

# I/O-Efficient Refinement of Triangulated Terrains

An Honors Paper for the Department of Computer Science

By Jonathan Roman Todd

Bowdoin College, 2005

© 2005 Jonathan Roman Todd

# Abstract

This project investigates the use of triangulations (TINs) to represent and compute on very large terrains in Geographic Information Systems (GIS). While grid-based terrains are highly researched, both in theory and in practice, the exploration of triangulated terrains has been mostly theoretical. We study the problem of simplifying a terrain into a TIN that approximates it within a specified error threshold. Our main contribution is an I/O-efficient terrain simplification algorithm that converts grids into TINs. We present an implementation of our algorithm and analyze its performance and scalability to large datasets. We port the algorithm into the open source GIS package GRASS for use and experimentation by the GIS community. This is the first grid simplification system that incorporates theoretical and practical applications on triangulated terrains while maintaining scalability. Finally, we show how to extend our algorithm to simplify arbitrary point-data, and explain how this will be useful for processing LIDAR data.

# Contents

<b>Abstract</b>	<b>ii</b>
<b>List of Tables</b>	<b>vi</b>
<b>List of Figures</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Outline of the Paper . . . . .	2
<b>2 Background</b>	<b>3</b>
2.1 Terrain Models . . . . .	3
2.2 Comparison of Terrain Models . . . . .	4
2.3 Data Origins . . . . .	6
2.4 Delaunay and Voronoi Properties . . . . .	6
2.4.1 Definition of a Triangulation . . . . .	7
2.4.2 Delaunay Triangulation . . . . .	7
2.4.3 Voronoi Diagrams . . . . .	10
2.5 TIN Conversion and Simplification . . . . .	11
2.5.1 Refinement . . . . .	11
2.5.2 Decimation . . . . .	12
2.6 Flow Modeling . . . . .	15
2.6.1 Flow on Grids . . . . .	15
2.6.2 Flow on TINs . . . . .	16
2.7 Large Data Sets and the I/O-Model . . . . .	18
2.7.1 Realistic I/O-efficiency Example . . . . .	19

2.8	Project Overview . . . . .	20
<b>3</b>	<b>TIN Simplification</b>	<b>22</b>
3.1	Overview . . . . .	22
3.2	Tiling Raster Files . . . . .	23
3.3	Refining Tiles . . . . .	24
3.4	TIN Structure in Memory . . . . .	27
3.5	Combining Tiles . . . . .	28
3.5.1	Maintaining Boundary Consistency . . . . .	29
3.5.2	Maintaining Delaunay with Tiles . . . . .	30
<b>4</b>	<b>TIN Traversal</b>	<b>31</b>
4.1	Defining Edges . . . . .	31
4.2	Degenerate Cases . . . . .	32
4.3	Choosing the Next Triangle . . . . .	33
4.4	Using The TIN Traversal Algorithm . . . . .	34
<b>5</b>	<b>Experimental Results</b>	<b>37</b>
5.1	Datasets and Test Platform . . . . .	37
5.2	Effects of Tiling . . . . .	38
5.3	Grid vs. TIN Comparison . . . . .	41
5.4	Effects of Maintaining Delaunay . . . . .	43
<b>6</b>	<b>Conclusions and Future Work</b>	<b>45</b>
6.1	Conclusion . . . . .	45
6.2	LIDAR to TIN . . . . .	46
6.3	Flow Modeling on TINs . . . . .	46

6.3.1	Triangle-based Flow . . . . .	47
6.3.2	Vertex-based Flow . . . . .	47

# List of Tables

5.1	Characteristics of terrain datasets. . . . .	37
-----	--	----

# List of Figures

2.1	Examples of DEM representations. . . . .	3
2.2	Grid vs TIN on flat areas. . . . .	5
2.3	Convex Hull example. . . . .	7
2.4	Circumcircle Property. . . . .	8
2.5	Adding points to a triangulation. . . . .	9
2.6	Randomized Incremental Refinement . . . . .	9
2.7	Voronoi Regions. . . . .	10
2.8	Interpolating the error of a point in a triangle. . . . .	12
2.9	Refinement: Heller’s Heuristic. . . . .	13
2.10	Initializing Decimation. . . . .	13
2.11	Decimation: Lee’s Drop Heuristic. . . . .	14
2.12	D8 flow modeling on grids. . . . .	15
2.13	Channel edge example. . . . .	17
2.14	Memory hierarchy. . . . .	18
3.1	Initializing TIN for Refinement . . . . .	24
3.2	Our Refinement Algorithm. . . . .	26
3.3	TIN Structure. . . . .	28
3.4	Tile boundaries. . . . .	29
4.1	Example of TIN traversal path. . . . .	32

4.2	Edge classifications . . . . .	33
5.1	Tile vs Untiled Comparison On Sierra . . . . .	39
5.2	Tile vs Untiled Comparison On Cumberlands . . . . .	39
5.3	Tile vs Untiled Comparison On Appalachians . . . . .	40
5.4	Grid vs. TIN Point Comparison (0.1% Error) . . . . .	41
5.5	Grid vs. TIN Space Efficiency (0.1% Error) . . . . .	42
5.6	Effect of Maintaining Delaunay on Triangles . . . . .	43
5.7	Triangulation comparison. . . . .	44
6.1	Triangle based flow model. . . . .	47
6.2	Vertex based flow model. . . . .	48

# Chapter 1

## Introduction

The study of Geographic Information Systems (GIS) is becoming increasingly important to the computer science community due in part to the increase in availability of data and the need for more efficient computations on this digitized data. Unprecedented quantities of data are available from numerous sources, such as the U.S. Geological Survey (USGS) web-site [2] and the NASA EOS project [1]. The availability of this data is largely due to remote sensing projects such as NASA's shuttle radar topography mission (SRTM), which mapped 80% of the Earth's land mass at 30 meter resolution amounting to roughly 10 terabytes of data [18]. SRTM, USGS and other sites provide free access to terabytes worth of data.

Much work has been done in GIS with fast computations on moderately sized datasets, however, as terrain data becomes more available at higher resolutions we find that many of these algorithms do not scale. This is because while they make efficient use of the CPU, they are typically designed with the assumption that all data being processed can fit into memory. Since this assumption does not hold for massive datasets, data needs to be swapped between the hard-disk and memory as needed by the algorithm. This operation can be extremely time intensive if not planned for. Research in the area of I/O-efficient algorithms has shown that designing algorithms that make efficient use of the memory/disk transfers can significantly reduce the running time on massive data sets.

## 1.1 Outline of the Paper

This paper will first provide a thorough background for the theoretical basis of our project in Chapter 2. Concepts and terminology will be introduced for terrain modeling, flow modeling, and the I/O model in this section. We will then discuss the details our implementation of TIN simplification in Chapter 3. In Chapter 4, we will present details of our implementation of TIN traversal and how we modify the theoretical algorithm to work in practice. Next, we present an empirical analysis of our I/O-efficient algorithm in Chapter 5 by comparing its performance under various constraints. Finally in Chapter 6 we propose methods for flow modeling on TINs and discuss areas for future work.

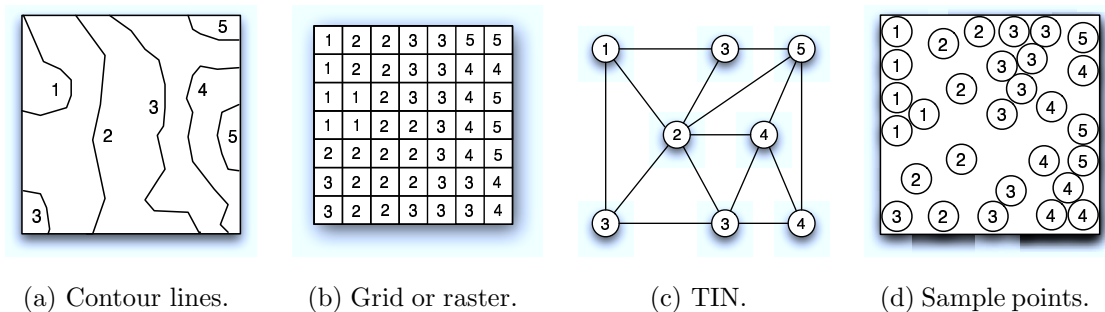
# Chapter 2

## Background

### 2.1 Terrain Models

While there is data available on many different features of terrain, the most prevalent is elevation data. These elevations are represented in a *digital elevation model* (DEM) which is a continuous function in two variables such that  $z = f(x, y)$ . In this function,  $x$  and  $y$  typically correspond to longitude and latitude respectively, and  $z$  most often represents elevation though it can also represent other features. DEMs can be used to model geomorphological, hydrological, and biological processes on terrains like, sediment flow, erosion potential, soil water content, and plant species distribution [17].

Elevation data can be modeled by one of four representations, *contour lines*, *grids* (*rasters*), *triangulated irregular networks* (TINs), or *point samples*; see Figure 2.1. A contour line is typically represented by a sequence of  $(x, y)$  values that are connected to form a polygon at a given elevation. The contour line model is then a collection



**Figure 2.1:** Examples of DEM representations.

of contour lines at various elevations over the terrain. While contour lines are very useful for importing data from paper maps and displaying a three-dimensional terrain on a two-dimensional map, they do not lend themselves to applications that we are concerned with, and are outside the scope of this project.

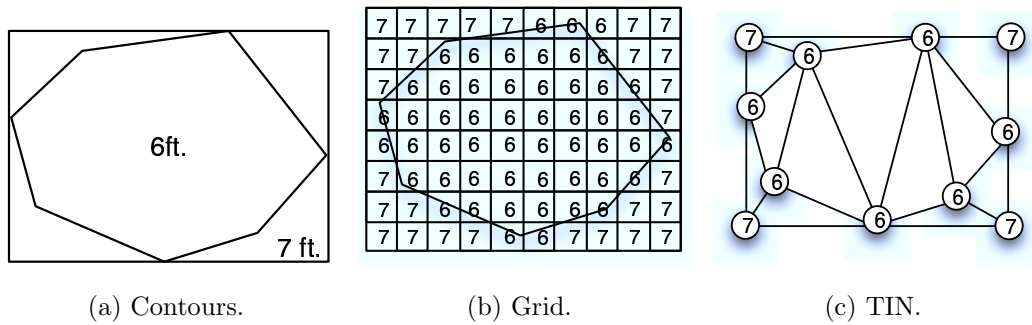
Grids, which are the most common representation to date, can be defined as a uniform tessellation of square cells over a terrain where each cell possesses one elevation value. Using this model one can extract the height at any given point from the elevation value of the surrounding square.

TINs are a common way to represent a terrain with the benefit that they do not need a fixed resolution. A TIN is a *triangulation* which is a non-uniform and non-overlapping tessellation of triangles over a surface, see Section 2.4 for a formal definition. There are various properties that can be maintained in the construction of a triangulation. The most commonly used triangulation is the *Delaunay triangulation* (DT); the Delaunay triangulation enforces the desirable property that the triangles are fatter which makes for better formed triangulations [22]. Refer to Figure 5.7 for a visual example.

Another representation of a DEM is a point sample. This structure is simply a set of  $(x, y, z)$  values that are not necessarily uniformly distributed over the terrain. Data of this form is usually converted into another form for computations as it is not particularly useful in its unorganized state.

## 2.2 Comparison of Terrain Models

Grids are the most commonly used format for elevation models since they are simple in structure and are readily available in this form from remote sensing devices. Furthermore, grid data is relatively easier to write algorithms for since grids have a fixed resolution which can easily be stored in an indexable data structure like an



**Figure 2.2:** (a) Contours of a relatively flat area. (b) Grid representation with 80 cells. (c) TIN representation with 11 points and 12 triangles.

array. This fixed size property is also a disadvantage since they over-represent flat areas, and under-represent varied terrains. This is to say that grids cannot adapt to the variation of terrains due to their uniform nature.

