Greedy Cuts: An Advancing Front Terrain Triangulation Algorithm

Cláudio T. Silva*
IBM T. J. Watson Research Center

Joseph S. B. Mitchell†
State University of New York at Stony Brook

Abstract

We apply an advancing front technique to 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* algorithm uses a simple triangulation-growth procedure. We improve on our earlier advancing-front technique, which was not able to backtrack in its triangulation decisions, resulting in triangulations that may have low quality. The new algorithm we propose overcomes this shortcoming by maintaining 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 earlier advancing-front technique, while not significantly affecting memory usage.

`GcTin`, our terrain triangulation tool, is publicly available for research purposes. The original version of `GcTin` has been in use at several commercial and non-commercial sites since 1995. The new algorithms described here are integrated in the latest release and result in substantially improved triangulations.

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 [1]). 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 [6, 7, 8]. We refer the readers to the recent survey of Garland and Heckbert [4] or of van Kreveld [5] for a more 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 focussed on "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.

In a predecessor [9] to this paper, we developed an advancing-front algorithm and its implementation (`GcTin`). While the algorithm was shown to have favorable properties in terms of memory consumption and total number of triangles, it was much less effective at producing high *quality* triangulations from the point of view of angles that are close to zero or close to π.

The main contribution of this paper is a *hybrid* method 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, which do not permit backtracking of 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 maintain much higher quality triangulation, while preserving the low-memory feature of the advancing-front method.

*csilva@watson.ibm.com; Visual and Geometric Computing Group, P.O. Box 704, IBM T. J. Watson Research Center, Yorktown Heights, NY, 10598

†jsbm@ams.sunysb.edu; Department of Applied Mathematics and Statistics, State University of New York at Stony Brook, Stony Brook, NY 11794–3600

The input to our problem is a height field, $H(x,y)$, giving elevations at a regular array of points (x,y) within a rectangle R in the plane. We are also given a user-specified error tolerance, $\varepsilon > 0$. We desire to output a triangulated surface (TIN) that represents the terrain on R, such that the TIN has few triangles, the triangles are of good quality, and the TIN represents a surface that is ε-close to that represented by $H(x,y)$.

2 Review of Greedy Cuts

This section gives a high-level description of the Greedy Cuts method as originally proposed in [9].

The algorithm maintains a list of *untriangulated simple polygons*, P, which represent the portion of R over which no triangulated surface has yet been constructed. At each step, our goal is to select a "large" 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). 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 a good 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. Thus, at each step of the algorithm, a *bite* (triangle) T is taken out of the polygon P at the head of the list 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), T. There is no need to store in memory the list 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 counterclockwise order. Thus, only very small and simple data structures are required.

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 algorithm simply applies the above three operations, giving priority (in order) to ear cutting, greedy biting, and then edge splitting. A short description of the core of our (old) algorithm is outlined as follows:

```
GreedyCuts(Polygon *p)
{
  if(p == NULL)
    return;
  while(empty(p) == false) {
    if(EarCut(p) == false)
      if(GreedyCut(p) == false)
        if(SplitEdges(p) == false)
          NonFeasibleEarCut(p);
  }
  GreedyCuts(p->pNext);
}
```

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 that 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. 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 the mechanism the algorithm uses to adapt itself to lower sampling rates (larger triangles).

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 (see [9] for details), as it involves also "polygon splitting" when v lies close enough to another edge of 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, recursively, 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}).

Triangle Feasibility. We now define precisely what we mean by a triangle (in space) being "feasible" for input terrain H, with respect to a given ε. The input DEM H can be regarded as an instance of a TIN (a polyhedral surface, S), if we establish a set of edges that form a triangulation of the grid points for which H gives elevation values. Specifically, we fix a triangulation of the input data by considering 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 $(x, y, H(x, y) - \varepsilon)$ and $(x, y, H(x, y) + \varepsilon)$; i.e., T' fits the function at the relevant internal grid points. Note that any triangle T' that has a "skinny" or "small" enough projection, containing no grid points, is automatically weakly feasible. 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 surface obtained by shifting S downwards (resp., upwards) by ε. Strong feasibility implies weak feasibility, but the converse is not true in general. 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. Strong feasibility, however, requires that we also check the altitudes corresponding to those points that lie at the intersections of an edge of T with a grid edge.

