Fast Triangular Approximation of Terrains and Height Fields

Michael Garland and Paul S. Heckbert
Carnegie Mellon University*

May 2, 1997

Abstract

We present efficient algorithms for approximating a height field using a piecewise-linear triangulated surface. The algorithms attempt to minimize both the error and the number of triangles in the approximation. The methods we examine are variants of the *greedy insertion algorithm*. This method begins with a simple triangulation of the domain as an initial approximation. It then iteratively finds the input point with highest error in the current approximation and inserts it as a vertex in the triangulation. We describe optimized algorithms using both Delaunay and data-dependent triangulation criteria. The algorithms have typical costs of $O((m+n)\log m)$, where n is the number of points in the input height field and m is the number of vertices in the final approximation. We also present empirical comparisons of several variants of the algorithms on large digital elevation models. We have made a C++ implementation of our algorithms publicly available.

Keywords: surface simplification, surface approximation, Delaunay triangulation, data-dependent triangulation, triangulated irregular network, greedy insertion.

1 Introduction

A *height field* is a set of data values sampled at points in a planar domain. Terrain data, a common type of height field, is used in many applications, including flight simulators, ground vehicle simulators, video games, and in computer graphics for entertainment. Computer vision uses height fields to represent range data acquired by stereo and laser range finders. In all of these applications, an efficient data structure for representing and displaying the height field is desirable.

Our primary motivation is to render height field data rapidly and with high fidelity. Since almost all graphics hardware uses the polygon as the fundamental building block for object description, it seems natural to represent the terrain as a mesh of polygonal elements. The raw sample data can be trivially converted into polygons by placing edges between each pair of neighboring samples (see Figure 6). However, for terrains of any significant size, rendering this full model is prohibitively expensive. For example, the 2,000,000 triangles in a $1,000 \times 1,000$ grid take about seven seconds to render on current mid-range graphics workstations, which can display roughly 10,000 triangles in real time (every 30th of a second). Even as the fastest graphics workstations speed up in coming years, typical workstations and personal computers will remain far slower. More fundamentally, the detail of the full model is highly redundant when it is viewed from a distance, and its use in such cases is unnecessary and wasteful. Many terrains have large, nearly planar regions which are well approximated by large polygons. Ideally, we would like to render models of arbitrary height fields with just enough detail for visual accuracy. Additionally, in systems which are highly constrained, we would like to use a less detailed model in order to conserve memory, disk space, or network bandwidth.

To render a height field quickly, we can use multiresolution modeling, preprocessing it to construct approximations of the surface at various levels of detail [2, 15]. When rendering the height field, we can choose an approximation with an appropriate level of detail and use it in place of the original.

In some applications, such as flight simulators, the speed of simplification is unimportant, because database preparation is done off-line, once, while rendering of the simplified terrain is done thousands of times. However, in more general computer graphics and computer animation applications, the scene being simplified might be changing, so a slow simplification method might be useless. Finding a simplification algorithm that is fast is therefore quite important to us.

Our focus in this paper will be to generate simplified

*Computer Science Dept., Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh PA 15213-3891, USA. {garland,ph}@cs.cmu.edu http://www.cs.cmu.edu/~garland/scape

models of a height field from the original model. The simplified model should accurately approximate the original model, it should be as compact as possible, and the process of simplification should be as rapid as possible. We measure the compactness of a model in terms of the number of triangles (or, proportionally, the number of vertices), since that is the major factor influencing rendering speed, and fast rendering with accurate approximations is our main goal.

We do not attempt to survey other methods for surface simplification here; for a taxonomy and comparison of polygonal surface simplification methods, see our survey paper [16].

The remainder of this paper contains the following sections: We begin by stating the problem we are solving. Next we describe the greedy insertion algorithm and present three simple optimizations that speed it up dramatically. We explore the use of both Delaunay triangulation and data-dependent triangulation. The paper concludes with a discussion of empirical results, ideas for future work, and a summary.

2 Background

A height field is a function of two variables, $H(x, y)$, which represents a surface. The input to our system is a height field represented by a set of sample points in the plane and a set of data values associated with those points. For now, we will assume this to be a set of height samples, but later we will consider generalized values. We will assume that the set of sample points is arranged on a rectangular grid at integral coordinates, but the methods we will describe are easily generalized to scattered data points. This discrete representation can be transformed into a continuous surface by triangulating its set of points and defining a piecewise-linear function over this triangulation. An approximation of H can be constructed by triangulating a subset of the sample points of H such that the triangulation covers the entire domain of H. Such an approximation is often referred to as a *triangulated irregular network*, or TIN, in the cartography literature.

Our goal is to find an approximation $S(x, y)$ of $H(x, y)$ which approximates H as accurately as possible using as few points as possible, and to compute its triangulation as quickly as possible. We will let n be the number of input points in H and m be the number of vertices in the approximation S.

Delaunay and Data-Dependent Triangulation. In this paper, we explore the use of both Delaunay and data-dependent triangulations for constructing the approximate surface. *Delaunay triangulation* is a purely two-dimensional method; it uses only the xy projections of the input points. It finds a triangulation that maximizes the minimum angle of all triangles, among all triangulations of a given point set [12, 20]. This helps to minimize the occurrence of very thin *sliver* triangles. *Data-dependent triangulation*, in contrast, uses the heights of points in addition to their x and y coordinates [7, 26]. It can achieve lower error approximations than Delaunay triangulation, but it generates more slivers.

The incremental Delaunay triangulation algorithm starts with a simple, initial triangulation and inserts the points as vertices in the triangulation one-by-one, performing local topological updates after each insertion [12, 20]. The procedure to insert a single vertex is illustrated in Figure 1, and works as follows: To insert a vertex A, locate its containing triangle, or, if it lies on an edge, delete that edge and find its containing quadrilateral. Add "spoke" edges from A to the vertices of this containing polygon. All perimeter edges of the containing polygon are suspect and their validity must be checked. An edge is valid iff it passes the circle test: if A lies outside the circumcircle of the triangle that is on the opposite side of the edge from A. All invalid edges must be swapped with the other diagonal of the quadrilateral containing them, at which point the containing polygon acquires two new suspect edges. The process continues until no suspect edges remain. The resulting triangulation is Delaunay.

Refinement Methods. Refinement is one of the general approaches to surface simplification [16]. Refinement methods are multi-pass algorithms that begin with an initial approximation and iteratively add new vertices to the triangulation until some goal is achieved. This goal is typically stated in terms of a desired error threshold or a maximum number of vertices. In order to choose which points to add to the approximation, refinement methods rank the available input points using some *importance measure*.

In choosing an importance measure, we reject those that make use of implicit knowledge about the nature of terrains, such as the existence of ridge lines. We would like our algorithms to apply to general height fields, where assumptions that are valid for terrains might fail. Even if we were to constrain ourselves to terrains alone, we are not aware of conclusive evidence suggesting that high fidelity results (as measured by an objective L_2 or L_∞ metric[1]) require high level knowledge of terrains. After experiment-

[1] In this paper, we use the following error metrics: We define the L_2 error between two n-vectors $\mathbf{u}$ and $\mathbf{v}$ as $||\mathbf{u}-\mathbf{v}||_2 = \left[\sum_{i=1}^{n}(u_i - v_i)^2\right]^{1/2}$. The L_∞ error, also called the *maximum error*, is $||\mathbf{u}-\mathbf{v}||_\infty = \max_{i=1}^{n}|u_i - v_i|$. We define the *squared error* to be the square of the L_2 error, and the *root mean square* or RMS error to be the L_2 error divided by $\sqrt{n}$. Optimization with respect to the L_2 and L_∞ metrics are called *least squares* and *minimax* optimization, and we call such solutions L_2–*optimal* and L_∞–*optimal*, respectively.

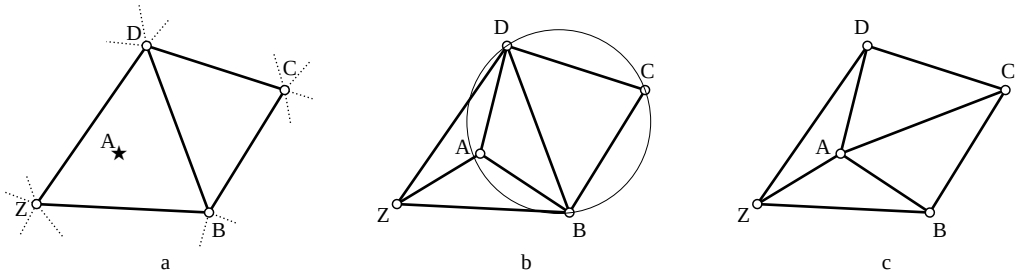

Figure 1: Incremental Delaunay triangulation: a) Point A is about to be inserted. Spoke edges from A to the containing polygon ZBD are added. b) The quadrilateral around suspect edge BD is checked using the circle test. The circumcircle of BCD contains A, so edge BD is invalid. c) After swapping edge BD for AC, edges BC and CD become suspect. The polygon ZBCD is the only area of the mesh that has changed.

ing with more complex alternatives [11], we chose one of the simplest importance measures, the local approximation error: $|H(x,y) - S(x,y)|$, which is simply the vertical distance at a point between the input data and the approximation.