TINs are in many ways the antithesis of grids. They are much more complicated to implement and query; however, they are capable of representing the same terrain as a grid with fewer points. TINs are able to represent terrains with fewer points since they only store the bounding points for flat areas and can provide higher resolution in areas with varied elevation. Figure 2.2 gives a more concrete example for how TINs can represent flat areas with fewer points. For the given contour which could be a lake for example, the grid representation stores 80 elevation points while the TIN only need to store the 11 boundary points of the lake and it's bounding box.

This ability for TINs to use fewer points to represent a terrain comes with the trade-off that TINs take up more space per point since they abstract more about the terrain. Further advantages and disadvantages of various DEM representations can be found in [21, 17].

## 2.3 Data Origins

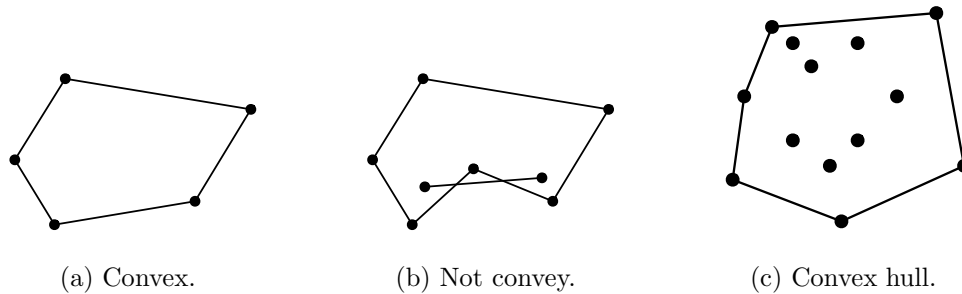
Although we have mentioned how DEMs can be represented, it is important to know where the data is coming from originally and what implications that has on selecting a representation. The first form of elevation data, before the use of computers, was gathered by hand and drawn into contour lines on maps. From the contour lines one could interpolate the elevation at a given point in the map with a marginal accuracy that was very much dependent on scale. When this data was being digitized for GIS applications, the obvious choice for representation was the contour line model.

With the more recent release of remote sensing data by NASA and USGS, the preferred representation became grids. As noted above, this was partially due to the fact that this was how the data was coming in from the satellites and also due to simplicity.

A third and increasingly popular method for collecting very high resolution data is through the use of airplanes that map out the terrain in a format called *LIDAR* (LIght Detection And Ranging). This data arrives as a point sample of the form  $(x, y, z)$  with no particular structure. While LIDAR data can be transformed into a grid form, a TIN would be a much more accurate and likely more efficient means of representation since they can guarantee accuracy without the need for superfluous data points.

## 2.4 Delaunay and Voronoi Properties

Before we talk about building a triangulation from a grid we need to define some terms and properties that we use in our triangulations. We will discuss the Delaunay property, a common convention used to create well formed triangles and we will present a definition for a Voronoi diagram and discuss how that relates to a Delaunay



**Figure 2.3:** (a) Example of a convex polygon. (b) Example of a polygon that it is not convex because a line can be drawn between two points which is not entirely inside the polygon. (c) Example of a convex hull of a set of points.

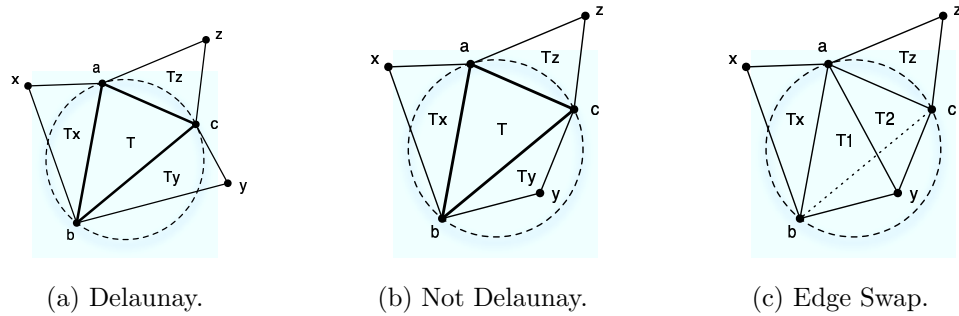
triangulation.

### 2.4.1 Definition of a Triangulation

Consider a point set  $S \subseteq \mathbb{R}^2$ . A set  $S$  is called *convex* if for every pair of points  $p, q$  in  $S$ , the line segment  $pq$  is completely contained in  $S$ , see Figure 2.3(a,b). The *convex hull* of  $S$  is the smallest convex set containing  $S$ . Intuitively, it is the shape taken by a rubber band that has been stretched around the set  $S$  and released to conform to the outermost points in  $S$ , see Figure 2.3(c). A *triangulation* of a set of points  $S$  is then defined as a set of line segments whose endpoints are in  $S$ , which only intersect each other at endpoints, and which partition the convex hull of  $S$  into triangles.

### 2.4.2 Delaunay Triangulation

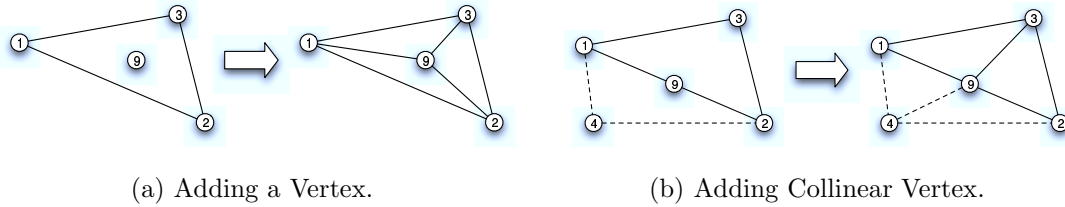
There are many triangulations for a point set  $S$ . The most commonly used is Delaunay triangulation (DT). We can define a Delaunay triangle using the circumcircle claim as follows: Let  $S \subseteq \mathbb{R}^2$  be a finite and in general position, and let  $a, b, c \in S$  be three points. Then  $abc$  is locally *Delaunay* if and only if the circumcircle of  $abc$  does not contain any other points in  $S$ . This property becomes more clear looking at



**Figure 2.4:** Circumcircle Property. (a) Example of a Delaunay triangle. (b) Example of a triangle that is not Delaunay. (c) Example of an edge swap to fix the triangle in (b).

Figure 2.4. As seen in Figure 2.4(a), the circumcircle to triangle  $abc$  does not contain any other vertices therefore it is *locally* Delaunay. In Figure 2.4(b) we see that  $abc$  is not locally Delaunay because  $y$  lies inside its circumcircle. A triangulation  $S$  is a *Delaunay triangulation* if all its triangles are locally Delaunay.

Given a point set  $S$  of  $n$  points in the plane, there are several known algorithms that build a Delaunay triangulation in  $O(n \log n)$  time. One of them is randomized incremental construction (RIC) [6]. The basic idea in RIC is to start from a set  $S$  and build a Delaunay triangulation  $D$  by adding points one at a time and enforcing that all triangles are locally Delaunay. Define  $S_i = \{x, y, z, p_1, p_2, \dots, p_i\}$  and let  $D_i$  be the Delaunay triangulation of  $S_i$ .  $D_0$  is made up of one triangle  $xyz$  which contains all other points, that is to say that  $xyz$  is the convex hull of  $S$ . For each point  $p_i$  in  $S$ , locate the triangle  $r_{i-1}$  in  $D_{i-1}$  which contains the point  $p_i$ . Then add  $p_i$  to  $D$  by splitting the triangle that contains it ( $r_{i-1}$ ) into two or three triangles, see Figure 2.5 for the two cases. If an added triangle is not locally Delaunay then an edge flip is performed. As seen in Figure 2.4(c), we can flip the internal edge to the quadrilateral  $abcy$  such that we have two new triangles,  $aby$  and  $acy$ . Since this swap may affect other triangles in the triangulation we will need to verify that the new triangles are



**Figure 2.5:** Examples of adding a point to a triangulation. (a) Example of adding a vertex which lies inside the triangle. (b) Example of adding a vertex to a vertex on the edge of the triangle (the collinear/degenerate case).

```

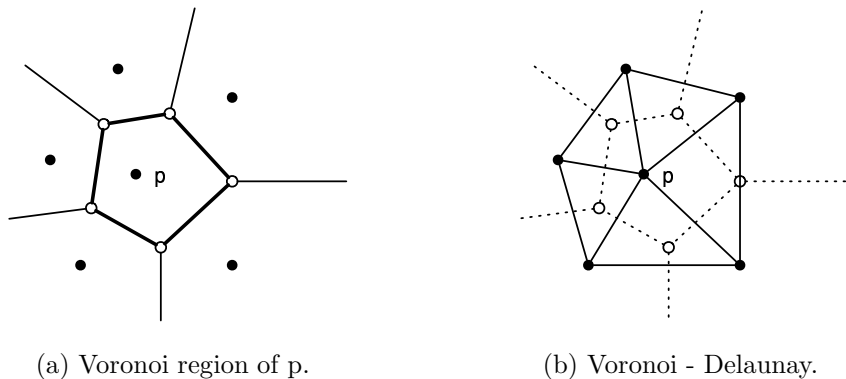
for  $i = 1$  to  $n$  do
  find  $r_{i-1} \in D_{i-1}$  containing  $p_i$ 
  add  $p_i$  by splitting  $r_{i-1}$  into two or three
  while  $\exists ab$  not locally Delaunay
    flip  $bc$  to other diagonal  $ay$ 
  endwhile
endfor

```

**Figure 2.6:** Randomized Incremental Refinement

Delaunay. If they are Delaunay then we can stop checking. If not then we continue to swap and check until all new triangles are Delaunay, see Figure 2.6.

It can be shown that this algorithm runs in  $O(n \log n)$  worst-case. To analyze its running time one can break up the algorithms into two parts; locating a point in the Delaunay triangulation and maintaining Delaunay for that point. Finding a single point takes  $O(\log n)$  using a point location structure [3], so it will take  $O(n \log n)$  to locate all points in the Delaunay triangulation. Then it can be proved that there are at most  $3n$  flips needed to maintain Delaunay on  $n$  points [6], so it will take  $O(n)$  to maintain Delaunay since each flip takes constant time. Thus the overall runtime is  $O(n \log n) + O(n) = O(n \log n)$ .



**Figure 2.7:** Voronoi Regions. (a) Example of the Voronoi region for  $p$ . (b) Example of a dual diagram which displays the relationship between the dotted Voronoi edges and the solid Delaunay edges.

### 2.4.3 Voronoi Diagrams

Like Delaunay triangulations, Voronoi diagrams are also used to spatially segment a set of points. In a Voronoi diagram, each vertex has a surrounding *cell* or *Voronoi region*. This region can be defined as follows. Let  $S \subseteq \mathbb{R}^2$  be a set of  $n$  points. The Voronoi region of  $p \in S$  is the set of points  $x \in \mathbb{R}^2$  that are at least as close to  $p$  as to any other point in  $S$ . As we can see in Figure 2.7(a), the boundaries of the Voronoi edges are equidistant to  $p$  and their other respective nearest point. That is to say that any point inside this polygon is closer to  $p$  than any other point in the set of points  $S$ .

The Delaunay triangulation and Voronoi diagram of a point set are *dual* to each other. That is, it can be shown that a Delaunay triangulation is formed from a Voronoi diagram by connecting points  $p, q \in S$  if and only if their Voronoi regions intersect along a common line segment (refer to Figure 2.7(b)).