Weaknesses. The main shortcoming of the original Greedy-Cuts triangulation algorithm of [9] is that it is not possible to backtrack at all during the generation of triangles: Each tri-
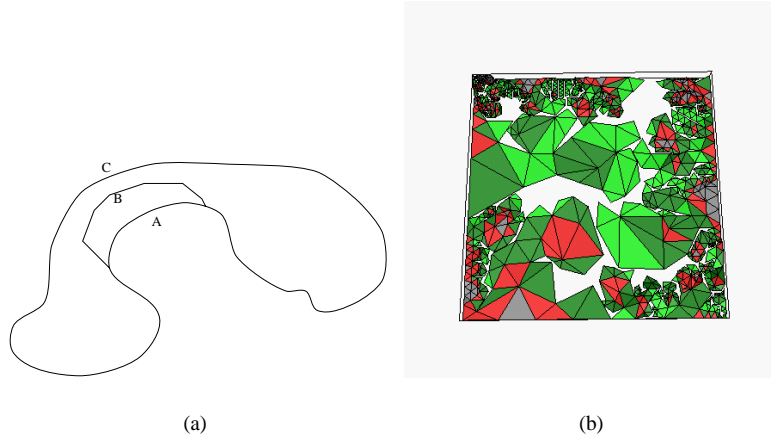
(a)   (b)

Figure 1: 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 neatly close 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.

angle 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 Fig. 1. 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 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.

3   The New Algorithm

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 trian-

gulated 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 [9], we now enforce a quality criterion based on Gueziec's notion of "compactness" [3]. Given a triangle with edge lengths l_0, l_1, l_2, the compactness measure g is given by

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

where A is the (positive) area of the triangle. Note that $0 \le g \le 1$, 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 Fig. 2a, we illustrate a standard ear cut, which generates triangle (p, q, r), while causing vertex q to be removed from the active list. Fig. 2b 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.

A high-level description of the ear cutting is:

```
int EarCut(Polygon *pg)
{
  workFlag = 0;
  curEdge = pg->firstEdge;
  do {
    p = curEdge;
    q = curEdge->nextPtr;
    r = curEdge->nextPtr->nextPtr;
    if(Diagonal(p, r)) {
```
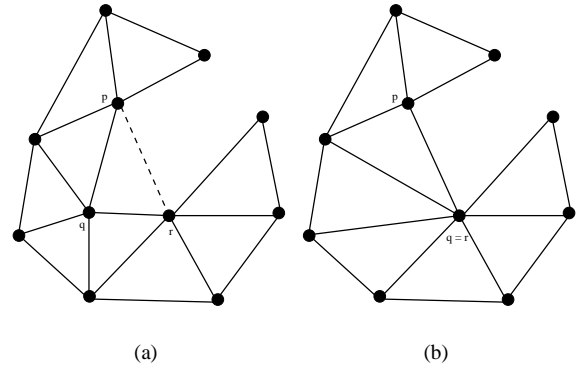

Figure 2: (a). A standard ear cut results in the addition of the edge (p, r) and the creation of the new triangle (p, q, r). (b). An edge collapse, of edge (q, r), results in the triangulation shown.

```
      if(CollapseEdge(p, q) == false)
        if(IsFeasibleTriangle(p, q, r)) {
          ExecuteCut(curEdge, edgeList, 1);
          workFlag = 1;
        }
    curEdge = curEdge->nextPtr;
  } while(curEdge != pg->firstEdge);
  return workFlag;
}
```

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 Fig. 3, 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.
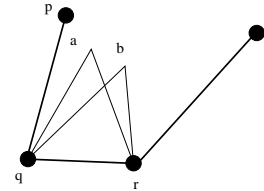

Figure 3: Angle considerations in selecting a new triangle. A quality measure is used in order to bias the algorithm in favor of nicely shaped triangles during triangulation.