3 Greedy Insertion

We call refinement algorithms that insert the point(s) of highest error on each pass *greedy insertion* algorithms, "greedy" because they make irrevocable decisions as they go, and "insertion" because on each pass they insert one or more vertices into the triangulation.

3.1 Previous Work

Many variations on the greedy insertion algorithm have been explored over the years; apparently the algorithm has been reinvented many times. However, in the previous work, analysis and testing on large problems were not always done.

In early work, Fowler and Little used a hybrid of feature methods and refinement methods to approximate a terrain [9]. A purer greedy insertion algorithm using Delaunay triangulation, with no optimizations described, was given by De Floriani *et al.* in 1983 [4, 5]. Others presented similar algorithms, demonstrating them on larger input data [24, 25]. In 1989 De Floriani described a more optimized version of the algorithm [3] with a typical cost, by our analysis, of $O(n \log m + m^2)$. Others independently developed essentially the same algorithm [10, 8]. Variations of this algorithm using least squares fitting, and using data-dependent triangulation instead of Delaunay triangulation, were given by Rippa [26]. Both variants typically improve the fit. The basic algorithm was optimized further by Heller, using a heap to reduce the typical cost, by our analysis, to $O((m+n)\log m)$ [17, p. 168].

Our contributions build on this work. We describe the greedy insertion approach in greater detail than it has been covered previously, optimize it, provide theoretical analysis of both worst case and typical complexity, and include extensive empirical testing on large terrain data sets. We have compared our greedy insertion method to several of the methods summarized above and found ours to be faster and/or more accurate.

3.2 Naive Algorithm

First, we consider simple and unoptimized greedy insertion [4, 5, 26, 24, 25]. We build an initial approximation of two triangles using the corner points of H. In repeated passes, we scan the points to find the one with the largest error and insert it into the current approximation using incremental Delaunay triangulation.

Cost Analysis of Naive Algorithm. Within each pass, we classify costs into three categories:

selection to pick the point of highest error,

insertion to insert a vertex into the mesh, and

recalculation to recalculate errors at grid points.

Recall that n is the number of points in the input grid and m is the number of points in the final approximation. Let L denote the time to locate one point in the Delaunay mesh and I be the time to insert a vertex into the mesh. Note that both I and L are mesh-dependent, and hence iteration-dependent.

For point location, we use the simple "walking method" described by Guibas-Stolfi [12, p. 121]. This algorithm starts on an edge of the mesh, and walks through the mesh toward the target point until it arrives at the target. Our variant of the algorithm, which generalizes it to a broader class of triangulations, is described in [11]. Logarithmic-time point location algorithms are known [13], of course, but it turns out that we will not need them.

In this simple implementation of greedy insertion, the costs per pass are: $O(n)$ to scan the points and select the

point of highest error, I for insertion, and $O(nL)$ to recalculate the errors at all of the n points. Recalculation, in this unoptimized algorithm, requires a point location of cost L to find the enclosing triangle, and an interpolation within the triangle of cost $O(1)$ for each mesh point.

In the worst case, a single location or insertion takes $O(i)$ time on pass i, and thus $L = I = O(i)$. This would yield an overall complexity of $\sum_{i=1}^{m} O(ni) = O(m^2 n)$ for this algorithm. Fortunately, the worst case behavior is very unlikely.

We will use the term "typical cost" to describe costs observed in practice (we do not use the term "expected cost", since we don't have a probabilistic model of the input). The typical cost of point location is only $L = O(1)$, since successive location sites are mostly in scanline order [12, 11]. Insertion requires point location, and our location method typically costs $O(\sqrt{i})$ in a triangulation with $O(i)$ vertices [12, 11]. Thus the typical insertion cost is $I = O(\sqrt{i})$. Therefore, the typical total cost is dominated by recalculation: $\sum_{i=1}^{m} O(n) = O(mn)$.

3.3 Optimized Algorithm

The algorithm above yields high quality results. However, even our typical time complexity estimate of $O(mn)$ is expensive, and the worst case complexity of $O(m^2 n)$ is exorbitant. Fortunately, we can improve upon our naive algorithm with three optimizations: (1) faster recalculation, (2) faster selection, and (3) the elimination of point location.

Faster Recalculation. After a Delaunay insertion, the point inserted will have edges emanating from it to the corners of a surrounding polygon [12]. This polygon defines the area in which the triangulation has been altered, and hence, it defines the area in which the approximation has changed (see Figure 1c). We call this polygon the *update region*. Such coherence permits our first significant optimization: we will cache the error values and only recompute errors within this update region. This can be done with polygon scan conversion (rasterization).

We can also use scan conversion to improve the efficiency of recalculation. With the naive algorithm, recalculation involved an interpolation within the enclosing triangle of each point. During scan conversion of a triangle, we can precompute a plane equation once and interpolate to that plane incrementally. This cuts the cost of recalculation by a significant constant factor.

Faster Selection. The next critical observation is that selection can be done more quickly with a heap or other fast priority queue. We define the *candidate point* of a triangle to be a grid point within the triangle that has the highest error in the current approximation. Each triangle can have at most one candidate point. Most triangles have one candidate, but if the maximum error inside the triangle is negligible, or there are no input points inside the triangle, then the triangle has none. We compute the candidate for each triangle as we scan convert it. These candidates are maintained in a heap keyed on their errors. During each pass, we simply extract the best candidate from the top of the heap.

Elimination of Point Location. In the naive algorithm, recalculation required a point location to find the enclosing triangle of each point. Insertion also required a point location. We can eliminate point location altogether by recording with each candidate a pointer to its containing triangle.

Data Structures. Our algorithm, listed below, has the following primary data structures: planes, height fields, triangulations, and heaps. A plane structure simply stores the coefficients a, b, and c for a plane equation $z = ax + by + c$. The height field consists of a rectangular array of points, each of which contains a height value $H(x, y)$, and a bit to record if the input point has been used by the triangulation. For the triangulation, we use a slight modification to Guibas and Stolfi's quad-edge data structure [12], which tracks triangles as well as 2-D points and directed edges. Triangles track their candidate's position (*candpos*), their location in the heap (*heapptr*), and their an error measurement (*err*). The heap contains triangles keyed on the error of their candidate point.

We state the conditions for termination abstractly as a function GOAL_MET(); they would typically be based on the number of points selected, the maximum error of the approximation, or the squared error of the approximation.

Delaunay Greedy Insertion:

```
HeapNode HEAP_CHANGE(HeapNode h, float key, Triangle T):
  % Set the key for heap node h to key,
  % set its triangle pointer to T, and adjust heap.
  % Return (possibly new) heap node.
  if h ≠ nil then
    if key > 0 then
      % update existing heap node
      HEAP_UPDATE(h, key)
      return h
    else
      % delete obsolete heap node
      HEAP_DELETE(h)
      return nil
  else
    if key > 0 then
      % insert new heap node
      return HEAP_INSERT(key, T)
    else
      return nil

MESH_INSERT(Point p, Triangle T):
  Insert a new vertex in triangle T
  Update the Delaunay mesh,
    calling HEAP_DELETE on candidates of deleted triangles
    and setting heapptr ← nil on new triangles

SCAN_TRIANGLE(Triangle T):
  plane ← FIND_TRIANGLE_PLANE(T)
  best ← nil
  maxerr ← 0
  forall points p inside triangle T do
    err ← |H(p) − INTERPOLATE_TO_PLANE(p, plane)|
    if err > maxerr then
      maxerr ← err
      best ← p
  T.heapptr ← HEAP_CHANGE(T.heapptr, maxerr, T)
  T.candpos ← best

INSERT(Point p, Triangle T):
  mark p as used
  MESH_INSERT(p, T)
  forall triangles U adjacent to p do
    SCAN_TRIANGLE(U)

GREEDY_INSERT():
  initialize mesh to two triangles formed by grid corners
  forall initial triangles T do
    SCAN_TRIANGLE(T)
  while not GOAL_MET() do
    T ← HEAP_DELETE_MAX()
    INSERT(T.candpos, T)
```

3.3.1 Cost Analysis of Optimized Algorithm

The three optimizations we have made speed up the algorithm significantly, both in theory and in practice.

In the optimized algorithm, time for selection is spent in three places: heap insertion, heap extraction, and heap updates. The growth of the heap per pass is bounded by the net growth in the number of triangles, which is 2. However, the heap does not always grow this fast. In particular, as triangles become so small or so well fit to the height field as to have no candidate points, they will be removed from the heap, and eventually the heap will shrink. Typically, the approximations that we wish to produce are much smaller than the original height fields. Consequently, the algorithm will rarely realize any significant benefit from shrinking heap sizes. To be conservative, we will assume that the size of the heap on pass i is $O(i)$, and thus, that individual heap operations require $O(\log i)$ time.

The number of changes made to the heap per pass is 3 plus the number of edge flips performed during mesh insertion. We assume that this is a small constant number. This is empirically confirmed on our sample data; in practice, the number of calls to HEAP_CHANGE per pass is roughly 3–5. Given this assumption, the total heap cost, and hence selection cost, is $O(\log i)$ per pass.

The other two tasks, insertion and recalculation, are also cheaper now, since neither performs locations, and recalculation also exploits locality and uses cached plane equations. The cost of recalculation has dropped from $O(nL)$ to $O(A)$, where A is the area of the update region.