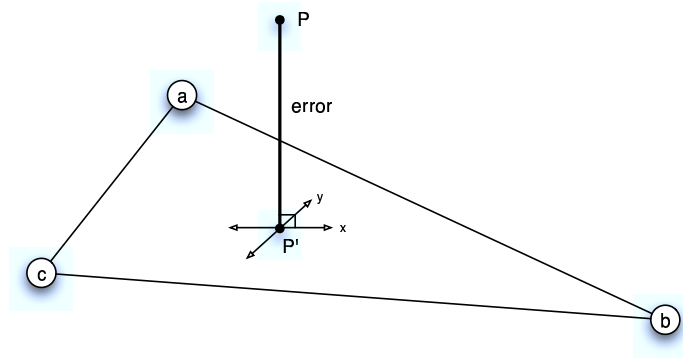
## 2.5 TIN Conversion and Simplification

In GIS applications it is common to need to convert one terrain representation model to another and to reduce the number of points in the representation of a terrain. This process is known as *terrain simplification*. Let  $S$  be a set of sample points of a terrain and  $DT(S)$  be the Delaunay triangulation of the  $x, y$  projection of  $S$ . The goal of terrain simplification is to find a subset  $Q$  of  $S$  which, when triangulated, approximates  $S$  within a desired error  $\epsilon$ . That is, the distance between  $S$  and  $DT(Q)$ ,  $d(S, DT(Q))$ , should be less than  $\epsilon$ . We define  $d(S, DT(Q)) = \max_{p \in S} \{ d(p, DT(Q)) \}$ , that is, the difference between the original set of  $S$  and the new set  $Q$  is equal to the error of the point in  $S$  that has the greatest distance from the Delaunay triangulation  $Q$ . The most commonly used distance is the vertical distance between the point  $p$  and the interpolated approximation of  $DT(Q)$  at that point (refer to Figure 2.8).

There are a variety of methods for terrain simplification. We will just mention two of the most common methods; Heller's *refinement* heuristic [11] and Lee's Drop Heuristic also known as *decimation* [13, 14]. Both methods not only allow the grid to be converted into a TIN but they also allow the terrain to be simplified by removing points that don't add some level of accuracy to the representation. This threshold for accuracy or *error* determines how much difference will be allowed between the original grid and the simplified TIN.

### 2.5.1 Refinement

The *refinement* approach starts with a set of points  $P$  and an initial set  $S$  containing only the corner vertices of  $P$ . From these four points we create a Delaunay triangulation with two triangles,  $DT(S)$  where each triangle has a list of points which lie on or in its  $x, y$  plane (see Figure 3.1). All points will lie in precisely one of the two initial triangles. We go through  $P$  and for every  $p \in P$  we find which of the current



**Figure 2.8:** Interpolating  $P$  on  $abc$  produces  $P'$ . The error of  $P = |P - P'|$  [11].

triangles in  $DT(S)$  contains  $p$ . The error for each point is computed and a pointer to the vertex with the highest error is stored by the triangle. The error of a point  $P$  inside triangle  $abc$  is defined as the vertical distance between  $P$  and the interpolated point  $P'$  on triangle  $abc$ ; refer to Figure 2.8.

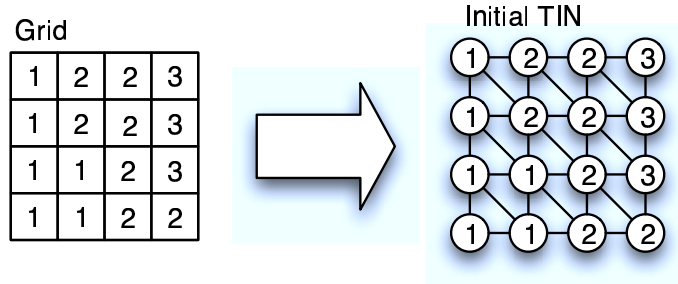
The Refinement algorithm then finds the point with the largest error and adds it to the triangulation  $S$  maintaining the Delaunay property if and only if the error is greater than some given parameter  $\epsilon$ . When adding a point  $p$  to  $S$  we remove the existing triangle containing  $p$  and add either two or three triangles depending on whether or not  $p$  is on an edge (see Figure 2.5). Once the triangles are added, refinement continues the loop on the point with the next largest error. If there are no more points with error greater than  $\epsilon$  then refinement is done. The overall algorithm is outlined in Figure 2.9.

## 2.5.2 Decimation

The *decimation* method for TIN simplification is the reverse of refinement: instead of adding points to the initial TIN, decimation takes the input points and builds a TIN where every point is included in the TIN (see Figure 2.10). Then each vertex  $v$  of the TIN is temporarily removed from the TIN and the vertical distance between

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

**Figure 2.9:** Refinement: Heller’s Heuristic [13, 14]



**Figure 2.10:** Example of initializing a TIN from a grid in decimation.

the removed vertex and the simplified terrain is computed. If the distance is less than a given error  $\epsilon$ , then the point is removed from the TIN permanently and the heuristic progresses on to the next vertex. Otherwise, that vertex is put back into the triangulation and the heuristic moves on to another vertex.

In either method, if  $\epsilon$  is zero then all of the input points will be included in the triangulation. If  $\epsilon$  is greater than zero then the TIN will have fewer points in flatter areas where difference in vertical distance is low. It can be shown that both algorithms run in  $O(n \log n)$  average time where  $n$  is the number of points being considered for the triangulation [21]. Note that decimation will run relatively faster than refinement when the desired error margin  $\epsilon$  is low because decimation will have

1. For each vertex  $v$  in the TIN:

- Temporarily remove  $v$ .
- Compute the Delaunay triangulation of the appearing polygon.
- Determine the vertical distance  $error(v)$  of  $v$  to the new TIN.

Store  $error(v)$  for each vertex  $v$  sorted in a balanced binary tree  $\mathcal{T}$ . At each node of  $\mathcal{T}$  storing some  $error(v)$ , store a pointer to the vertex  $v$  in the TIN. At  $v$  we store a pointer back to the corresponding node in  $\mathcal{T}$ .

2. Consider the node with smallest  $error(v)$  in  $\mathcal{T}$ . If it is greater than the prespecified maximum error, the algorithm stops. Otherwise it proceeds with the next step.

3. Remove the node storing the smallest  $error(v)$  from  $\mathcal{T}$ . Remove the corresponding vertex  $v$  from the TIN structure. Let  $w_1, \dots, w_j$  be the vertices adjacent to  $v$ . Re-triangulate the polygon defined by  $w_1, \dots, w_j$  using the Delaunay triangulation.

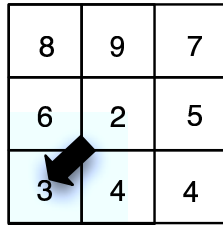
4. For every vertex  $w_i \in \{w_1, \dots, w_j\}$ :

- Remove the node that stores  $error(w_i)$  from  $\mathcal{T}$ .
- Recompute the vertical distance to the terrain if  $w_i$  were removed as we did in the first step.
- Insert the new  $error(w_i)$  in  $\mathcal{T}$ .

Continue at step 2.

**Figure 2.11:** Decimation: Lee's Drop Heuristic [21]

to make far fewer changes to its initial TIN. Likewise, refinement will be faster than decimation where  $\epsilon$  is large since there will be fewer points to add. The one major difference between the two is that decimation cannot guarantee that all points in the final TIN have errors smaller than  $\epsilon$  where refinement can. This is because points in the decimation process are removed based their error given by neighboring triangle's vertices. However, later one of these neighboring triangles may be removed which may increase the error above  $\epsilon$ .



**Figure 2.12:** D8 flow modeling on grids.

## 2.6 Flow Modeling

One of the most important processes that can be simulated on a DEM is water flow. The basic question is where does water go if it rains? What are the areas susceptible to flooding? What is the drainage network and watershed (area of a drainage that flows to the same point) for a given point in the terrain? All these questions and many others can be answered by modeling flow.

Flow can be modeled using two basic parameters, *flow direction* and *flow accumulation*. Assuming that water flows uniformly over a terrain and that water flows downhill, the flow direction at a given point represents the direction that water flows at that point. Flow accumulation of that point then represents how much water flows through that point if water follows the calculated flow directions for all points in terrain. Intuitively, flow accumulation is low on peaks and high in valleys and channels. Thus flow accumulation naturally models the drainage network of a terrain (all points with a flow accumulation value greater than some threshold are rivers).

### 2.6.1 Flow on Grids

Among the various representations for DEMs mentioned before, grids and TINs are best suited for flow modeling because they maintain information about topology. Flow modeling has been widely researched and implemented on grids. The most common, and simple, method assigns a flow direction of a grid cell to the lowest

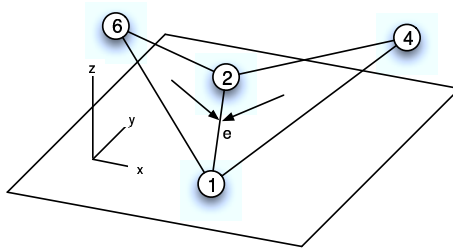
down-sloping neighbor cell (D8 flow modeling) [12] (see Figure 2.12). If a cell has no downward neighbors then it is a *pit* and the flow stops there. Flow direction defined this way assume that water flows down hill using the steepest path and thus cannot create cycles. Once flow direction for every grid cell has been computed, we can then compute the flow accumulation for a given cell by counting all the grid squares which have a flow path into it.

Although the D8 flow model on grids is the most commonly used method for flow computation, it possesses some limitations. Flow is restricted to 45 degree drainage directions, and the grid's regular structure can lead to bias in some processes.

## 2.6.2 Flow on TINs

While many of the intricacies of modeling flow on grids have been left out [5], it should be apparent that modeling flow on TINs is even less straightforward. This is due to the fact that there are no regularly sized partitions to assign flow to. Furthermore, if one is to try to assign flow from a triangle, it is not as clear as to where it should go. The flow may go from a triangle to vertex or an edge or from a vertex back to a triangle. Furthermore, various special cases of terrain like saddle points and ridges create ambiguities as to how to assign flow. To date, there have been two theoretical approaches to modeling flow on TINs.

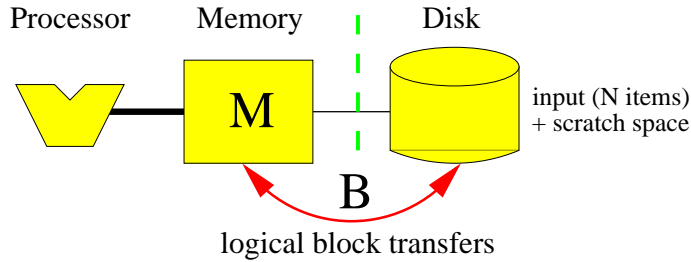
One method for modeling flow on TINs naturally extends the grid D8 approach; by treating the terrain as a *discrete* graph where flow is assigned to vertices and routed down the steepest incident edge. One such method was proposed by Tucker et. al [20]. Assuming that one starts with a Delaunay triangulation of the terrain, one computes a Voronoi diagram; that is for each vertex in the terrain we compute the surrounded Voronoi cell (refer back to Figure 2.7). In that model a point represents the Voronoi cell around it. Flow is assigned to vertices and is routed along the



**Figure 2.13:** Example of a channel edge (valley). The numbers represent elevations in the  $z$  axis.

steepest incident edge to the node. The flow accumulation at a given point  $x$  is equal to the sum of the Voronoi cells areas for all points, including  $x$ , which flow to  $x$ . This system has the limitation that flow paths are forced to follow TIN edges instead of the true steepest downward sloping path. This could be improved by using a more general flow routing procedure, like having multiple flow paths.

Another explored method for modeling flow on TINs takes a *continuous* approach. This model has been explored more rigorously in the computational geometry community [24, 16, 15], and offers proofs for consistency and bounds on operations. The basic idea behind this method is to assign the flow direction of a point to the steepest downward path regardless of whether that path goes *through* a triangle or along an edge. A point in the terrain is defined to be on the drainage network if the surface integral of the area that flows to it is not equal to zero [15]. An edge  $e$  can be defined as a channel edge or valley, if the normals of both incident triangles point to  $e$ , refer to Figure 2.13. Yu and McAllister [24, 16, 15] show that the drainage network consists of the steepest downslope path of all channel edges; thus one does not need to trace down steepest paths from *all* points in the TIN, but only for the points on channel edges. Tracing down a steepest downslope edge takes time linear in the size of the path. It is shown however that the size of the drainage network can be  $O(n^3)$  in the worst-case (i.e. a flow path can intersect a triangle many times).



