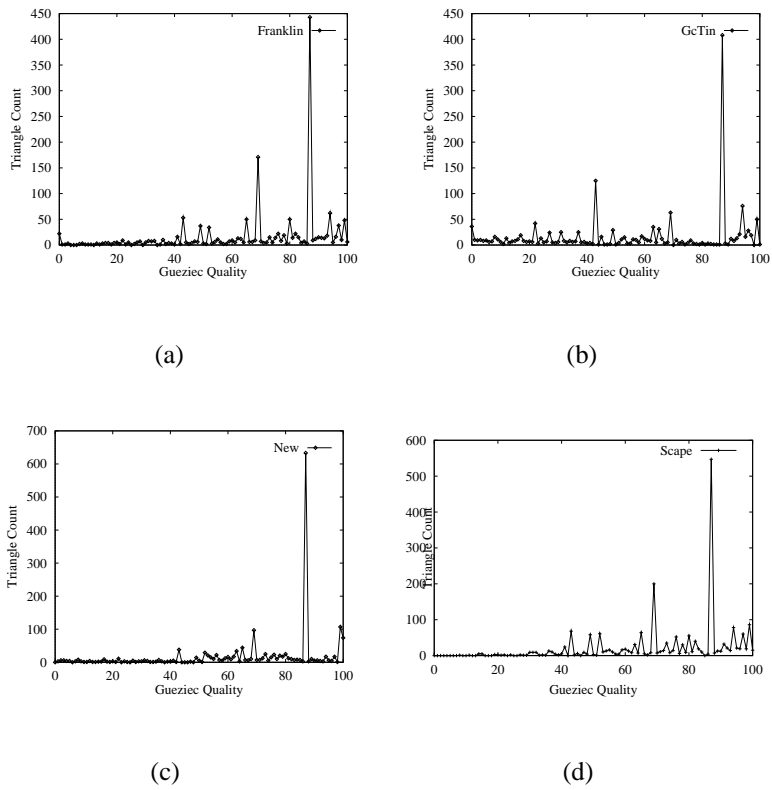
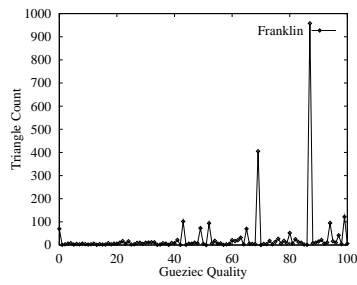
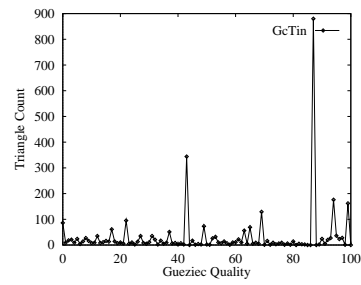


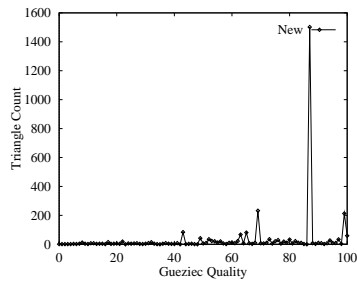
Figure 15: Histogram of Gueziec's quality measure for Eagle Pass terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new MF-GcTin algorithm in (c) and Scape in (d).



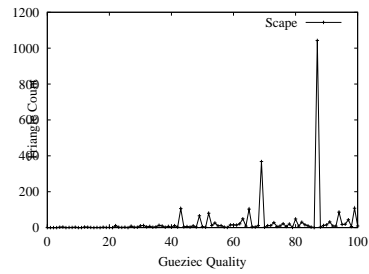
(a)



(b)

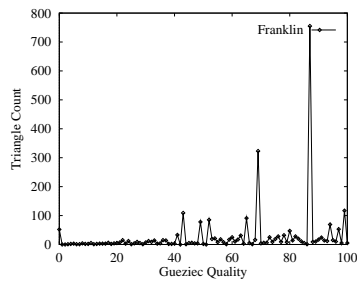


(c)

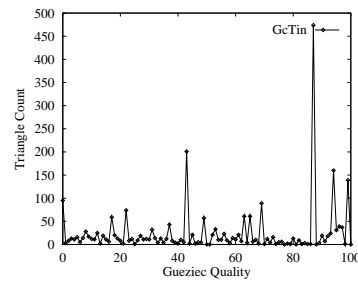


(d)

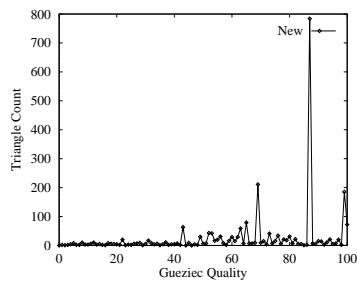
Figure 16: Histogram of Gueziec's quality measure for Grand Canyon terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new MF-GcTin algorithm in (c) and Scape in (d).



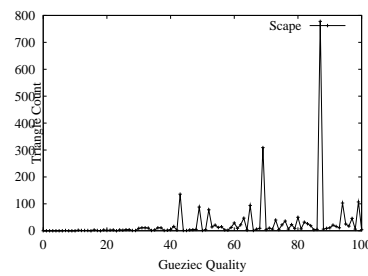
(a)



(b)

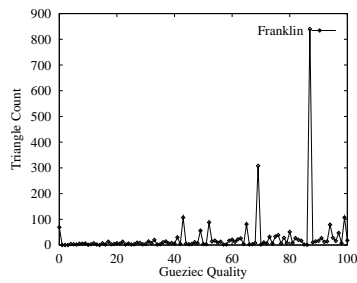


(c)

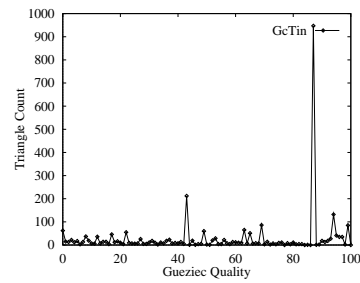


(d)

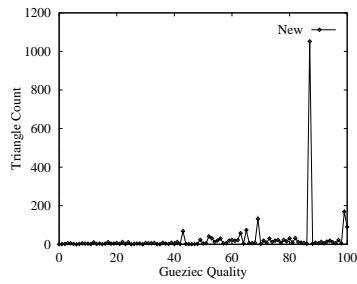
Figure 17: Histogram of Gueziec's quality measure for Moab terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new GcTin algorithm in (c) and Scape in (d).



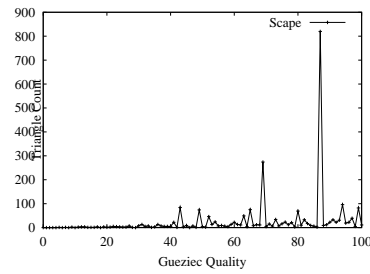
(a)



(b)



(c)



(d)

Figure 18: Histogram of Guezic's quality measure for Seattle terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new MF-GcTin algorithm in (c) and Scape in (d).