Worst Case Time Cost. In the worst case, the insertion time is $I = O(i)$ and the update area is $A = O(n)$, so the costs per pass are: $O(\log i)$ for selection, $O(i)$ for insertion, and $O(n)$ for recalculation. The asymptotically dominant term is recalculation, as before, but it is much smaller now; the total worst case cost is only $\sum_{i=1}^{m} O(n) = O(mn)$.

Typical Case Time Cost. The typical number of edge flips is constant, so the cost of insertion is $I = O(1)$ and the size of the update region is $A = O(n/i)$. The costs per pass are thus: $O(\log i)$ for selection, $O(1)$ for insertion, and $O(n/i)$ for recalculation. The selection cost grows as the passes progress, while the recalculation cost shrinks. These two are the dominant terms. The total cost in the typical case is thus: $\sum_{i=1}^{m} O(\log i + n/i) = O((m+n)\log m)$.

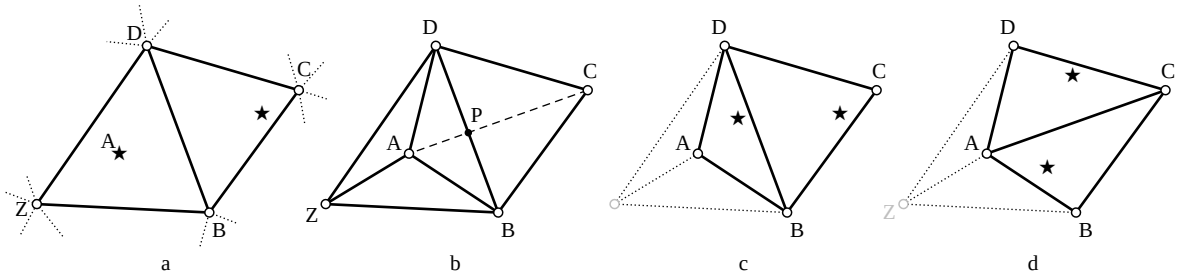

Figure 2: Data-dependent triangulation. a) Point A, the candidate of triangle ZBD, is about to be inserted. Stars denote candidate points. Spoke edges from A to the containing polygon ZBD are added. b) The quadrilateral ABCD around suspect edge BD is checked. ABCD can be triangulated in two ways, using diagonals BD or AC, which intersect at point P. c) If BD yields the lowest error, then we have the new triangle ABD and the old triangle CDB. d) If AC has lowest error, then we have the new triangles DAC and BCA, the containing polygon expands to ZBCD, and edges BC and CD become suspect.

3.4 Data-Dependent Triangulation

So far we have used Delaunay triangulation, which employs only two-dimensional (xy) information, and strives to create well-shaped triangles in 2-D. More accurate approximation is possible using data-dependent triangulation, where the topology of the triangulation is chosen based on the three-dimensional fit of the approximating surface to the input points.

Data-dependent variants of the greedy insertion algorithms described can be created by replacing Delaunay triangulation with data-dependent triangulation, as discussed by Rippa and Hamann-Chen [26, 14]. The vertex to insert in the triangulation on each pass is chosen as before, but the triangulation is done differently.

The incremental Delaunay triangulation algorithm used above tested suspect edges with a purely two-dimensional geometric test involving circumcircles. A generalization of this approach, Lawson's local optimization procedure [20], uses other tests. For data-dependent triangulation, instead of checking the validity of an edge with the circle test, the rule we adopt is that an edge is swapped if the change decreases the error of the approximation. We defer the definition of "error" until later. When used with the circle test, the local optimization procedure finds a global optimum, the Delaunay triangulation, but when used with more general tests, it is only guaranteed to find a local optimum.

Figure 2 illustrates our local optimization procedure for the data-dependent triangulation algorithm. Figure 2a: suppose that point A has the highest error of all candidates. It will be the next vertex inserted in the triangulation. Figure 2b: spokes are added connecting it to the containing polygon (a triangle, if A falls inside a triangle; a quadrilateral, if A falls on an edge). Each edge of the containing polygon is suspect, and must be tested. In some cases, the quadrilateral containing the edge will be concave, and can only be triangulated one way, but in most cases, the quadrilateral will be convex, and the other diagonal must be tested to see if it yields lower error.

The most straightforward way to test validity of an edge BD would be the following recursive procedure: Test both ways of triangulating the quadrilateral ABCD containing the edge. If edge BD yields lower global error (Figure 2c), then no new suspect edges are added, and we stop recursing. If swapping edge BD for edge AC would reduce the global error of the approximation (Figure 2d), then swap the edge to AC, and recurse on the four new suspect edges, BC, CD, AB, and AD.

Edges AB and AD are not suspect with the Delaunay criterion, but they are suspect when using a data-dependent criterion in Lawson's local optimization procedure, since swapping them might decrease the error. In our implementation, we do not test spokes such as AB and AD for swapping. This rule guarantees that the update region is exactly the containing polygon. This simplifies the implementation, but may sacrifice some quality.

When all suspect edges have been tested, it is then necessary to update the candidates for all the triangles in the containing polygon. This straightforward approach requires scan converting most of the triangles in the local neighborhood twice: once to test for swapping and once to find candidates.

A faster alternative is to scan convert once, computing both the global error and the candidate in one pass. This is about twice as fast. To do this, we split the quadrilateral ABCD with its two diagonals into four subtriangles: PDA, PAB, PBC, and PCD, where P is the intersection point of the two diagonals (Figure 2b). This splitting is conceptual; it is not a change to the data structures. As each of the four

subtriangles is scan converted, two piecewise-planar approximations are tested. For subtriangle ABP, for example, the planes defined by ABD and BCA are both considered. The other subtriangles have different plane pairs. During scan conversion of each subtriangle, for each of its two planes, the contribution to the triangulation's total error is calculated, and the best candidate point and its error is calculated. After scan conversion, the subtriangles' errors and candidates are combined pairwise to determine the error and candidates for each of the two pairs of triangles: ABD and CDB, versus BCA and DAC. Note that, with one exception, the triangle CDB is an old triangle, so its error and candidate have previously been computed, and need not be recomputed. The sole exception to this rule is during initialization when the first two triangles are being created.

Data Structures. The algorithm uses all of the previous data structures plus a new one, the *FitPlane*. A *FitPlane* is a temporary data structure that stores an approximation plane and other information. During scan conversion of the four subtriangles, it accumulates information about the error and candidate for a triangle approximated by a plane. Specifically, it contains the coefficients for the planar approximation function *plane*, the candidate's position *candpos* and error *canderr*, the error over the triangle *err*, and a *done* bit recording whether the triangle was previously scanned.

The *Triangle* and *Heap* data structures cache information about candidates and errors that is re-used during data-dependent insertion. A *FitPlane* can be initialized from this information with the subroutine FITPLANE_EXTRACT(*Triangle T*), which also marks the *FitPlane* as *done*. When a new triangle is being tested, the call FITPLANE_INIT(*a, b, c*) will initialize a *FitPlane* to a plane through the three points *a*, *b*, and *c*, with errors set to 0, and *done* = nil. One or more subsequent calls to SCAN_TRIANGLE_DATADEP are made to accumulate error and candidate information in the *FitPlane*. If this approximation plane turns out to be the best one, the heap is updated and the error and candidate information is saved for later use with a call to SET_CANDIDATE (listed below).

The routines LEFT_TRIANGLE and RIGHT_TRIANGLE return the triangles to the left and the right of a directed edge, respectively. The keyword "var" marks call-by-reference parameters.