**Figure 2.14:** Memory hierarchy. This show how data is transferred between disk and memory through blocks.

## 2.7 Large Data Sets and the I/O-Model

Grid-based data received from remote sensing is already producing datasets too large to work with. Now with the low cost, high resolution collection of LIDAR data being available, there is even a greater need for algorithms that can handle this massive quantity of data.

Currently most algorithms for GIS are designed to optimize CPU use under the *RAM model* of computation. The main assumptions of the RAM model are that all data fits in memory and accessing any data in memory costs the same amount of time. These assumptions are not true for datasets so large that they don't fit into memory. With very large datasets, even “linear” time algorithms can face extremely long run times. This is due to the inefficiency of hard disk access which can be 1000 times slower than memory [23]. Since it takes a long time for the hard drive to access the data, it brings large blocks of information into memory based on the heuristic that other information in that block will be used at some point close in time. In order to create algorithms that can make efficient use of both disk and CPU one must consider that data may not fit into main memory and specifically optimize for disk I/O.

The most commonly used model to design efficient algorithms for large data is the two level I/O-model [3], see Figure 2.14. This model takes into account the

large transfer block sizes which are used to amortize the extremely long access time of disks relative to that of internal memory. The model defines parameters as follows:

$N$  = number of elements in the problem (standard in RAM-model)

$M$  = number of elements that can fit into internal memory

$B$  = number of elements per disk block

such that  $M < N$  and  $M > 2B$ . One *Input/Output* (or *I/O*) is defined as the transfer of one consecutive block of information between disk and memory. The I/O-complexity of an algorithm in this model is given by the number of I/Os needed to solve the problem. An algorithm is called I/O-efficient for external memory if it attempts to optimize its I/O-complexity. The basic I/O-complexity bounds that come up in the analysis of algorithms are scanning and sorting. The *scan* or linear bound,  $\text{scan}(N) = \theta(N/B)$ , is the number of I/Os needed to read  $N$  contiguous items from the disk. The *sorting* bound,  $\text{sort}(N) = \theta((N/B) \log_{M/B}(N/B))$ , represents the number of I/Os required to sort  $N$  items contiguously on disk [3]. For realistic values for  $B$  and  $M$ ,  $\text{scan}(N) < \text{sort}(N) \ll N$ . In practice the difference between  $\theta(N)$  I/Os and  $\theta(\text{sort}(N))$  I/Os can be large. The goal of I/O-efficient algorithms is to take advantage of the big block size and process the entire block while loading the block only once. For surveys of I/O-model results see [23, 4].

### 2.7.1 Realistic I/O-efficiency Example

Suppose the following:

$N = 2^{30}$  items at 1 byte each (1 Gb of data)

$B = 2^{14}$  (16Kb block size)

$M = 2^{29}$  (512Mb memory)

$O(\text{scan}(n))$  I/O's =  $2^{16}$  block transfers = approximately 16 second running time

$O(n)$  I/O's =  $2^{30}$  block transfers = approximately 280 hour running time

In this realistic example,  $O(\text{scan}(n))$  I/O's is 62,500 times faster than  $O(n)$  I/Os.

## 2.8 Project Overview

Now that we have some understanding of the various terrain models and the I/O-model we are ready to apply the theory to a real problem in GIS. Most of the GIS community currently uses grids to represent DEMs collected from various sources. Grids are less efficient spatial representations than TINs. Furthermore, there is a growing use of LIDAR data which lends itself better to TINs. Also, LIDAR data is very high resolution which creates very large datasets. These facts reveal a need for work on I/O-efficient algorithms for TINs. While a great deal of work has been done in the area of theoretical algorithms for TINs [21], there is little progress being made with implementation in the GIS community.

In this paper we present an I/O-efficient algorithm for grid to TIN conversion and refinement. Our algorithm has been ported into the open source package GRASS (Geographic Resources Analysis Support System) where other users in the GIS community can benefit from it. We present an experimental algorithm that (1) compares TINs versus grids in terms of space efficiency and (2) compares the standard (non-I/O-efficient) TIN refinement algorithm from Section 2.5.1 and our I/O-efficient refinement algorithm. Our experiments show that TINs are superior to grids for reasonably low errors, and that an I/O-efficient algorithm drastically outperforms the standard in-memory refinement algorithm. Our approach can be extended to arbitrary point data, and we propose how to extend our existing implementation to work with LIDAR data. Finally, as an example of TIN application, we explore modeling flow on TINs and propose two methods for assigning flow direction and computing

flow accumulation on TINs for future work.

# Chapter 3

## TIN Simplification

In this chapter we describe our I/O-efficient algorithm for grid (raster) to TIN simplification. We first provide an overview and then discuss details of the algorithm, prove its I/O-complexity and highlight areas for improvement. For the rest of the paper we make the assumption that the input grid has  $\sqrt{n} \times \sqrt{n} = n$  elements. Although a grid need not be square, we omit the details for simplicity.

### 3.1 Overview

Our implementation of TIN simplification is based on the *refinement* heuristic discussed earlier in Section 2.5.1. This method starts with a TIN of four corner points and two bounding triangles for the terrain and incrementally adds the points to the triangulation which have an error above some prespecified  $\epsilon$ . We chose this method over decimation since refinement can actually guarantee that every point in the triangulation has an error above  $\epsilon$ .

A straightforward implementation of the refinement algorithm takes  $O(n)$  I/Os if all  $n$  elements do not fit into memory; this is because points are added to the triangulation in random order and in the worst-case each of the  $n$  points in the terrain will need  $O(1)$  I/Os to be read from disk. For larger datasets, and smaller errors, the standard refinement algorithm does not scale; for empirical evidence of this see our experimental analysis in Chapter 5.

In order to obtain an I/O-efficient refinement algorithm we use the standard technique of *tiling*. We partition the grid into tiles, and simplify each tile individually. The size of a tile is chosen such that the tile and all the other data structures needed

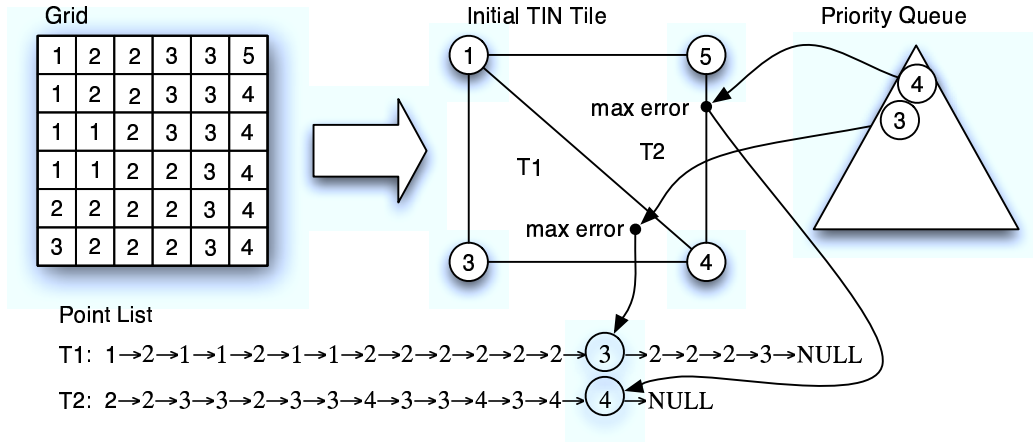
by the refinement algorithm fit into memory on the machine. Thus, at any point in the refinement process, information needed by refinement fits in memory.

We prove that the I/O complexity of our refinement algorithm is  $O(\text{scan}(n))$ , which is a factor of  $B$  improvement over the straightforward implementation of refinement. From the realistic example discussed in Chapter 2, a factor of  $B$  improvement can be the difference between a 16 second and a 280 hour runtime.

## 3.2 Tiling Raster Files

The grid data that we are using comes in the Arc ASCII file format (.asc). Each Arc ASCII file contains a header, and the data itself, which is essentially a list of elevation points. The header provides a number of rows, number of columns, latitude, longitude, and cell size. This provides all the information necessary to determine where each given elevation value is located on the Earth.

Once we have read in the header information, the next step is to compute the size of the tiles as a function of the main memory available. The size of a tile is computed such that each tile can be refined while residing entirely in memory. This is done by allowing the user to specify on the command line the amount of memory that is available for the algorithm to use. The smaller the specified memory, the smaller the tiles will be. While it would be possible to partition the terrain using a variety of shapes, we chose to use squares for simplicity. Let the size of a tile be  $\sqrt{r}$  by  $\sqrt{r}$ , with  $r$  to be determined. It can be proved using Euler's formula that the number of triangles in a TIN is two times the number of vertices [19]. Thus at any one time in the refinement process, the most space we will need is  $\sqrt{r} * \sqrt{r} = r$  grid vertices and  $2 * r$  triangles plus some constant amount of space needed to store the tile structure and temporary variables. A simple calculation allows us to compute the tile size  $r$  as a function of the specified memory value and the size of the grids.



**Figure 3.1:** Example of initializing a TIN for the refinement algorithm. The Priority Queue is used to quickly retrieve the point with the largest error from the TIN.

### 3.3 Refining Tiles

After the data has been broken up into tiles we can then use the refinement algorithm to refine each tile individually. The algorithm follows the steps of the generic refinement algorithm presented in Section 2.5; we give details below.

We load a tile into memory and then create the initial TIN structure by adding the four corner points of the tile as vertices to the TIN. From these four points we will have two triangles in our initial structure. Each triangle stores a list of points which lie inside that triangle and a pointer to the vertex with the largest vertical distance from the triangle called the maximum *error*. After we have our two initial triangles we then add each of the non-corner points from the tile into the point list of its surrounding triangle. If a point lies on the edge between two triangles, we just put that point into one of the triangles and handle this special case later. As we build this point list we keep track of the point with the maximal error. When both point lists are created we add the point with the maximum error for each triangle to a maximal priority queue such that the triangle with the largest error is at the root of the PQ tree. (see Figure 3.1).

Now that the initial triangulation is complete, we can begin the refinement loop. This loop extracts the triangle  $s$  with the largest error from the priority queue. It then tests if the point to be added is on one of the edges of the triangle (this is the degenerate collinear case).

1. If the maximum error point is inside the triangle  $s$  then we will add three new triangles in place of  $s$  (refer back to Figure 2.5(a)).
2. If it is collinear, that is, if the maximum error point forms a line with two vertices of the triangle, then we will add four triangles total. Two will be added in place of  $s$  and two will be added in place of the triangle adjacent to  $s$  that shares the collinear edge (refer back to Figure 2.5(b)).

In either case, collinear or not, we need to distribute the point list for  $s$  among its newly created child triangles. In the process of doing this we will not only create the point lists for the new triangles but we will also get the new maximum error for each of the new triangles. These triangles will be added to the priority queue if they have a maximum error greater than the given error  $\epsilon$ , otherwise they are considered done. After we remove the parent triangle  $s$  which has been displaced by its children, we check to see that *each* of the newly created triangles maintains the Delaunay circumcircle property (refer to Figure 2.4). If it does not, we perform an edge swap as in Figure 2.4(c) and recursively check the newly swapped triangles until all are locally Delaunay. After Delaunay is enforced the loop starts again and continues until there are no triangles left in the priority queue. This assures that every triangle in the TIN is locally Delaunay and does not contain any points which have an error greater than  $e$ . The overall algorithm is outlined in Figure 3.2.