Greedy Biting and Edge Splitting. As with ear cutting, our new greedy biting procedure has two 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
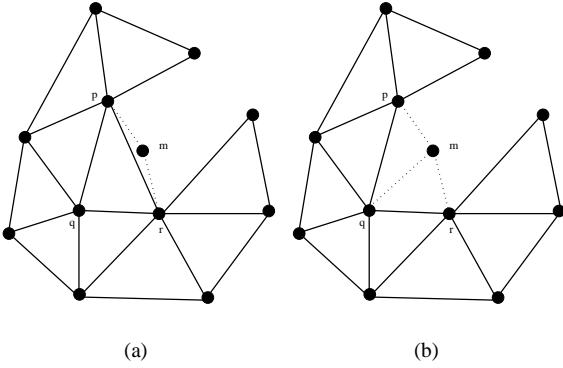
(a)   (b)

Figure 4: 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 there was not 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 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 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 Fig. 4.

(3) The polygon splitting that takes place in [9] when a bite results in a new vertex close to an existing (opposite) front edge is now replaced by an edge collapse operation.

A high-level description of the greedy biting operation is as follows:

```
int GreedyBite(Polygon *pg)
{
  workFlag = 0;
  curEdge = pg->firstEdge;
  do {
    successFlag = SplitAtEdge(curEdge, pg);
    if(successFlag) {
      workFlag = 1; return workFlag;
    }
    successFlag = BiteEdge(curEdge, pg);
    if(successFlag) {
      workFlag = 1;
    }
    curEdge = curEdge->nextPtr;
  } while(curEdge != pg->firstEdge);
  return workFlag;
}
```

Initial Boundary Smoothing. During the initialization phase of [9], we perform a curve fitting for the boundary of the terrain, finding a minimum-link approximation of the boundary,

subject to the error tolerance ε. 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, splitting 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. This procedure ensures a logarithmic scale on the size of the edges.

4   Experimental Results

We have implemented our new algorithm and compared it experimentally with the prior Greedy-Cuts algorithm [9] as well as an algorithm of Franklin [2]. We compared average error of the approximating surface, the complexity of the output (number of triangles), and the quality of the output triangles.

Franklin's Algorithm. Franklin's algorithm [2] is an incremental refinement method. 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 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 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 all three algorithms on seven real terrain datasets. We ran Franklin's algorithm, the original `GcTin` [9], and our new algorithm. 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 Fig. 5(a)–(d) for screen shots of partial triangulations of the Denver terrain during the running of the new algorithm.

In our previous work ([9]), we determined that `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 our new algorithm 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 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
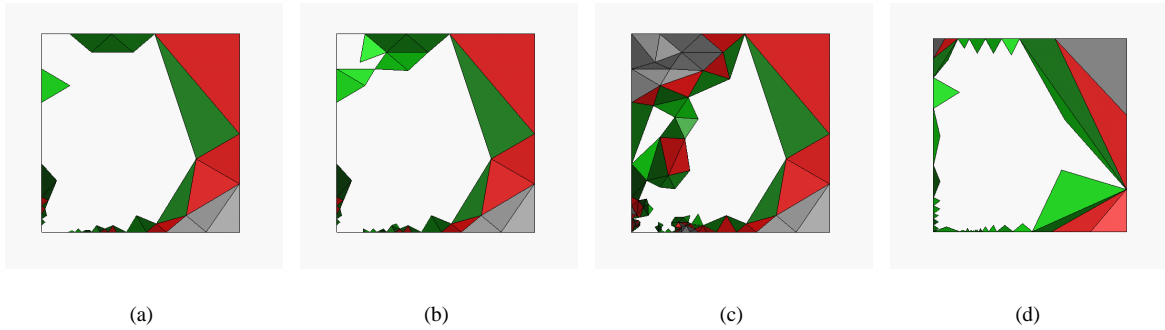
| (a) | (b) | (c) | (d) |

Figure 5: 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.