Data-Dependent Greedy Insertion:

```
SET_CANDIDATE(var Triangle T, FitPlane fit):
  T.heapptr ← HEAP_CHANGE(T.heapptr, fit.canderr, T)
  T.candpos ← fit.candpos
  T.err ← fit.err
```

```
SCAN_POINT(Point x, var FitPlane fit):
  err ← |H(x) − INTERPOLATE_TO_PLANE(x, fit.plane)|
  fit.err ← ERROR_ACCUM(fit.err, err)
  if err > fit.err then
    fit.canderr ← err
    fit.candpos ← x
```

```
SCAN_TRIANGLE_DATADEP(Point p, Point q, Point r,
    var FitPlane u, var FitPlane v):
  % Scan convert triangle pqr,
  % updating error and candidate for planes u and v.
  % Plane u might be nonexistent or already done.
  forall points x inside triangle pqr do
    if u ≠ nil and not u.done then
      SCAN_POINT(x, u)
    SCAN_POINT(x, v)
```

```
FIRST_BETTER(float q1, float q2, float e1, float e2):
  % Return true iff edge 1 yields better triangulation of a
  % quadrilateral than edge 2, according to shape and fit.
  % q1 and q2 are "shape quality", and e1 and e2 are
  % fit error of the corresponding triangulations.
  qratio ← MIN(q1, q2) / MAX(q1, q2)
  % Use shape criterion if shape of one triangulation
  % is much better than the other, otherwise use fit.
  if qratio ≤ qthresh then
    return (q1 ≥ q2)          % shape criterion
  else
    return (e1 ≤ e2)          % fit error criterion
```

```
CHECK_SWAP(DirectedEdge e, FitPlane abd):
  % Checks edge e, swapping it if that reduces error,
  % updating triangulation and heap.
  % Error and candidate for the triangle to the left of e
  % is passed in in abd, if available.
  % Points a, b, c, d, and p are as shown in figure 2b,
  % and e is edge from b to d.
  if abd = nil then
    abd ← FITPLANE_INIT(a, b, d)
  if edge e is on boundary of input grid
      or quadrilateral abcd is concave then
    % Edge bd is good and edge ac is bad.
    if not abd.done then
      SCAN_TRIANGLE_DATADEP(a, b, d, nil, abd)
    SET_CANDIDATE(LEFT_TRIANGLE(e), abd)
  else
    % Check whether diagonal bd or ac has lower error.
    FitPlane cdb ← FITPLANE_EXTRACT(RIGHT_TRIANGLE(e))
    FitPlane dac ← FITPLANE_INIT(d, a, c)
    FitPlane bca ← FITPLANE_INIT(b, c, a)
    % scan convert the four subtriangles
    SCAN_TRIANGLE_DATADEP(p, d, a, abd, dac)
    SCAN_TRIANGLE_DATADEP(p, a, b, abd, bca)
    SCAN_TRIANGLE_DATADEP(p, b, c, cdb, bca)
    SCAN_TRIANGLE_DATADEP(p, c, d, cdb, dac)
    if FIRST_BETTER(SHAPE_QUALITY(a, b, c, d),
        SHAPE_QUALITY(b, c, d, a),
        ERROR_COMBINE(abd.err, cdb.err),
        ERROR_COMBINE(dac.err, bca.err)) then
      % keep edge bd
      SET_CANDIDATE(LEFT_TRIANGLE(e), abd)
      if not cdb.done then
        SET_CANDIDATE(RIGHT_TRIANGLE(e), cdb)
    else
      Swap edge e from bd to ac.
      dac.done ← bca.done ← true
      CHECK_SWAP(DirectedEdge cd, dac)  % recurse
      CHECK_SWAP(DirectedEdge bc, bca)

INSERT_DATADEP(Point a, Triangle T):
  Mark input point at a as used.
  In triangulation, add spoke edges connecting a to vertices
    of containing polygon (T and possibly a neighbor of T).
  forall counterclockwise perimeter edges e
      of containing polygon do
    CHECK_SWAP(e, nil)

GREEDY_INSERT_DATADEP():
  Initialize mesh to two triangles spanning height field.
  e ← (either directed edge along diagonal)
  CHECK_SWAP(e, nil)
  while not GOAL_MET() do
    T ← HEAP_DELETE_MAX()
    INSERT_DATADEP(T.candpos, T)
```

3.4.1 Data-Dependent Criterion

The routines ERROR_ACCUM and ERROR_COMBINE are used to accumulate the error over a subtriangle, and to total the error of a pair of triangles, respectively. These can be defined in various ways. For an L_2 error measure, they should be defined:

```
float ERROR_ACCUM(float accum, float x):
  return accum+x*x
float ERROR_COMBINE(float err1, float err2):
  return err1+err2
```

and for an L_∞ error measure, they should be defined:

```
float ERROR_ACCUM(float accum, float x):
  return MAX(accum, x)
float ERROR_COMBINE(float err1, float err2):
  return MAX(err1, err2)
```

The data-dependent-based method described above is slower than the Delaunay-based algorithm because it requires about twice as many error recalculations during scan conversion. However, the asymptotic complexities are identical to the Delaunay-based algorithm. Thus, the worst case cost is $O(mn)$ and its typical cost is $O((m+n)\log m)$.

3.4.2 Combating Slivers

Pure data-dependent triangulation, which makes swapping decisions based exclusively on fit error, will sometimes generate very thin sliver triangles. If the triangles fit the data well, and the surface is being displayed in shaded (not vector) form, then slivers by themselves are not a problem. But sometimes these slivers do not fit the data well, and lead to globally inaccurate approximations. After experiments with several sliver-avoidance schemes (see [11, 26]), we adopted a hybrid of data-dependent and Delaunay triangulation.

The pseudocode above implements this hybrid. The procedure SHAPE_QUALITY(a, b, c, d) returns a numerical rating of the shape of the triangles when quadrilateral *abcd* is split by edge *bd*. This rating should be constructed so that higher values indicate "better" shape. The parameter *qthresh* used in FIRST_BETTER is a quality threshold. When set to 0, pure data-dependent triangulation results, when set to 1, pure shape-dependent triangulation results, and when set in between, a hybrid results. If SHAPE_QUALITY returns the minimum angle of the triangles *abd* and *cdb*, then this shape-dependent triangulation will in fact be Delaunay triangulation. The hybrid method typically yielded lower error approximations than pure shape- or pure data-dependent triangulation.

Name	Dimensions	Location
West US	1024×1024	Idaho/Wyoming border
NTC	1024×1024	Tiefort Mtns., California
Ozark	369×462	Ozark, Missouri
Crater	336×459	Crater Lake, Oregon
Ashby	346×452	Ashby Gap, Virginia

Table 1: DEM datasets used for testing the simplification algorithms.

3.5 Extended Height Fields

So far, we have developed algorithms for simplifying basic height fields, and we have described techniques for making them faster without sacrificing quality. The greedy insertion algorithm can also be used to simplify data other than scalar height fields.

Consider the case in which our data specifies more than just height. For instance, the grid might contain measurements for some material property of the surface such as color, expressed as an RGB triple. Our algorithm can be easily adapted to support such extended height fields. Up to now, we have considered the simple case of surfaces of the form $H(x, y) = z$. An *extended height field* is one where the data values are tuples rather than single numbers. For example, we might model a color-textured terrain as a surface $H(x, y) = (z, r, g, b)$. We can think of this as sampling a set of distinct surfaces, one in *xyz*-space, one in *xyr*-space, and so on. We see here another reason to reject triangulation schemes that attempt to fit specific surface characteristics; we now have 4 distinct surfaces which need have no features in common. Given data for a generic set of surfaces, we can apply the importance measure to each surface separately and then compute some kind of average of these values. But when we know the precise interpretation of the data (i.e. the values represent height and color), we can construct a more informed measure. Our old measure was simply $|\Delta z|$; a reasonable extension to deal with color is $|\Delta z| + \frac{M}{3}(|\Delta r| + |\Delta g| + |\Delta b|)$. Here, M is the z-range of H; the $\frac{M}{3}$ term scales the total color difference to fit the range of the total height difference (here we assume that color values are between 0 and 1). In order to achieve greater flexibility, we can also add a color emphasis parameter, w, controlling the relative importance of height difference and color difference. The final error formula would be: $(1-w)|\Delta z| + w\frac{M}{3}(|\Delta r| + |\Delta g| + |\Delta b|)$.

To implement these changes, we simply added fields to the height field to record r, g, and b, modified the *FitPlane* to retain planar approximations to these three additional surfaces, and changed the error procedure to use the extended formula above.

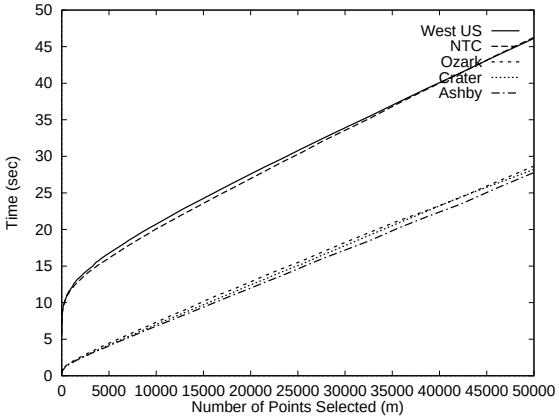

Figure 3: Running time of Delaunay greedy insertion on several DEM datasets.

These extensions allow our algorithm to be used to simplify terrains with color texture or planar color images [27]. The output of such a simplification is a triangulated surface with linear interpolation of color across each triangle. Such models are ideally suited for hardware-assisted Gouraud shading on most graphics workstations, and are a possible substitute for texture mapping when that is not supported by hardware.

4 Results

Our combined implementation of the Delaunay and data-dependent algorithms consists of about 5,200 lines of C++. The incremental Delaunay triangulation module is adapted from Lischinski's code [22].

Figures 4–11 are a demonstration of greedy insertion based approximation of a digital elevation model (DEM) for the western half of Crater Lake. Figure 4 shows the full DEM dataset (a rectangular grid with each quadrilateral split into two triangles). Our first approximation, shown in Figure 5, shows an approximation using 1% of the total points. This approximation has captured the major features of the terrain. However, it is still clearly different from the original. Figure 8 is an approximation using 5% of the original points. This model contains most of the features of the original. Thus, using only a fraction of the original data points, we can build high fidelity approximations. In a multiresolution database, we would produce a series of approximations, for use at varying distances.

Color Figures 16–18 illustrate the application of height field simplification methods to the approximation of planar color images by Gouraud shaded triangles. Figure 17 shows approximation by uniform subsampling, and Figure 18 shows approximation by data-dependent greedy in-

sertion, both using the same number of vertices. In both cases, the best results (shown) were achieved by low pass filtering the input before approximating. Clearly, greedy insertion yields a much better approximation.

4.1 Speed of the Algorithms

We have tested the performance of our simplification algorithms on a Silicon Graphics Indigo2 with a 150 MHz MIPS R4400 processor and 64 megabytes of main memory. For our timing tests we have used several digital elevation models. They are summarized in Table 1.