**Lemma 1** *Let  $r$  be the size of a tile. Refining a tile takes  $O(\frac{r}{B})$  I/O and  $O(r^2)$  CPU in the worst-case,  $O(r \log r)$  CPU average case, assuming points are evenly*

```

REFINETINTILE( $e, tile$ )
1   $pq \leftarrow$  PQ-INIT()
2   $tt \leftarrow$  INITTINTILE( $tile, pq$ )
3  while  $s \leftarrow$  PQ-EXTRACTMAX( $pq$ ) and  $s \neq$  NIL
4  do if ISCOLLINEAR( $s$ )
5      then FIXCOLLINEAR( $s$ )
6      else  $t1 \leftarrow$  ADDTRI( $s, s.p1, s.p2, s.maxError, tt$ )
7            $t2 \leftarrow$  ADDTRI( $s, s.p1, s.maxError, s.p3, tt$ )
8            $t3 \leftarrow$  ADDTRI( $s, s.maxError, s.p2, s.p3, tt$ )
9           DISTRIBUTEPOINTS( $t1, t2, t3, s, e, pq, tt$ )
10          REMOVETRIANGLE( $s$ )
11          ENFORCEDELAUNAY( $t1, t1.p1, t1.p2, t1.p3, e, tt$ )
12          ENFORCEDELAUNAY( $t2, t1.p1, t1.p3, t1.p2, e, tt$ )
13          ENFORCEDELAUNAY( $t3, t1.p2, t1.p3, t1.p1, e, tt$ )
14  return  $tt$ 

```

**Figure 3.2:** Our Refinement Algorithm.

*distributed.*

**Proof CPU:** We will analyze the CPU complexity of our algorithm in Figure 3.2 step by step. Consider a tile of  $r$  points and let  $k$  be the number of points in the refined tile,  $k \leq r$ . We assume  $r \leq M$  such that each tile fits completely in memory. Step 1 takes constant time to initialize the priority queue. Step 2 takes  $O(1)$  time per tile. The loop in Step 3 is run  $k$  times, once for each point added to the refined TIN. Step 3 takes  $O(\log r)$  time. Steps 5-8 take constant time.

Step 9 could take  $O(k)$  time since in the worst-case a triangle may have to distribute  $k - 1$  points. Thus, the entire algorithm takes  $O(\sum k) = O(r^2)$  worst-case. However, we can prove a much better bound on the average, assuming that the points that are distributed fall evenly among the triangles. Let  $i$  be the total number of points in the refined triangulation. Thus each triangle contains approximately  $\frac{O(r-i)}{\Omega(i)} = O(\frac{r}{i})$  points. When the next vertex is inserted, distributing the points of the triangle takes  $O(\frac{r}{i})$  time. Summed over all points that are inserted this is

$$\sum_{i=1}^n \frac{r}{i} = r \sum_{i=1}^n \frac{1}{i} = \theta(r \log k) = \theta(r \log r)$$

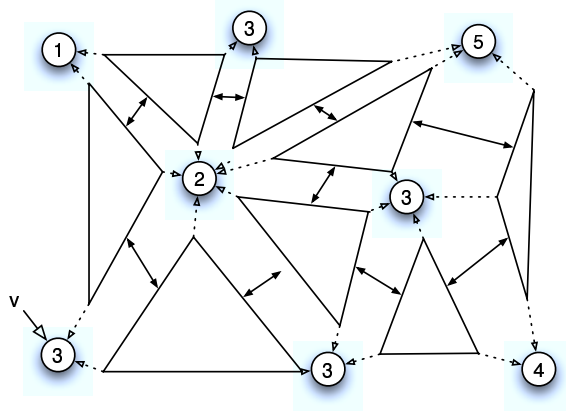
Step 10 takes constant time. For the final Steps 11-13 each call to enforce Delaunay may cause that edge to be flipped, which is done in constant time. In addition, it may cause a cascading set of edge flips. Note that if edge  $e$  across from vertex  $v$  is flipped, the new edge  $r$  is incident to  $v$  (Figure 2.4 (c), edge  $ay$  is incident to  $a$ ). Similarly, all subsequent cascading of the edges flipped are incident to  $v$ . Thus, even though we cannot bound the number of edges per vertex, overall, in the final triangulation, the total number of edges is  $O(r)$ ; thus the total number of edge flips performed by all enforceDelaunay calls is  $O(k)$  (Note: this argument is called backward analysis as in RIC [6]). Since each call to enforceDelaunay also has a hidden call to distribute points we have an additional  $O(r)$  average time per edge flip.

Thus the total runtime for this algorithm is  $O(r \log r)$ .

**I/O:** Since each tile is stored contiguously on disk, it takes  $\frac{r}{B}$  I/Os to read one tile into memory. Since a tile fits entirely in memory no other I/Os are needed and the entire refinement takes  $O(\frac{r}{B})$  I/Os.

### 3.4 TIN Structure in Memory

There are a variety of ways to represent a TIN in memory depending on the application [21]. One could imagine storing a TIN using a linked list of triangles, or even a vector of edges. We describe a method which maintains the topology (adjacency information) of the terrain while using minimal space. That is, for every edge, and triangle in this structure, we will know what the neighbor edges, vertices and triangles are, and where it fits into the overall structure. Using a TIN representation that stores topology explicitly is important for applications like flow modeling where it is necessary to know which triangles are adjacent to a given edge, and is also needed to maintain Delaunay.

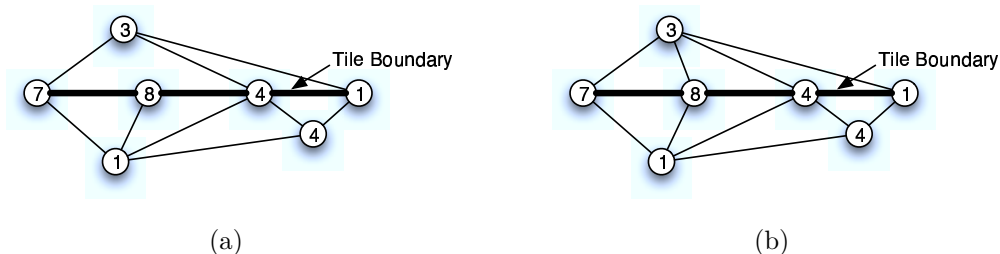


**Figure 3.3:** Storing a TIN: each triangle stores a pointer to its vertices and the neighbor triangles.

Our TIN representation is composed of two basic structures: points and triangles (see Figure 3.3). Points store their  $(x, y, z)$  location in one of two ways. Points in a given triangle  $t$  that are not yet part of the TIN are stored in a list for that triangle. Points which are vertices in the triangulation are removed from triangle lists and are pointed to by triangles. Triangles store three pointers to their three neighboring triangles and three pointers to their three vertices. In addition, in order to have an access point into this network of points and triangles, we store a pointer to the lower left triangle and vertex of the TIN. From this point one can reach any other part of the terrain by traversal of the structure (see Chapter 4).

### 3.5 Combining Tiles

The algorithm in Figure 3.2 describes the refinement process of an individual tile in the terrain without giving details on how to combine tiles. Once the set of tiles has been refined, it is necessary to address how they can be combined to create one contiguous refined terrain.



**Figure 3.4:** Tile Boundaries (a) Tile boundaries are inconsistent with point 8. (b) Tile boundaries are consistent.

### 3.5.1 Maintaining Boundary Consistency

The first issue that must be addressed when combining tiles is maintaining consistency along the boundaries while upholding the invariant that only  $O(1)$  tiles are in memory at one time. In order for two adjacent tiles to be consistent with one another, each point and triangle on one side of the boundary must match up to a point and triangle on the other side. Figure 3.4(a) shows a boundary that is inconsistent since point 8 is in the lower tile and not in its upper neighbor tile. Figure 3.4(b) shows the same boundary with consistent triangles and points.

In our implementation we maintained consistency by first initializing all tiles with their two initial triangles. Then we refine each tile individually. When we add a point to the *right* or *bottom* boundary of a tile we add that point and necessary triangles to the adjacent tile.

Since we start by refining the upper left tile and then continue refining each row of tiles from left to right, we are guaranteed to have a right and bottom tile neighbor that has not yet been refined. It follows that if we maintain consistency with these two edges for every tile, then all tile edges will be consistent. This means that in the worst case we only need less than three tiles to fit in memory at the same time. This is easily handled since we control the size of tiles at the command line.

### 3.5.2 Maintaining Delaunay with Tiles

While tiling is I/O-efficient, it does not guarantee a globally Delaunay triangulation, but only Delaunay within a tile. First, terrains modeled as a TIN need not be Delaunay. A Delaunay triangulation is desirable because of its nice mathematical properties and because as empirically shown in Chapter 4, of better approximations of the terrain with fewer triangles. However there is no absolute requirement that a TIN be Delaunay.

We trade-off the property of global Delaunay with I/O-efficiency. It remains a question whether it is possible to get an I/O-efficient algorithm for refinement which maintains Delaunay globally. Such an algorithm is not known. Moreover, for the simpler problem of finding a Delaunay triangulation of a point set, a  $sort(n)$  I/O-algorithm is known but it is extremely complicated and most-likely impractical.

With our algorithm we could attempt to enforce Delaunay after all of the tiles have been refined. For instance, in a second pass, collect all non-locally Delaunay edges and flip them. However the problem still exists that subsequent cascading flips may not necessarily stay local in a tile and thus may cause access to neighbor tiles which causes a scattered I/O with the same  $O(n)$  bound as the standard refinement algorithm.

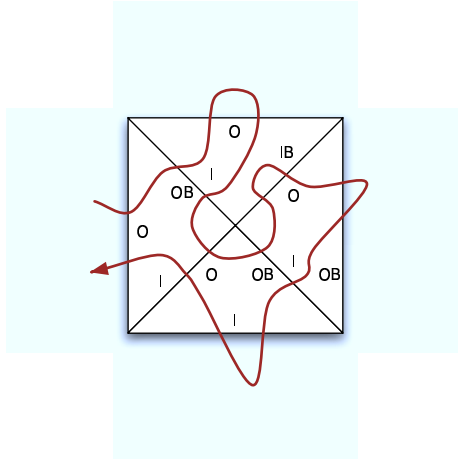
# Chapter 4

## TIN Traversal

After a TIN has been created there are many instances where it is necessary to traverse the TIN efficiently. For example, one may want to visualize the terrain or output it to a file. In either case it is necessary that the traversal only reports a triangle once, otherwise the resulting output will have multiple entries for each triangle. Given this requirement a straightforward approach might be to choose a search algorithm like depth-first search (DFS), and then have a marker byte for each triangle that signifies whether it has already been visited. While this certainly is a viable option, on large datasets markers bits are very expensive and there is a better solution to this problem which does not use them [7, 9, 8]. The idea is to classify the edges of a given triangle in constant time as being *in*, *out*, *in-and-back*, or *out-and-back*. The algorithm starts with the lower leftmost triangle of the TIN and walks from triangle to triangle until coming back to the start triangle, see Figure 4.1. The triangle to be visited next can be determined based on the type of edge being crossed and the type of triangle being entered. With this method of traversal, one can visit all  $n$  triangles in a TIN in  $O(n)$  time and no extra space is required. This method only requires an access point to the lower left most triangle of the TIN.

### 4.1 Defining Edges

In order to choose the next triangle to visit in constant time we must classify the edges of a triangle and choose our next triangle based on the type of edge being crossed. The edges of a triangle are classified with respect to the lower leftmost vertex of the triangulation  $v$ . An edge  $e$  of triangle  $t$  is called:



**Figure 4.1:** Example of TIN traversal path.

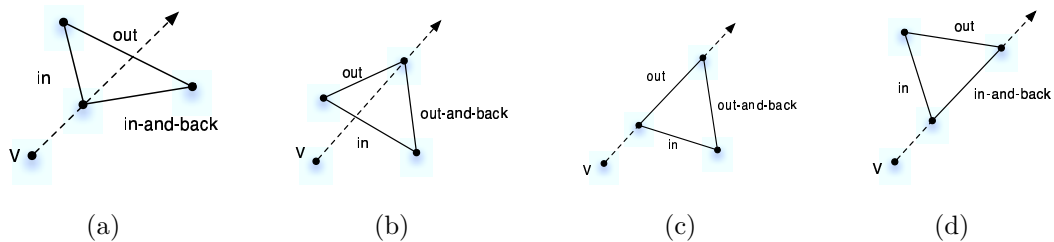
*in* if the line that contains  $e$  separates  $v$  and the third vertex of the triangle.

*out* if  $v$  and the third vertex of the triangle are on the same side of the line containing  $e$ .