Terrain	ε	Franklin	GcTin	New
Buffalo	2.5	2044	2279	2610, 1758
Denver	2.5	2698	2849	2661, 1833
Eagle Pass	1.5	1559	1578	1653, 1377
Gr. Canyon	15	2804	3115	3050, 2050
Jackson	0.5	1430	1127	1430, 928
Moab	15	2572	2430	2358, 2204
Seattle	5	2703	2763	2476, 2107

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

both the number of triangles generated as well as their number, with respect to the original GcTin code. The memory cost is about the same, with a slight increase over GcTin, since we are storing a small number of (tentative) triangles. (In [9], we show that GcTin uses an order of magnitude less memory than Franklin's code.)

Triangle Quality Measures. Figs. 6–11 show histograms of Gueziec's quality measure for six (of the seven) terrains. It is clear that the new algorithm gives a substantially better distribution of triangle quality that either Franklin's algorithm or our earlier GcTin. In particular, we can see that the number of good triangles (the major peak) is always higher in the (c) column.

5 Final Remarks

Preliminary input from our current users have been very favorable on the released version of GcTin. We plan to release a new version of the software with the new 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.

As future work, we plan to extend our technique to allow real-time triangulation and to avoid actually storing the TINs for viewing purposes. Other interesting directions are the handling of gigabyte-size terrains, and more general data types.

The web site (http://cg.ams.sunysb.edu/˜csilva/gctin.tgz)

contains the current GcTin code. The updated version discussed in this paper will be available soon at the same url.

References

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

[2] W. 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/.

[3] A. Gueziec. Surface simplification with variable tolerance. In *Second Annual International Symposium on Medical Robotics and Computer Assisted Surgery*, pages 132–139, 1995.

[4] P. Heckbert and M. Garland. A Survey of Terrain Triangulation Algorithms. Technical report, Carnegie Mellon University, 1995.

[5] M. van Kreveld. Digital elevation models and TIN algorithms. In *Algorithmic Foundations of Geographic Information Systems*, LNCS volume 1340, pages 37–78, Springer-Verlag, 1997.

[6] 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.

[7] 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.