Figure 3 shows the running time of the Delaunay-based algorithm on the various DEM datasets as a function of points selected. In all cases, it was able to select 50,000 points in under one minute. In Figures 3, 12, 13, and 15, n is fixed (although it varies between datasets of different sizes) and the horizontal axis is m. All of the data points in Figure 3 are fit by the function

$$\text{time}(m,n) = .000001303\, n \log m - .0000259\, m \log m + .00000326\, n + .000845\, m - 0.178 \log m + .1 \text{ sec.}$$

with a maximum error of 1.7 seconds, supporting our $O((m+n)\log m)$ typical cost formula.

To quantify the improvement in efficiency due to our optimizations, we also implemented a naive greedy insertion algorithm. The optimizations proved to be very significant. In the time it takes our optimized algorithm to select 50,000 points from a $1{,}024 \times 1{,}024$ terrain (46 seconds), the naive algorithm can only manage to select a few hundred points from a 65×65 terrain [11]. These speedups were achieved without sacrificing quality; our optimizations increase processing speed with no meaningful change in the points selected or the approximation[2].

4.2 Memory Use

The optimized algorithm uses memory for three main purposes: the height field, the mesh, and the heap. The height field uses space proportional to the number of grid points n, and the mesh and heap use space proportional to the number of vertices in the mesh, m. Asymptotically, the memory cost is thus $O(m+n)$.

We now detail the memory requirements of our current implementation. For every point in the height field, we store one 2-byte integer for the z value, and a 1-byte Boolean indicating whether this point has been used. Thus, these arrays consume $3n$ bytes. In the mesh, 16 bytes are used to store each vertex's position, 68 bytes are used per edge, and 24 bytes per triangle, so assuming there are about $3m$ edges and $2m$ triangles, the memory required for a mesh with m vertices is $268m$ bytes. The heap uses 12 bytes per node, so heap memory requirements are about $12 \cdot 2m = 24m$ bytes.

Total memory requirements of the data structures in our implementation are therefore $3n + 292m$ bytes. Thus, for example, we estimate that selecting $m = 10,000$ (1% of total) points from an $n = 1,024^2$ DEM would require about 6 megabytes of memory.

Our current implementation stores floating point numbers using double precision (8 bytes), and uses the quad-edge data structure to store the mesh. The quad-edge structure is less compact than some triangulation data structures, and double precision floating point may not always be necessary. So the program's memory requirements could probably be cut significantly.

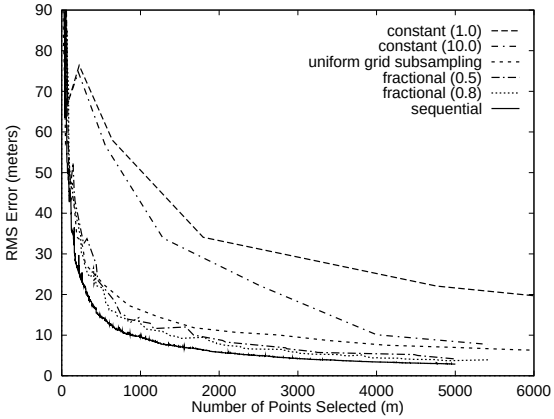

Figure 12: RMS error of approximation as vertices are added to the mesh, for Crater Lake DEM, for uniform grid subsampling and several variants of Delaunay greedy insertion: sequential insertion, fractional threshold parallel insertion with two different fractions α, and constant threshold parallel insertion with two different thresholds ϵ.

4.3 Quality of the Approximations

We believe that the greedy insertion algorithm yields good results on most reasonably smooth height fields. This can be verified both visually and with objective error metrics. Figure 12 shows the RMS error as an approximation for the Crater Lake DEM is built one point at a time. This figure also shows the error behavior for some variant insertion policies, but we will ignore all but the "sequential" curve for the moment.

At a coarse level, the RMS error initially decreases quite rapidly and then slowly approaches 0. In the early phases

[2] In the case of ties between candidates of equal importance, implementation details might cause a difference in selection order.

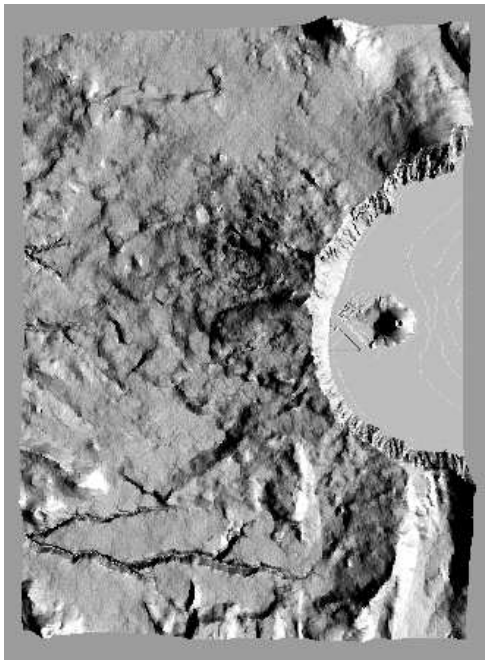

Figure 4: Original DEM data for west end of Crater Lake (154,224 vertices). Note the island in the lake.

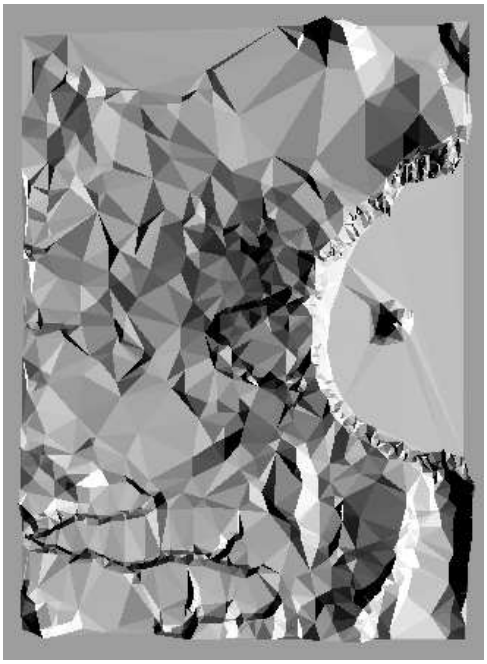

Figure 5: Delaunay approximation using 1% of the input points (1,542 vertices).

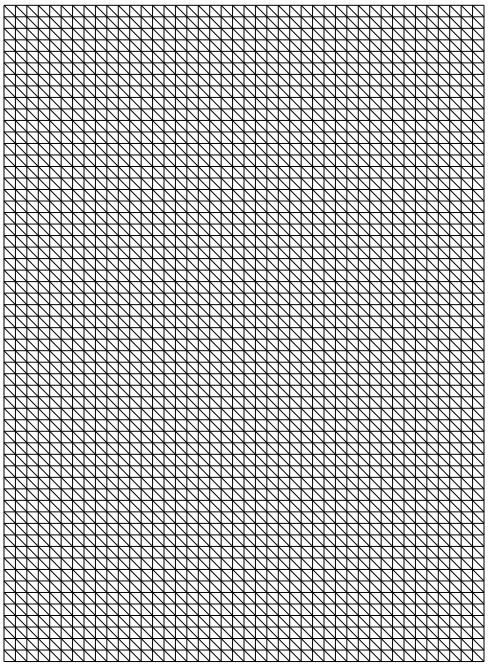

Figure 6: Mesh identical to the one used for the DEM picture above, except this mesh uses only every eighth point in x and y, for clarity.

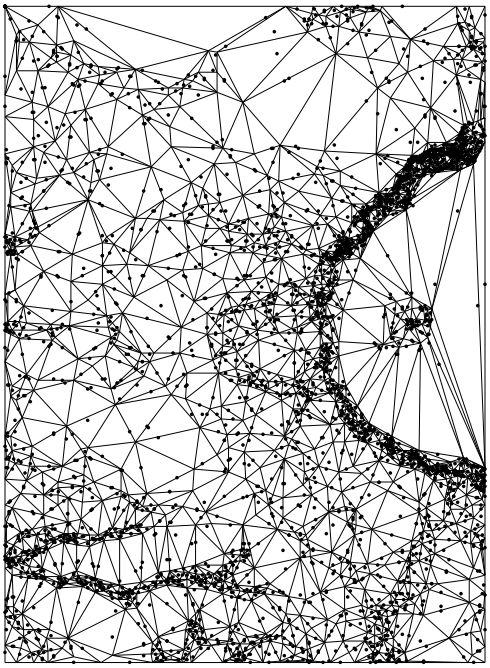

Figure 7: Delaunay mesh for the approximation above. Candidates are shown with dots. It is interesting to note that most candidates fall near edges.