With this method there will be two edges of the same type, either  $\{in, in, out\}$  or  $\{out, out, in\}$ . In either case the edge that is further left is the “real” edge and the other is the “back” edge. This means that if there is a triangle with two *in* edges, the one that is further left is the real *in* edge and the other one is called *in-and-back*, see Figure 4.2(a,b).

## 4.2 Degenerate Cases

The above system for classifying edges will work for all triangles that don’t fit two degenerate cases. A degenerate case occurs only when there are two *in* edges. In the first case  $v$  is collinear with just one edge  $e$  of the triangle. That is to say that  $v$  is on the line that contains  $e$ . If  $v$  is collinear with the *in* edge, then that edge becomes an *out* edge, the old *out* edge becomes an *out-and-back* edge, and the *in-and-back*



**Figure 4.2:** Edge classifications. (a) Two in edges, rightmost is in-and-back. (b) Two out edges, rightmost is out-and-back. (c) Collinear with left side. (d) Collinear with right side.

edge becomes an in edge. If  $v$  is collinear with the *in-and-back* edge, then there is no change, refer to Figure 4.2(c,d).

In the other degenerate case,  $v$  is collinear with two edges in the triangle. This can only occur with the lower leftmost triangle which contains the vertex  $v$ . In this case we do the same swapping of labels as above. With these special cases we can classify the entire TIN.

### 4.3 Choosing the Next Triangle

The algorithm starts crossing edge  $e$  which is the lower leftmost edge of the lower leftmost triangle  $t$ . When the algorithm crosses this edge a second time it is done. Assume the traversal reaches triangle  $t$ . The triangle visited next is chosen based on the type of edge  $e$  through which  $t$  was entered and the type of  $t$ . We must decide from six cases since there are two types of triangles and each has three edges.

1. If  $e$  is the only *in* edge of  $t$ , the let  $e'$  be the first *out* edge of  $t$ , and let  $t'$  be the triangle on the other side of  $e$ . Repeat the algorithm with  $t:=t'$  and  $e:=e'$ .
2. If  $e$  is the real *in* edge of  $t$ , the let  $e'$  be the *out* edge of  $t$ , and let  $t'$  be the triangle on the other side of  $e'$ . Repeat the algorithm with  $t:=t'$  and  $e:=e'$ .

3. If  $e$  is an *in-and-back* edge of  $t$ , then let  $t'$  be the triangle on the other side of  $e$ . Repeat the algorithm with  $t:=t'$  and  $e$ .
4. If  $t$  has two *out* edges and  $e$  is the first one, then let  $e'$  be the last *out* edge of  $t$ , and let  $t'$  be the triangle on the other side of it. Repeat the algorithm with  $t:=t'$  and  $e:=e'$ .
5. If  $t$  has two *out* edges and  $e$  is the last one, then let  $e'$  be the *in* edge of  $t$ , and let  $t'$  be the triangle on the other side of it. Repeat the algorithm with  $t:=t'$  and  $e:=e'$ .
6. If  $e$  is the only *out* edge of  $t$ , then let  $e'$  be the real *in* edge and  $t'$  the triangle on the other side of it. Repeat the algorithm with  $t:=t'$  and  $e:=e'$ . [21]

If  $t'$  is set to a triangle that does not exist since it is on the boundary of the terrain, then we will go back into  $t$  through the edge we left and proceed with the algorithm. This algorithm visits every triangle exactly three times, once through each edge. At any point of the traversal one only needs to know the current triangle  $t$ , the edge just crossed  $e$ , and the corner vertex of the TIN  $v$ .

## 4.4 Using The TIN Traversal Algorithm

Since a TIN is stored as a set of triangles and their adjacency information, there is no list of all triangles in the TIN. The only way to visit every triangle is to traverse it using adjacency information.

We have implemented the TIN traversal algorithm described in Sections 4.1 - 4.3 in order to render the TIN in an OpenGL window, to output it to file and to free a TIN from memory. Since the traversal algorithm visits each triangle three times, there are some subtleties that must be addressed in order to ensure efficient and accurate display, output and deallocation.

**Rendering a TIN:** One common use for TIN traversal is rendering the TIN in OpenGL. For fast rendering we need an algorithm that can quickly go through the TIN and draw every triangle once. Since the TIN traversal visits every triangle three times we needed to find a way to only display a triangle once per traversal. If we simply traverse the TIN and only print a triangle when we cross an *in* edge we will accomplish this goal since each triangle has one and only one of these edges.

**Outputting a TIN:** Another common task is to output a TIN to file. We use this both for storing TINs temporarily during tiling and for future computations like flow. Since there does not yet exist a standard format for a TIN file we had to devise one that would be easy and fast to read. We chose to use the TIN traversal algorithm and to output the triangle *every* time it is visited. While this may seem superfluous since each triangle will be written to disk three times, it allows the triangles to be read from the file with adjacency preserved since every adjacent entry in the file is also adjacent in the triangulation. If we output each triangle only once, we lose this adjacency information. To then reconstruct this adjacency information with such a structure would be time intensive since each of the  $n$  triangles would need to find at least one of its neighbors.

**Freeing triangles:** The last use we had for TIN traversal was to free a TIN tile from memory as we output it. We use this functionality when tiling so that we don't have to rely on a limited virtual memory system to store our tiles. A straight-forward approach would be to use TIN traversal to visit every triangle and when a triangle is seen for the third and last time, free it from memory. There is a problem with this however since there is no clear way to determine whether a triangle is being visited for the third time. This is necessary because if we were to free the triangle before its third visit we would surely come back to it at some later time and it would no longer

exist causing a segmentation fault.

While there may exist a more efficient solution using edge classification, we overcame this problem by using a counter for each triangle. Adding such a counter would typically be undesirable since it would add substantial space requirement per triangle. However since we only need the counter when we are deleting triangles our existing implementation already has free space to store a counter without having to add more space requirements per triangle. While we were able to circumvent this issue, it does seem like there is the potential for a more elegant solution using edge classification which we leave for future work.

# Chapter 5

## Experimental Results

In this chapter we present empirical results to: (1) investigate the scalability of our I/O-efficient refinement and (2) investigate the space efficiency of TINs versus grids. We then highlight the differences between using TINs instead of grids.

### 5.1 Datasets and Test Platform

In order to investigate the performance of our refinement algorithm, we performed experiments with real-life terrains of varying sizes and characteristics. Table 5.1 summarizes the characteristics of the various datasets we used. Nodata indicates the percentage of points in the terrain for which no elevation was recorded.

We ran all tests on Apple’s dual processor G5 with 2.5GHZ CPU and 1 GB of main memory. During the test no more than 100% of one CPU was used since we do not currently use threads to run refinement in parallel.

Dataset	Resolution	Grid Size	Raw Size	Nodata
Kaweah	1163 x 1424	$1.6 \cdot 10^6$	3.2MB	42%
Puerto Rico	4452 x 1378	$5.9 \cdot 10^6$	11.8MB	81%
Sierra Nevada	3750 x 2672	$9.5 \cdot 10^6$	19.0MB	4%
Hawaii	6784 x 4369	$28.2 \cdot 10^6$	56.4MB	93%
Cumberlands	8704 x 7673	$66.7 \cdot 10^6$	133.4MB	73%
Lower New England	9148 x 8509	$77.8 \cdot 10^6$	155.6MB	64%
Central Appalachians	12042 x 10136	$122.0 \cdot 10^6$	244.0MB	5%

**Table 5.1:** Characteristics of terrain datasets.

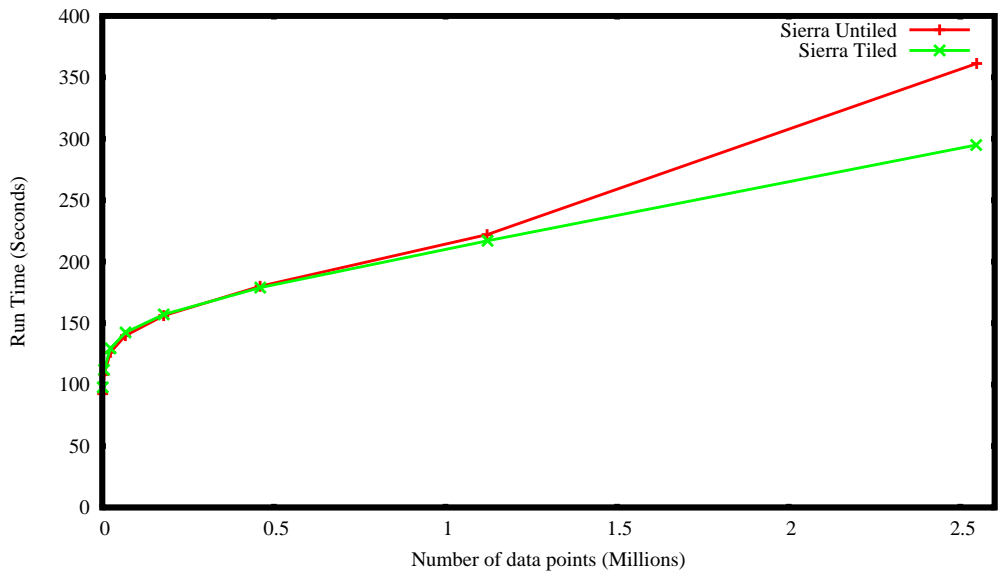
## 5.2 Effects of Tiling

Intuitively for small datasets which fit in memory, one would expect that tiling would actually take longer since there is the overhead of dividing the terrain into tile structures and operating on them one at a time. At the point where the terrain is too large to fit in memory however we expect to see a vast improvement in run time when using tiles.

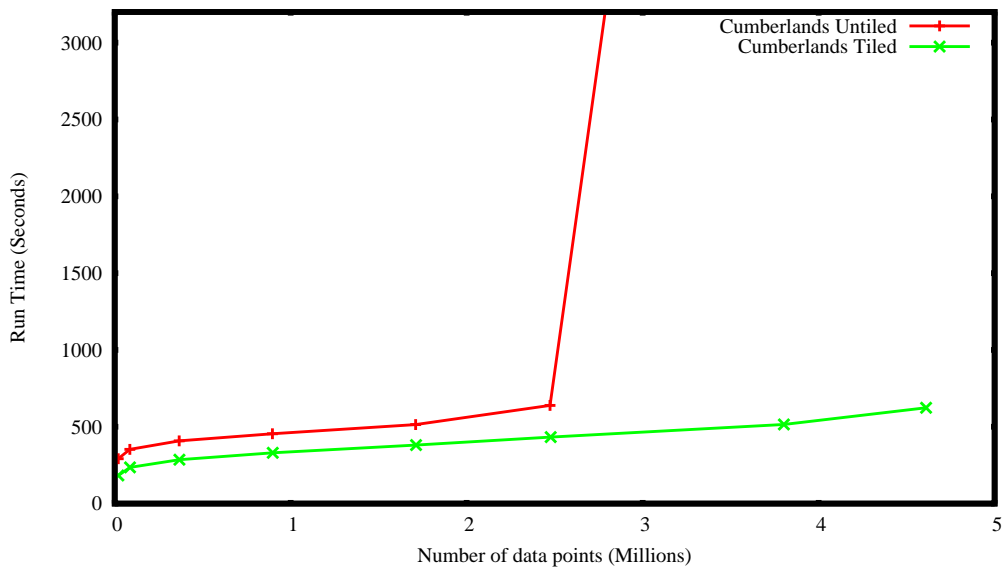
We set out to find this point by testing all the datasets listed in table 5.1 with error levels of  $\{10,5,2,1,0.5,0.25,0\}$  as a percentage of the maximum elevation in the terrain. Since error levels inversely relate to the number points added to a triangulation, one would expect that lower error levels would increase the number of points being added to the triangulation. Thus we would expect tiling to be more efficient at lower error levels where all points cannot fit in memory. Furthermore since the CPU and I/O time is based on the number of points being added, we would expect that tiling would become more efficient for the same number of points on any terrain which does not fit fully in memory.