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

[9] C. Silva, J. Mitchell, and A. Kaufman. Automatic generation of triangular irregular networks using greedy cuts. *Proc. IEEE Visualization '95*, pages 201–208, 1995.
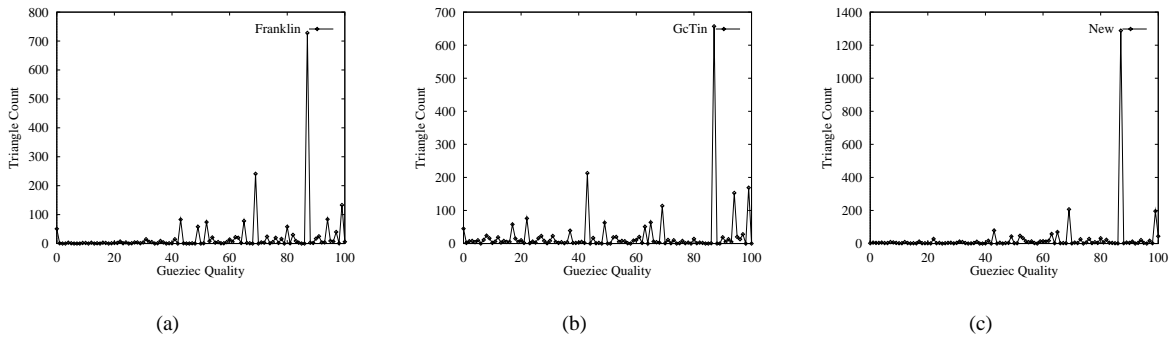
Figure 6: Histogram of Gueziec's quality measure for Buffalo terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new algorithm in (c).
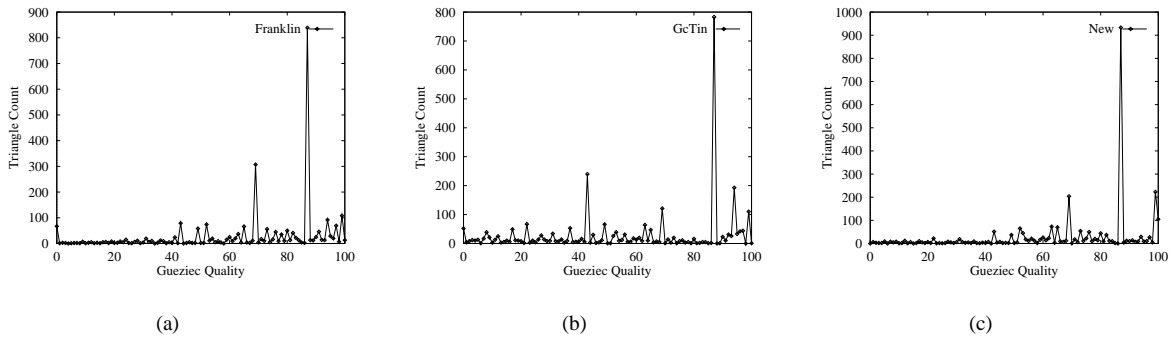

Figure 7: Histogram of Gueziec's quality measure for Denver terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new algorithm in (c).
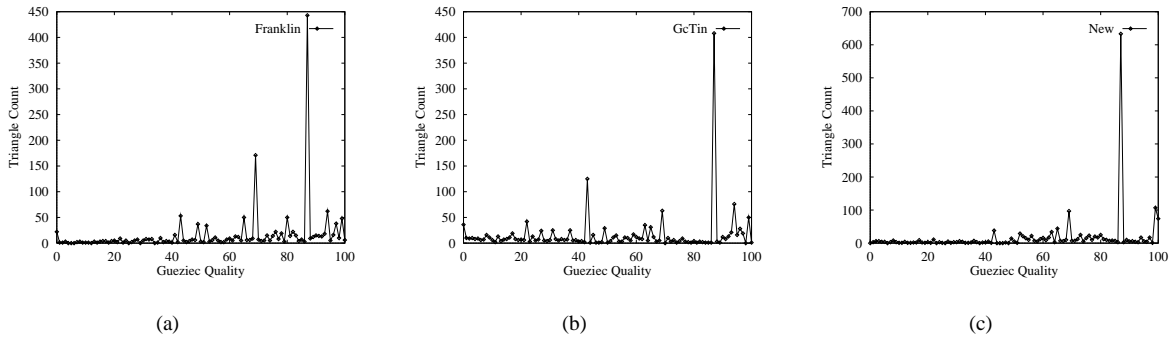