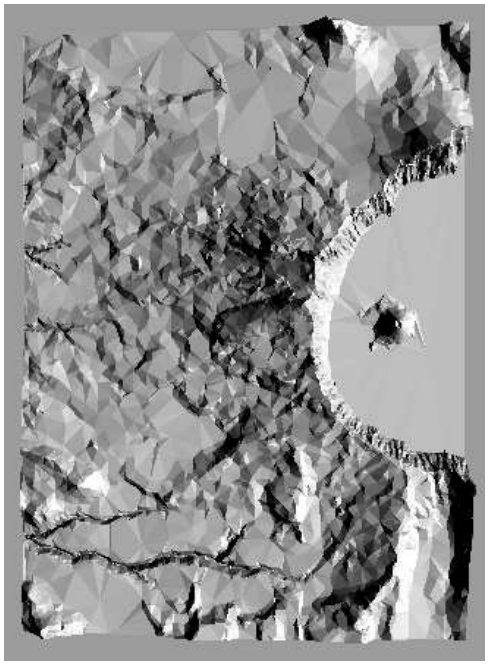

Figure 8: Delaunay approximation using 5% of the input points (7,711 vertices).

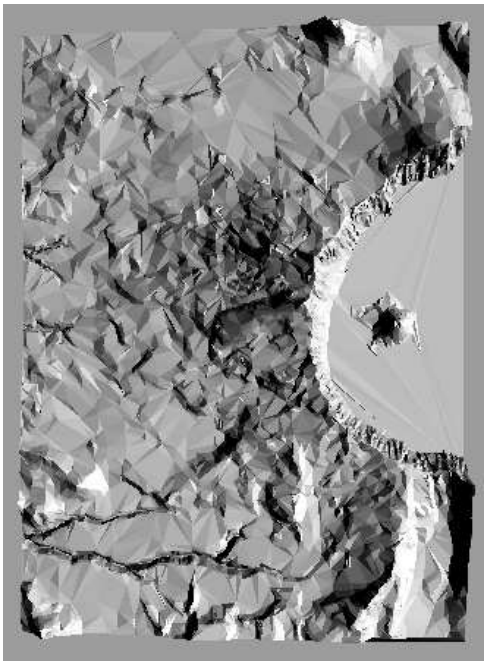

Figure 9: Data-dependent approximation using 5% of the input points (7,711 vertices).

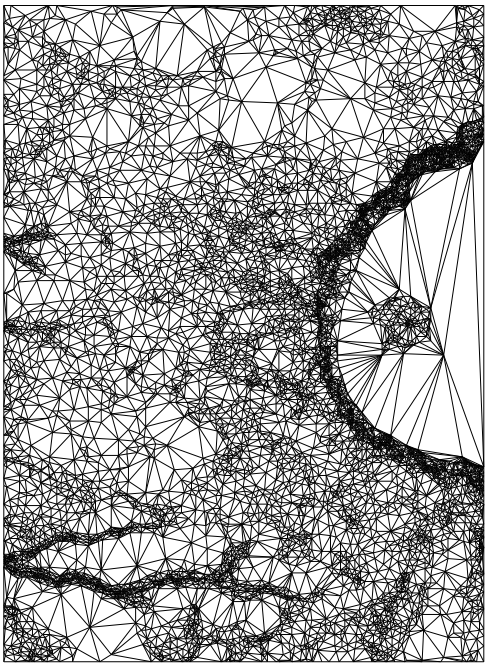

Figure 10: Delaunay mesh for the approximation above.

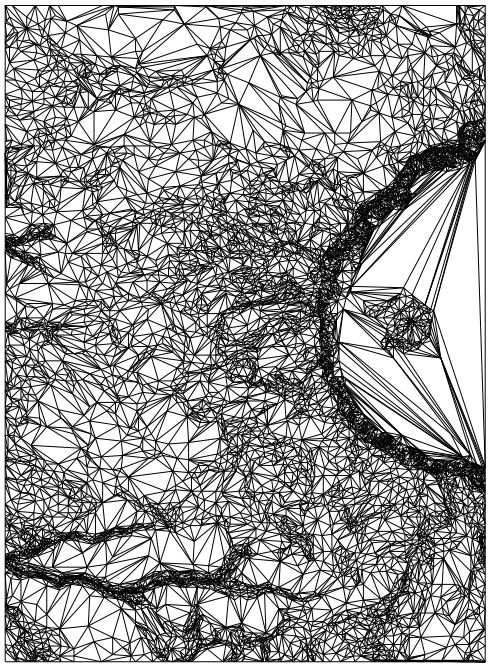

Figure 11: Data-dependent mesh for the approximation above.

of the algorithm, the error fluctuates rather chaotically, but it settles into a more stable decline. Theoretically, in the limit as $m \to \infty$, the error of the L_2-optimal triangulation converges as m^{-1} [23], but this empirical data is better fit by the function $m^{-.7}$. While we only show the error curve for a single terrain, we have tested the error behavior on several terrains, and the curves all share the same basic characteristics.

Data-dependent greedy insertion yielded the lowest error overall. For our SHAPE_QUALITY measure, we employed a simple formula which is the product of the areas of the two triangles divided by the product of their approximate diameters. While this does not yield Delaunay triangulation when $qthresh = 1$, we believe the difference is negligible. A shape quality threshold of $qthresh = .5$ gave the best results in most cases. By varying the ERROR_ACCUM and ERROR_COMBINE procedures as described in §3.4.1, several different error measures can be tested. The error differences were slight, but empirical tests showed that the lowest error approximations typically resulted when ERROR_ACCUM used the MAX function while ERROR_COMBINE used addition – thus, a combination of L_∞ and L_2 measures. We call this the sum-max criterion.

We also tested the angle between normals (ABN) criterion proposed by Dyn *et al.* [7]. This criterion swaps edges to minimize the angle between normals of adjacent triangles. The ABN criterion is thus data-dependent in the sense that it depends on the heights at the vertices, but unlike the L_∞ and L_2-based error measures our algorithms employ, the error measure is independent of the unselected input points. In our experiments, with $qthresh = .5$, the ABN criterion gave errors that were higher than the sum-max criterion in all cases, and often higher than the Delaunay criterion as well. (We did not test the more complex ABN hybrid described by Rippa [26, p. 1136]).

Delaunay greedy insertion is compared to data-dependent greedy insertion in Figures 13 and 14. The first shows that, for a given number of points, data-dependent triangulation with our sum-max criterion finds a slightly more accurate approximation than Delaunay triangulation. The ratio of data-dependent RMS error to Delaunay RMS error is about .8 to .9 for this height field. The second figure shows the time/quality tradeoff very clearly. With either algorithm, as the number of points selected increases, the error decreases while the time cost increases. To achieve a given error threshold, data-dependent greedy insertion takes about 3–4 times as long as Delaunay greedy insertion, but it generates a smaller mesh, which will display faster.

Data-dependent triangulation does dramatically better than Delaunay triangulation on certain surfaces [7]. The

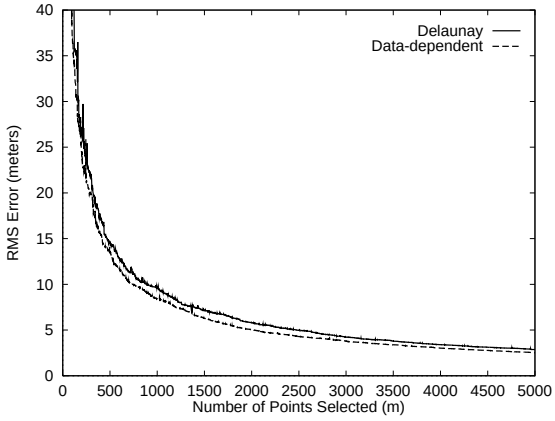

Figure 13: RMS error of approximation as vertices are added to the mesh, for Crater Lake DEM, comparing Delaunay triangulation (top curve) to data-dependent triangulation (bottom).

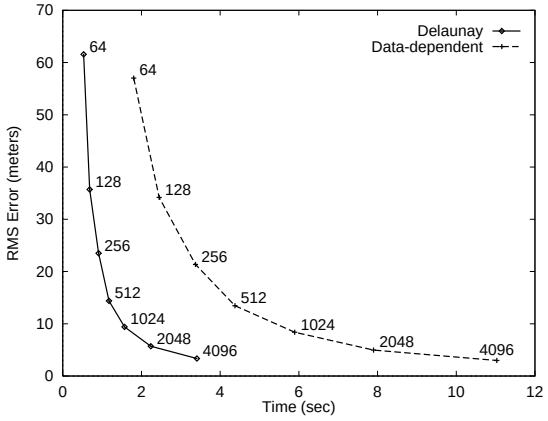

Figure 14: Time versus error plot for Delaunay and data-dependent triangulation on Crater Lake DEM. Data-dependent triangulation is slower, but higher quality. Data points are marked with *m*, the number of points selected.

optimal case for data-dependent triangulation is a ruled surface with zero curvature in one direction and nonzero curvature in another. Examples are cylinders, cones, and height fields of the form $H(x, y) = f(x) + ay$. On such a surface, if a Delaunay-triangulated approximation uses m roughly uniformly distributed vertices, then data-dependent triangulation could achieve the same error with about $2\sqrt{m}$ points using sliver triangles that span the rectangular domain.

From our empirical tests, it seems that the surfaces for which data-dependent triangulation excels are statistically uncommon among natural terrains. We conjecture that on natural terrains, data-dependent triangulation yields approximations that are only slightly higher quality than Delaunay triangulation, in general.

4.4 Sequential versus Parallel Greedy Insertion

Others have employed variants of the greedy insertion algorithm that insert more than one point on each pass. Methods that insert a single point on each pass we call *sequential greedy insertion* and methods that insert multiple points in parallel on each pass we call *parallel greedy insertion.* The words "sequential" and "parallel" here refer to the selection and re-evaluation process, not to the architecture of the machine.