Our tests for Kaweah, Puerto Rico, and Hawaii showed that both our tiled and untiled algorithm had the same runtime since all three datasets could fit completely in memory and even our tiled algorithm only used one tile. From the run-times for Sierra in Figure 5.1 we see that initially the untiled version takes less time than the tiled version as expected. From about 500,000 points on we see that tiling is marginally more efficient and while we don't see a exponential increase in runtime, we begin to see a noticeable difference around 2.5 million points.

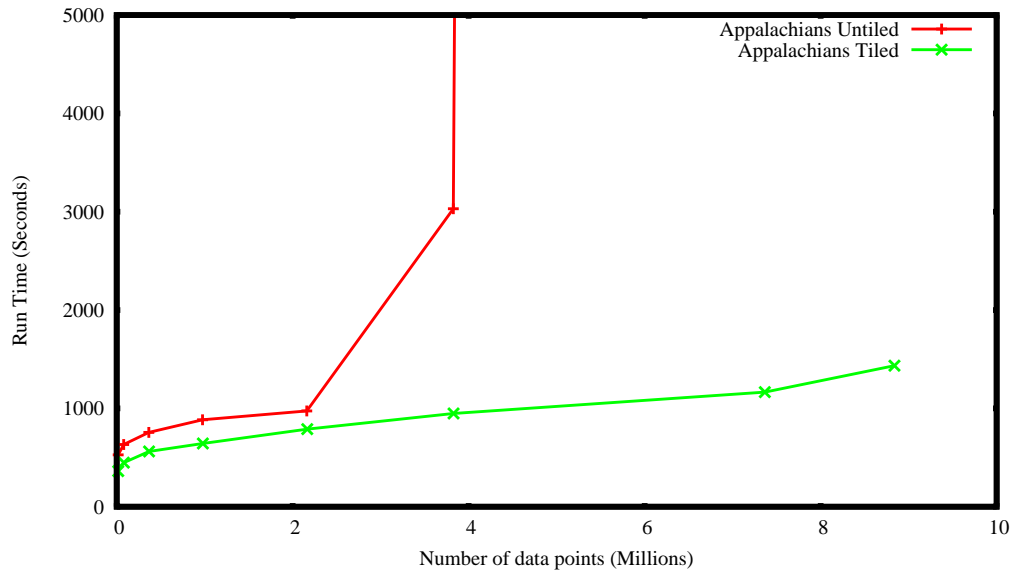
The run-times for Cumberlands in Figure 5.2 reveal that for this larger dataset, the tiled algorithm is always more efficient than the untiled version. We also see the expected jump in runtime for the untiled version around 2.5 million points which fits with our test on Sierra. Since the scaling on the graph cuts off the run-times for



**Figure 5.1:** Tile vs Untiled Comparison On Sierra



**Figure 5.2:** Tile vs Untiled Comparison On Cumberland

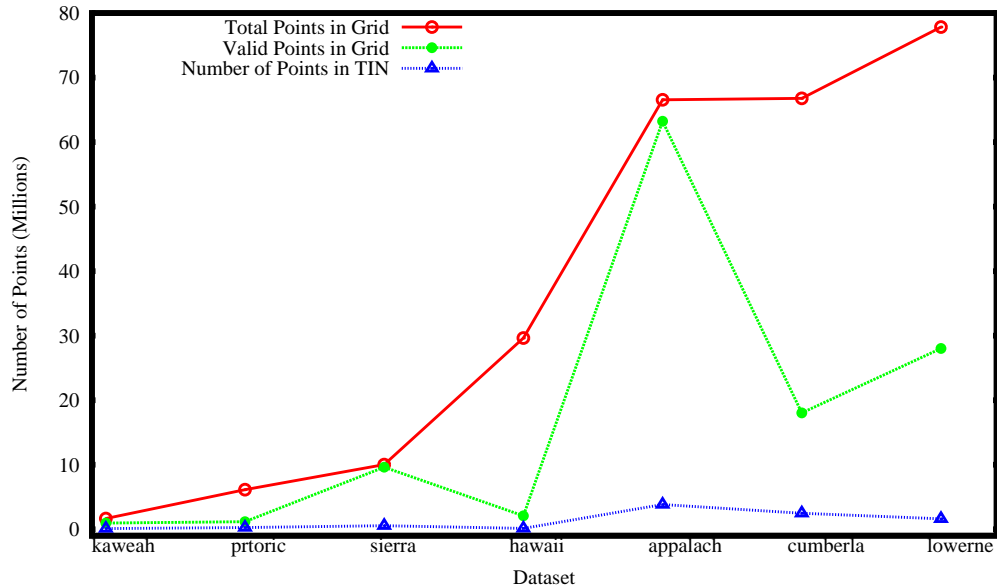


**Figure 5.3:** Tile vs Untiled Comparison On Appalachians

the untiled version, we can better illustrate the vast improvement with tiling with the test at error 0 containing 4.5 million points where our tiled algorithm takes 10 minutes while the untiled version takes approximately 13 hours.

The run-times for the Appalachians dataset shown in Figure 5.3 follow the similar trend with Cumberlands. As with our previous tests, the run-times are very similar until we reach about 2.5 million points where the untiled runtime takes off. When refining approximately 9 million points at error 0 on this dataset, the tiled version ran in 23 minutes while the untiled version ran for over 7 days without ever finishing. That means that for 9 million elements, the tiled version is at least 438 times faster than the untiled version.

From these results we found that there was a significant improvement of the standard refinement algorithm when we use tiling on large datasets. In fact for Cumberlands and Appalachians we found that tiling is always faster. This does not fit expectations for the tests with fewer points where we would predict untiled to be faster. However, we attribute this to the fact that our tiled version has less



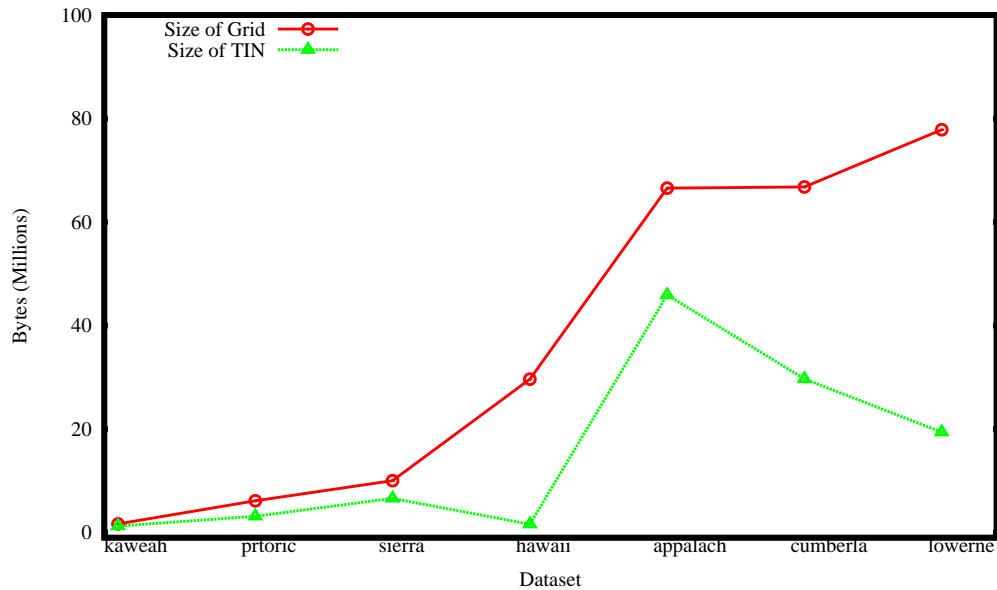
**Figure 5.4:** Grid vs. TIN Point Comparison (0.1% Error)

CPU overhead on boundaries since it does not maintain Delaunay across boundaries. Overall our results met expectations and proved that tiling becomes much more efficient when over 2.5 million points are added to the triangulation.

### 5.3 Grid vs. TIN Comparison

Another claim we made earlier is that a TIN can represent the same terrain as a grid by storing fewer points. Since the relative efficiency of the two methods will depends greatly on the terrain it was not clear in practice how much smaller a TIN would be. If a terrain has many flat areas, then a grid will still have to store every point on the flat area where a TIN can ignore all of them. Datasets of islands like *hawaii* and *prtórico* are particularly better suited for TINs since they have vast expanses of flat areas due to the ocean being recorded as flat no data values.

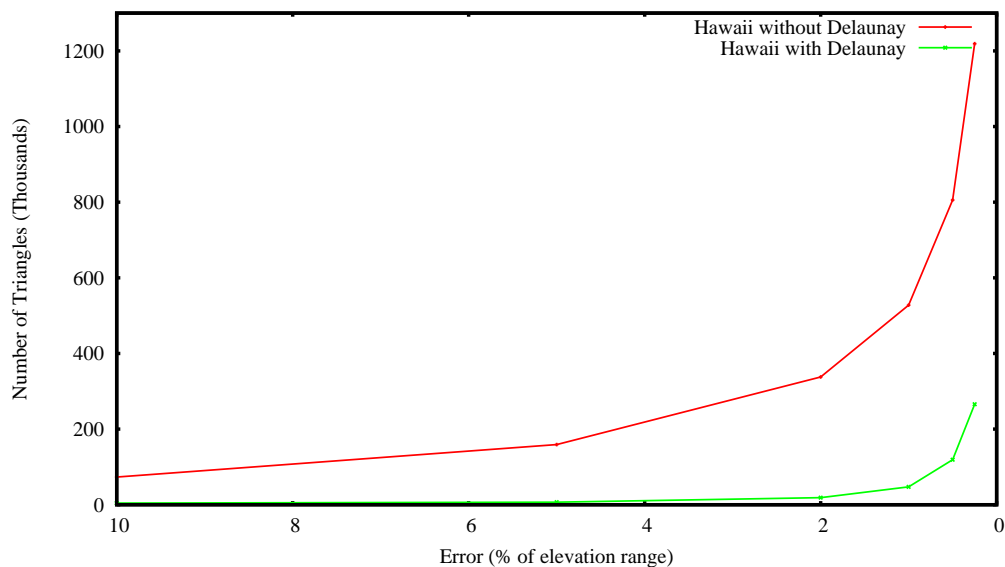
Figure 5.4 shows the number of points needed to store each dataset as both a grid and a TIN. We chose to compare the grid to a TIN with error 0.1% because



**Figure 5.5:** Grid vs. TIN Space Efficiency (0.1% Error)

it is just large enough to remove flat areas in the terrain. If we used error 0% the number of points would be equal to the number of valid points in the grid since every point that does not have a no data value would be included in the triangulation. As we can see from the graph, there is a large difference between the total points in a grid and number of points stored by a TIN for the same terrain. We also include the number of valid points in the grid to illustrate how many points in the grid are no data. For example, on the Hawaii dataset, there are relatively few valid points since there are so many no data values, thus the efficiency of the TIN is mostly due to no data points. On the Appalachians dataset, we see that almost all the data points are valid. Even still, the TIN requires far fewer points due to the flat areas of valid points.