Figure 8: Histogram of Gueziec's quality measure for Eagle Pass terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new algorithm in (c).
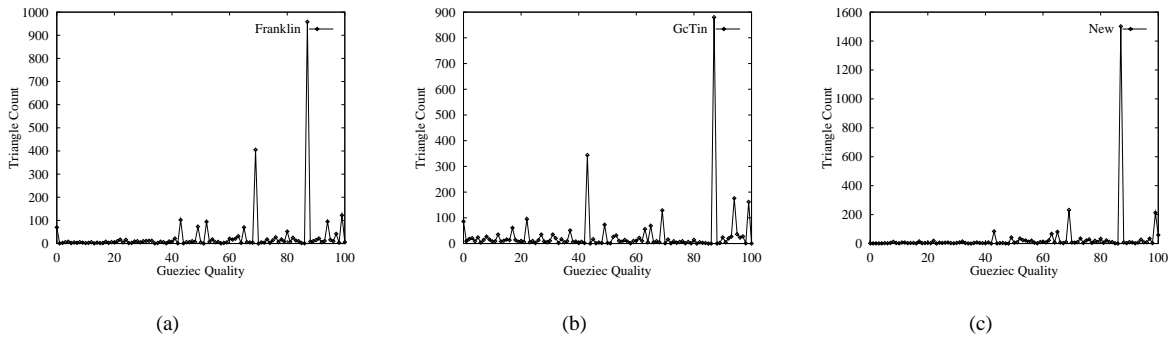
Figure 9: Histogram of Gueziec's quality measure for Grand Canyon terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new algorithm in (c).
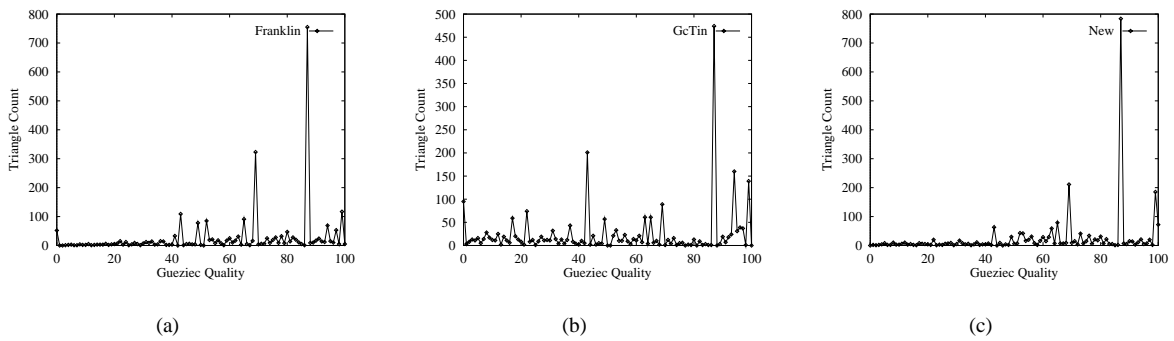

Figure 10: Histogram of Gueziec's quality measure for Moab terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new algorithm in (c).
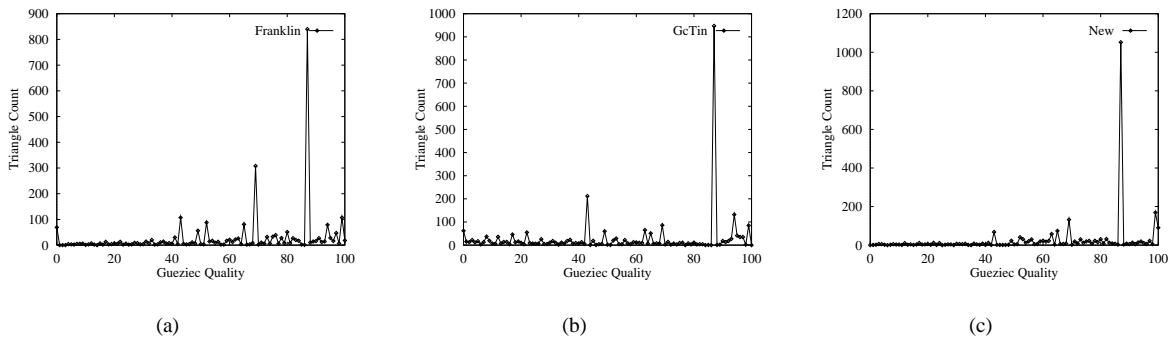

Figure 11: Histogram of Gueziec's quality measure for Seattle terrain. Franklin's algorithm is shown in (a). The original GcTin in (b), and our new algorithm in (c).