Puppo *et al.* showed statistics that suggest that parallel methods are better than sequential methods, saying: "…we show the results obtained by the sequential and the parallel algorithm …Because of the more even refinement of the TIN, which is due to the introduction of many points before the Delaunay optimization, our [parallel] approach needs considerably fewer points to achieve the same level of precision" [25, p. 123].

We tested this claim by comparing our sequential greedy insertion algorithm against two variants of parallel insertion. Both variants select and insert all candidate points p such that $\text{ERROR}(p) \geq \epsilon$, where ϵ is a threshold value.

The first insertion variant, which we call *fractional threshold* parallel insertion, selects all candidate points such that $\text{ERROR}(p) \geq \alpha e_{\max}$, where $e_{\max}$ is the maximum error of all candidates. This is an obvious generalization of sequential insertion, which selects a single point such that $\text{ERROR}(p) = e_{\max}$. Fractional thresholding with $\alpha = 1$ is almost identical to sequential insertion; it differs only in that it may select multiple points with the same error value (this is closely related to the approach of Polis and McKeown [24]). If $\alpha = 0$, fractional thresholding becomes highly aggressive and selects every triangle candidate. Looking at the error graphs in Figure 12, we can see that as α increases towards 1, the approximations become more accurate and converge to sequential insertion.

The second insertion variant is the rule used by Fowler-Little and Puppo *et al.* [9, 25]. We call it *constant threshold* parallel insertion. In this case, ϵ is the constant error threshold provided by the user. Thus, on each pass we select and insert all candidate points that do not meet the error tolerance. An algorithm very similar to this [19], called Latticetin, is used by the Arc/Info geographic information system which is sold by the Environmental Systems Research Institute (ESRI).

The error curves for the constant threshold method in Figure 12 show it performing much worse than sequential insertion or fractional thresholding. Similarly, Polis and McKeown found their algorithm (a form of fractional thresholding) superior to Latticetin (a form of constant thresholding) [24].

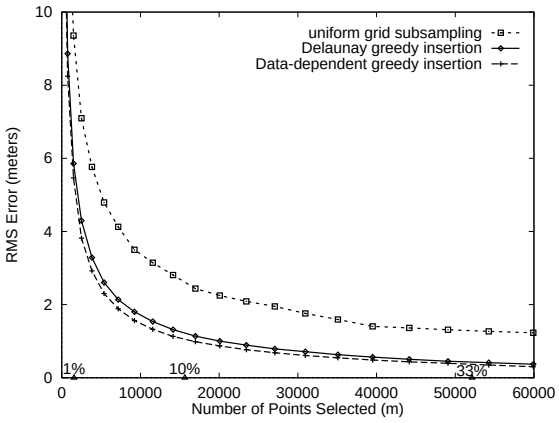

Figure 15: RMS error of approximation as vertices are added to the mesh, for two forms of greedy insertion and uniform grid subsampling, run on Ashby DEM.

When an insertion causes a small triangle to be created, it leads to a local change in the density of candidates. With the sequential method, smaller triangles are statistically less likely to have their candidates selected, because they will typically have smaller errors. In the parallel method, if the small triangles' candidate is over threshold, it will be selected, causing even more excessive subdivision in that area. Even on a simple surface like a paraboloid, which is optimally approximated by a uniform grid, the sequential method is better. On all tests we have run, sequential greedy insertion yields better approximations than parallel greedy insertion.

De Floriani seems to have reached a similar conclusion. While comparing her sequential insertion algorithm to a form of constant thresholding in which the selected points are not limited to one per triangle, she said: "parallel application of such an algorithm by a contemporaneous insertion of all points which have an associated search error greater than the tolerance and belong to the same search region, could lead to the insertion of points which are not meaningful for an improvement in the accuracy of the model" [4, p. 342].

4.5 Greedy Insertion versus Uniform Grids

We also compared greedy insertion with the simplest surface approximation method, uniform grids. In a large study, Kumler found that uniform grids (DEMs) were better than general triangulations (TINs), at least for the TIN generation methods he tested [19, p. 41]. We agree with this qualified conclusion, and go farther to point out certain aspects of his experiments that exaggerated the benefits of DEMs.

One should not conclude from Kumler's results that

DEMs are better than TINs in general, since the TIN algorithms he tested do not appear to be very good. The best DEM-to-TIN algorithm he tested was Latticetin, and we have found similar algorithms to be inferior to sequential greedy insertion (compare top three curves of Figure 12).

In addition, Kumler handicapped the TIN algorithms somewhat by comparing DEMs with n vertices to TINs with $n/3$ or $n/10$ vertices. These ratios are based on his assumption that storage size is the principal concern, and that TINs require 3–10 times the memory of DEMs. While these storage size comparisons are valid for most implementations, general triangulations can be compressed by a factor of 6–10 with very little error [6]. Also, to better understand the behavior of a simplification algorithm, we believe it is necessary to test it for a wide range of values of m/n. More importantly, we believe that, in many applications rendering speed is of much greater importance than storage space, so methods should be compared based primarily on error and speed, not on memory use.

Figures 12 and 15 show our own comparison of the errors of approximations created by sequential greedy insertion and subsampling on a uniform grid. The first figure shows coarse approximations (small values of m/n); the second shows finer approximations (larger values of m/n). Numerous comparisons of greedy insertion with subsampling were computed for the Ashby, Crater Lake, NTC, and WestUS datasets. Figure 15 shows the most representative of these four. Either form of greedy insertion is seen to generate significantly better approximations than a uniform grid. If storage size is the primary concern, and we assume that general triangulations require three times the memory of a uniform grid with the same number of vertices, then for $m/n < 1\%$ or so, uniform grids are probably best, but for larger values of m/n, a good adaptive triangulation scheme, such as sequential greedy insertion, appears better. If compression is used, then general triangulations are probably preferable in all cases.

5 Ideas for Future Research

Our experimentation has suggested several avenues for further research. These are discussed at greater length in our technical report [11].

Using Extra Grid Information. The algorithms we have described could easily be generalized to make use of additional information about the height field. Ridge lines, roads, block boundaries, and discontinuities could be preinserted, for example. The user could also be given control over point selection by allowing a weight to be assigned to each vertex [24].

Combating Slivers. Pure data-dependent triangulation generates too many slivers. It would be nice to find a more elegant solution to the sliver problem to replace the hybrid algorithm. Our current hypothesis to explain the inferiority of pure data-dependent triangulation in our algorithms is that it is caused by the short-sightedness of the greedy insertion algorithm. Sometimes more than one edge swap is required to correct a sliver problem, but our data-dependent greedy insertion algorithm never looks more than one move ahead, so it often gets stuck in local minima. Simulated annealing is one (expensive) remedy.

Discontinuities. The sequential greedy insertion algorithm does poorly when the height field contains a step discontinuity or "cliff". With the greedy insertion algorithm, a linear cliff of length k between two planar surfaces can use up about k vertices when, in fact, 4 would suffice. Cliffs similar to this arise when computer vision range data is approximated with this algorithm. This problem is caused by the short-sightedness of greedy insertion. One, somewhat *ad hoc*, solution is to find all cliffs and constrain the triangulation to follow them [1].

Dealing With Noise and High Frequencies. The greedy insertion algorithms we have described will work on noisy or high frequency data, but they will not do a very good job, as observed by us and others [8]. There are two causes of this problem. One is the simple-minded selection technique, which picks the point of highest error. Such an approach is very vulnerable to outliers. Finding a better strategy for point selection in the presence of noise appears quite difficult. A second cause is that triangles are not chosen to be the best fit to their enclosed data, but are constrained to interpolate their three vertices. Least squares fitting would solve this latter problem [26].

Better handling of noise and discontinuities would make these algorithms well suited to the simplification of computer vision range data.

A Hybrid Refinement/Decimation Approach. A technique that might permit better approximations of cliffs and better selection in the presence of noise is to alternate refinement and decimation passes, inserting several vertices with the greedy insertion approach, and then deleting a few vertices that appear the least important, using Lee's drop heuristic approach [21]. Although such a hybrid of refinement and decimation ideas resembles the algorithm of Hoppe *et al.* [18], it should be quite a bit faster since we already know how to do many of the steps quickly.

Generalization to Other Geometries. The algorithms presented here could easily be generalized from height field grids to scattered data approximation. That is, the *xy* projection of the input points need not form a rectangular grid, but could be any finite point set. This change would require each triangle to store a set of points [3, 17, 10, 8]. During re-triangulation, these sets would be merged and split. Instead of scan converting a triangle, one would visit all the points in that triangle's point set.

Generalization of these techniques from piecewise-planar approximations of functions of two variables to curved surface approximations and higher dimensional spaces [14] is fairly straightforward.

6 Summary

We have presented variants of the greedy insertion algorithm in significant detail, optimized them, analyzed their worst case and typical complexity, and presented empirical tests and comparisons.

Speed. Beginning with a very simple implementation of the greedy insertion algorithm, we optimized it in three ways: by only recalculating where necessary, by using a heap to find points of highest error, and by eliminating point location.