While it is apparent that TINs require substantially fewer points than grids, it is not clear whether or not they are actually more space efficient than grids since TINs store much more information about triangles and adjacency. Figure 5.5 shows the actual space difference between the two representations in bytes. The graph indicates



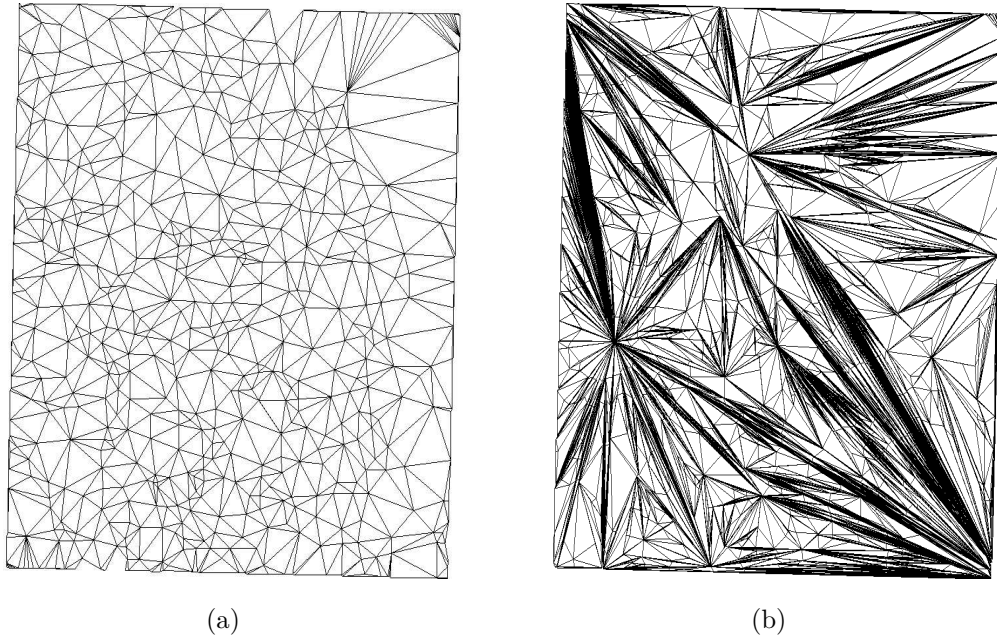
**Figure 5.6:** Effect of Maintaining Delaunay on Triangles

that TINs are in fact more space efficient representation for every dataset we tested. Intuitively, TINs are more space efficient on larger datasets with more flat areas or no data values like Hawaii and Lower New England.

From these experiments we find our claim that TINs are more space efficient than grids to be true. Furthermore, the TINs created in this experiment only removed flat areas, if one were to refine a grid into a TIN with a higher error, one would see an even greater efficiency in space.

## 5.4 Effects of Maintaining Delaunay

While Delaunay triangles are desirable for many other reasons mentioned above, the full benefit of maintaining this property was not clear before experimentation. As maintaining Delaunay adds some constant amount of processing time to check and swap triangles for each triangle added, we expected that Delaunay would increase our runtime by a constant factor. We found that on average Delaunay adds a constant



**Figure 5.7:** TINs on Set1 with same error. (a) Delaunay is maintained (b) Delaunay is ignored.

factor of 1.6 to the total runtime ( $O(1.6n) = O(n)$ ) which has a nominal impact on the runtime.

We tested various datasets with and without maintaining the Delaunay property throughout refinement and compared the space efficiency to see if the added CPU time has an effect on space. To do this we counted the number of triangles produced. Figure 5.6 shows that as the error decreases and number of data-points increase we see that there are far fewer triangles created when the Delaunay property is maintained. A more concrete example of this is shown in Figure 5.7 where we can see that for the same error on the same dataset, Delaunay produces far fewer triangles. Therefore the added CPU overhead in generating a Delaunay triangulation will greatly decrease the number of triangles making other operations like outputting or flow computation less time intensive.

# Chapter 6

## Conclusions and Future Work

In this chapter we end with a few concluding remarks and highlight areas for future work.

### 6.1 Conclusion

Currently our implementation of I/O-efficient refinement from grids to TINs runs in both the GRASS environment and as a stand-alone application. This project provides a starting point for further implementation and experimentation of algorithms on TINs in the GIS open source community.

Our experiments have shown that tiled refinement is much faster on large datasets than the standard refinement algorithm. For example we found that for the same given error level on the same terrain our tiled implementation took 16 minutes for refinement while the standard version took over 7 days and never finished. We also found that for large datasets, the TIN representation was significantly more efficient than a grid. Moreover, we found that Delaunay triangulations not only produced more equally sized triangles but it produced fewer triangles in the refinement process resulting in significantly less required space to store the triangulation.

The GIS community relies on software that many times does not scale. As the size of datasets increase and the models for this data become more complex, we will see the need for theoretically sound implementations which can handle large data. This project makes a step in that direction by implementing an I/O-efficient representation for terrains which has proven to be both theoretically and practically efficient.

## 6.2 LIDAR to TIN

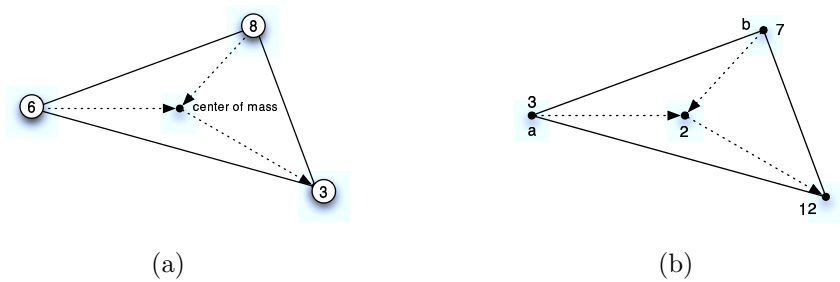
As LIDAR data is becoming more popular, we believe that the refinement of LIDAR to TIN will be useful functionality which currently does not exist in the GIS community. Furthermore, experiments are likely to show that LIDAR data can more effectively be represented by TINs which means that this may become the best representation for LIDAR data.

To do this one would need to generalize our refinement algorithm described in Section 2.5 to handle sample points instead of grids. The algorithm is similar in that it tiles the points and refines each tile one at a time. The only real difference is in how the tiles are created since sample point data is not evenly distributed over space.

To address the sporadic arrangement of points and to guarantee that each tile is no larger than the size of the memory, one will need to use a quad tree or grid file to partition the data into tiles. The overall details of the algorithm stay the same; except maintaining consistency of tiles which becomes more complicated.

## 6.3 Flow Modeling on TINs

After adding support for LIDAR data, it will be interesting to investigate how TINs model information about a terrain other than elevation. One of the most widely used models, as mentioned in Section 2.6, is water flow modeling. While we have mentioned two other methods for modeling flow on TINs, we propose two similar discrete models for flow on TINs. As many of the details of these approaches will need further exploration, we will describe them at a high level and leave them as future work.



**Figure 6.1:** Triangle based flow model. (a) Flow direction computed through center of mass. Circled numbers are elevations. (b) Flow accumulation computed for point  $c$ .  $c$  receives  $3 + 7 + 2 = 12$  units of water.

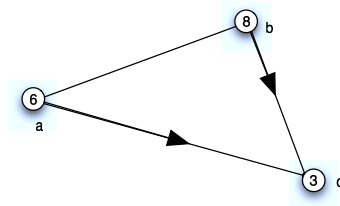
### 6.3.1 Triangle-based Flow

The first model is triangle based. We introduce a vertex per triangle and connect it to the 3 vertices of the triangle. This defines a graph with one vertex and three edges per triangle; call it the *flow direction graph*. For each edge in this graph we orient it from the high to the low end point, see Figure 6.1(a). We assume the vertex represents the triangle’s center of gravity and we find its height by interpolation.

Let  $n$  be the number of points in the TIN, the number of triangles is  $t = 3n - 6$  [19]. The flow direction graph has  $3n$  vertices and  $9n$  edges. Intuitively, this graph is acyclic since all vertices flow from high points to low points. Each vertex has a flow value equal to the area of the triangle it represents. Every vertex sends its own flow and its incoming flow using its value in the flow direction graph, see Figure 6.1(b).

### 6.3.2 Vertex-based Flow

Our second model is vertex-based and routes flow through TIN edges. It does not introduce new vertices or edges. For each triangle find the lowest point  $c$  and assign flow for the other two vertices to  $c$ , see Figure 6.2(a). Thus the flow direction graph is a subgraph of the TIN, and has  $O(n)$  vertices and  $O(4n) = O(n)$  edges. As above,



**Figure 6.2:** Vertex based flow model. Flow directions are assigned to edges.

the flow direction graph will be acyclic since all water flows down. To handle flow accumulation we assign a weight (flow quantity) to edges in the flow direction graph. Each edge carries a flow equal to the area of the part of the triangle that routes flow through that edge. Flow accumulation of a vertex is thus the sum of all the weights of all edges that have a path to that vertex.

The advantage of this second approach to flow modeling is that the flow direction graph has a smaller size. The disadvantage is that since the flow is not at the triangle level the flow model may not be as accurate (i.e. flow can zig-zag on slopes).

## Bibliography

- [1] NASA Earth Observing System (EOS) project. <http://eos.nasa.gov/>.
- [2] U.S. Geological Survey (USGS) digital elevation models. [http://mcmcweb.er.usgs.gov/status/dem\\_stat.html](http://mcmcweb.er.usgs.gov/status/dem_stat.html).
- [3] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
- [4] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.
- [5] L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. S. Vitter, and R. Wickremesinghe. Flow computation on massive grid terrains. *GeoInformatica*, 2003. Earlier version appeared in *Proc. 10<sup>th</sup> ACM International Symposium on Advances in Geographic Information Systems (ACM-GIS'01)*.
- [6] H. Edelsbrunner. Triangulations and meshes in computational geometry. *Acta Numerica*, pages 133–213, 2000.
- [7] C. Gold and S. Cormack. Spatially ordered networks and topographic reconstructions. In *Proc. 2nd Int. Sympos. Spatial Data Handling*, pages 74–85, 1986.
- [8] C. M. Gold, T. D. Charters, and J. Ramsden. Automated contour mapping using triangular element data structures and an interpolant over each irregular triangular domain. *SIGGRAPH Comput. Graph.*, 11(2):170–175, 1977.
- [9] C. M. Gold and U. Maydell. Triangulation and spatial ordering in computer cartography. In *Proc. Canad. Cartographic Association Annual Meeting*, pages 69–81, 1978.

- [10] GRASS Development Team. GRASS GIS homepage. <http://grass.itc.it/>.
- [11] M. Heller. Triangulation algorithms for adaptive terrain modeling. In *Proc. 4th Intl. Symp. on Spatial Data Handling*, volume 1, pages 163–174, Zürich, 1990.
- [12] S. Jenson and J. Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing*, 54(11):1593–1600, 1988.
- [13] J. Lee. A drop heuristic conversion method for extracting irregular network for digital elevation models. In *GIS/LIS '89 Proc.*, volume 1, pages 30–39. American Congress on Surveying and Mapping, Nov. 1989.
- [14] J. Lee. Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models. *Intl. J. of Geographical Information Systems*, 5(3):267–285, July-Sept. 1991.
- [15] M. McAllister. A watershed algorithm for triangulated terrains. pages 103–106.
- [16] M. McAllister and J. Snoeyink. Extracting consistent watersheds from digital river and elevation data, 1999.
- [17] I. D. Moore, R. B. Grayson, and A. R. Ladson. Digital terrain modeling: a review of hydrological, geomorphological and biological applications. *Hydrological Processes*, 5:3–30, 1991.
- [18] NASA Jet Propulsion Laboratory. NASA Shuttle Radar Topography Mission (SRTM). <http://www.jpl.nasa.gov/srtm/>.
- [19] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, second edition edition, 1998.

- [20] G. Tucker, S. Lancaster, N. Gasparini, and S. Rybarczyk. An object-oriented framework for hydrology and geomorphic modeling using triangulated irregular networks. *Computers and Geosciences*, 27(8):959–973, 2001.
- [21] M. van Kreveld. Digital elevation models and tin algorithms. In M. van Kreveld, J. Nievergelt, T. Roos, and P. W. (Eds.), editors, *Algorithmic Foundations of Geographic Information Systems*, LNCS 1340, chapter 3, pages 37–78. Springer-Verlag, Berlin, 1997.
- [22] M. van Kreveld, J. Nievergelt, T. Roos, and P. W. (Eds.). *Algorithmic Foundations of GIS*. LNCS 1340. Springer-Verlag, 1997.
- [23] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. 33(2):209–271, 2001.
- [24] S. Yu, M. van Kreveld, and J. Snoeyink. Drainage queries on TINs: from local to global and back again. In *Proc. 7th Int. Symp. on Spatial Data Handling*, pages 13A.1–13A.14, 1996.