When approximating an n point grid using an m vertex triangulated mesh, these optimizations sped up the algorithm from a typical time cost of $O(mn)$ to $O((m+n)\log m)$. This speedup is significant in practice as well as theory. For example, we can approximate a 1024×1024 grid to high quality using 1% of its points in about 21 seconds on a 150 MHz processor. The memory requirements of the algorithm are $O(m+n)$.

Quality. Delaunay and data-dependent triangulation methods were compared. The latter is capable of higher quality approximations because it chooses the triangulation based on quality of data fit, not on the shape of a triangle's *xy* projection.

Data-dependent triangulation can be relatively fast. Had we used the straightforward algorithm, data-dependent triangulation would have been many times slower than Delaunay triangulation, since it would scan convert about twice as many input points, doing more work at each point, and it would visit each of these points twice, once for swap testing and once for candidate selection. We described a new, faster data-dependent triangulation algorithm that merges swap testing and candidate selection into one pass, saving a factor of two in cost.

With our implementation, we found that data-dependent triangulation takes about 3–4 times as long as Delaunay triangulation, and yields slightly higher quality on typical terrains. In applications where simplification speed is critical, Delaunay triangulation would be preferred, but if the quality of the approximation is primary, and the height field will be rendered many times after simplification, then the simplification cost is less important, and the data-dependent method is recommended.

Generality. The greedy insertion algorithm is quite flexible. It makes no assumptions that limit its usage to terrains. For example, it can be generalized to approximate color raster images by a set of Gouraud shaded polygons, or approximate computer vision range data with triangulated surfaces.

Comparison to Other Methods. Our Delaunay greedy insertion algorithm should produce nearly identical approximations to several previously published methods [4, 5, 26, 10, 8], but from the information available, it appears that our algorithm is the fastest both in theory and in practice.

Part of the adaptive triangular mesh filtering technique briefly described by Heller [17, p. 168] appears nearly identical in quality and asymptotic complexity to our Delaunay-based algorithm. Because the initial pass of his algorithm uses feature selection, we suspect, however, that his could be faster but that our method will produce somewhat higher quality approximations.

We have tested our sequential greedy insertion algorithm against parallel greedy insertion algorithms similar to those used by Fowler-Little and Puppo *et al.* [9, 25], and found that sequential greedy insertion yields superior approximations in all cases tested. We compared our error-based data-dependent triangulation criterion to a version of the normal-based criterion recommended by Rippa and Dyn *et al.* [26, 7], and found ours to be superior. We also compared the approximations of our algorithm against uniform grids (DEMs), and found that sequential greedy insertion generates more accurate approximations with a given number of vertices than uniform grids, contradicting some previously published conclusions [19].

Code. Portable C++ code for our Delaunay and data-dependent greedy insertion algorithms, and test data, is available by World Wide Web from http://www.cs.cmu.edu/~garland/scape or by anonymous FTP from ftp.cs.cmu.edu in /afs/cs/user/garland/public/scape.

7 Acknowledgements

We thank Michael Polis, Stephen Gifford, and Dave McKeown for exchanging algorithmic ideas with us and for sharing DEM data, and Anoop Bhattacharjya and Jon Webb for their thoughts on the application of these techniques to computer vision range data. The CMU Engineering & Science library has been very helpful in locating obscure papers. This work was supported by ARPA contract F19628-93-C-0171 and NSF Young Investigator award CCR-9357763.

References

[1] Xin Chen and Francis Schmitt. Adaptive range data approximation by constrained surface triangulation. In B. Falcidieno and T. Kunii, editors, *Modeling in Computer Graphics: Methods and Applications*, pages 95–113. Springer-Verlag, Berlin, 1993.

[2] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, Oct. 1976.

[3] Leila De Floriani. A pyramidal data structure for triangle-based surface description. *IEEE Computer Graphics and Appl.*, 9(2):67–78, March 1989.

[4] Leila De Floriani, Bianca Falcidieno, and Caterina Pienovi. A Delaunay-based method for surface approximation. In *Eurographics '83*, pages 333–350. Elsevier Science, 1983.

[5] Leila De Floriani, Bianca Falcidieno, and Caterina Pienovi. Delaunay-based representation of surfaces defined over arbitrarily shaped domains. *Computer Vision, Graphics, and Image Processing*, 32:127–140, 1985.

[6] Michael Deering. Geometry compression. In *SIGGRAPH '95 Proc.*, pages 13–20. ACM, Aug. 1995.

[7] Nira Dyn, David Levin, and Shmuel Rippa. Data dependent triangulations for piecewise linear interpolation. *IMA J. Numer. Anal.*, 10(1):137–154, Jan. 1990.

[8] Per-Olof Fjällström. Evaluation of a Delaunay-based method for surface approximation. *Computer-Aided Design*, 25(11):711–719, 1993.

[9] Robert J. Fowler and James J. Little. Automatic extraction of irregular network digital terrain models. *Computer Graphics (SIGGRAPH '79 Proc.)*, 13(2):199–207, Aug. 1979.

[10] W. Randolph Franklin. tin.c, 1993. C code, ftp://ftp.cs.rpi.edu/pub/franklin/tin.tar.gz.

[11] Michael Garland and Paul S. Heckbert. Fast polygonal approximation of terrains and height fields. Technical report, CS Dept., Carnegie Mellon U., Sept. 1995. CMU-CS-95-181, http://www.cs.cmu.edu/~garland/scape.

[12] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):75–123, 1985.

[13] Leonidas J. Guibas, Donald E. Knuth, and Micha Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992. Also in Proc. 17th Intl. Colloq. — Automata, Languages, and Programming, Springer-Verlag, 1990, pp. 414–431.

[14] Bernd Hamann and Jiann-Liang Chen. Data point selection for piecewise trilinear approximation. *Computer-Aided Geometric Design*, 11:477–489, 1994.

[15] Paul S. Heckbert and Michael Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50, Banff, Canada, May 1994. Canadian Inf. Proc. Soc. http://www.cs.cmu.edu/~ph.

[16] Paul S. Heckbert and Michael Garland. Survey of polygonal surface simplification algorithms. Technical report, CS Dept., Carnegie Mellon U., to appear. http://www.cs.cmu.edu/~ph.

[17] Martin Heller. Triangulation algorithms for adaptive terrain modeling. In *Proc. 4th Intl. Symp. on Spatial Data Handling*, volume 1, pages 163–174, Zürich, 1990.

[18] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH '93 Proc.*, pages 19–26, Aug. 1993. http://www.research.microsoft.com/research/graphics/hoppe/.

[19] Mark P. Kumler. An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica*, 31(2), Summer 1994. Monograph 45.

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

[21] Jay 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.

[22] Dani Lischinski. Incremental Delaunay triangulation. In Paul Heckbert, editor, *Graphics Gems IV*, pages 47–59. Academic Press, Boston, 1994.

[23] Edmond Nadler. Piecewise linear best L_2 approximation on triangulations. In C. K. Chui et al., editors, *Approximation Theory V*, pages 499–502, Boston, 1986. Academic Press.

[24] Michael F. Polis and David M. McKeown, Jr. Issues in iterative TIN generation to support large scale simulations. In *Proc. of Auto-Carto 11 (Eleventh Intl. Symp. on Computer-Assisted Cartography)*, pages 267–277, November 1993. http://www.cs.cmu.edu/~MAPSLab.

[25] Enrico Puppo, Larry Davis, Daniel DeMenthon, and Y. Ansel Teng. Parallel terrain triangulation. *Intl. J. of Geographical Information Systems*, 8(2):105–128, 1994.

[26] Shmuel Rippa. Adaptive approximation by piecewise linear polynomials on triangulations of subsets of scattered data. *SIAM J. Sci. Stat. Comput.*, 13(5):1123–1141, Sept. 1992.

[27] David A. Southard. Piecewise planar surface models from sampled data. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*, pages 667–680, Tokyo, 1991. Springer-Verlag.

Figure 16: Mandrill original, a 200×200 raster image.

Figure 17: Mandrill approximated with Gouraud shaded triangles created by subsampling on a uniform 20×20 grid (400 vertices).

Figure 18: Mandrill approximated with Gouraud shaded triangles created by data-dependent greedy insertion (400 vertices).

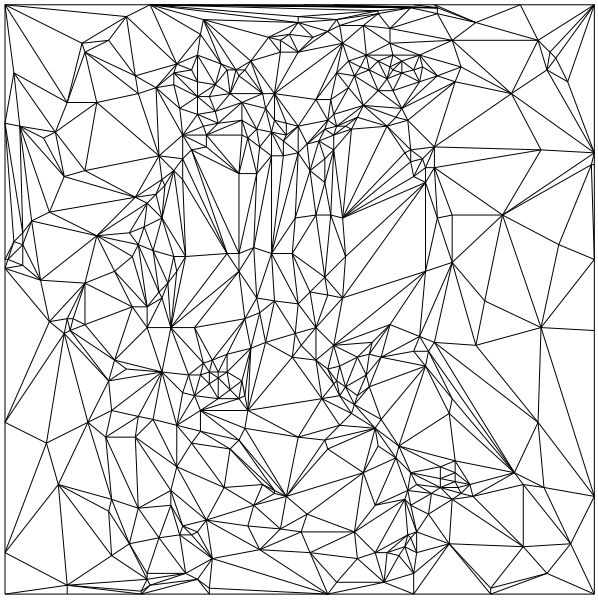

Figure 19: Mesh for the image to the left.