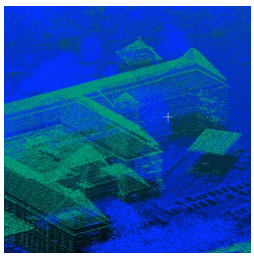
Conversion between terrain representations

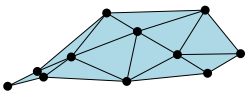
Lesson 08*

We consider in this lesson the following four terrain representations and discuss the conversions between them:



point cloud (PC)

raster



TIN

isolines

from/to	PC	raster	TIN	isolines
PC	—	interpolate at middle points of cells (§1)	create DT using 2D projection of points (ie using x and y only)	convert to TIN + extract from triangles (§2.2) + structure output (§2.3)
raster	keep middle points only	—	create TIN using middle points of cells + TIN simplification (§3)	extract from grid cells (§2.1) + structure output (§2.3)
TIN	keep only vertices	interpolate at middle points of cells (§1)	—	extract from triangles (§2.2) + structure output (§2.3)
isolines	keep only vertices — *warning: 'wedding cake' effect* (§4)	convert lines to points + interpolate (§1) — *warning: 'wedding cake' effect* (§4)	create DT using points — *warning: 'wedding cake' effect* (§4)	—

1 Conversion of PC/TIN to raster

As shown in Figure 1, this step is trivial: one needs to interpolate at the locations of the centre points of the raster cells. The interpolation method can be any of the ones described in Lessons 04 and 05.

2 Conversion to isolines

Reading a contour map requires some skill, however it is considerably easier to learn to interpret a contour map than to manually draw one from a limited set of sample points. Yet this was exactly the task of many cartographers in the past couple of centuries: it was intuitively done by imagining a local triangulation of sample points.

Isolines are usually directly extracted from either a TIN or a grid representation of a terrain. The basic idea, as shown in Figure 2, is to compute the intersection between a level value (eg 200m) and each cell of the terrain (triangle or grid cell in our case). Notice that the cells are

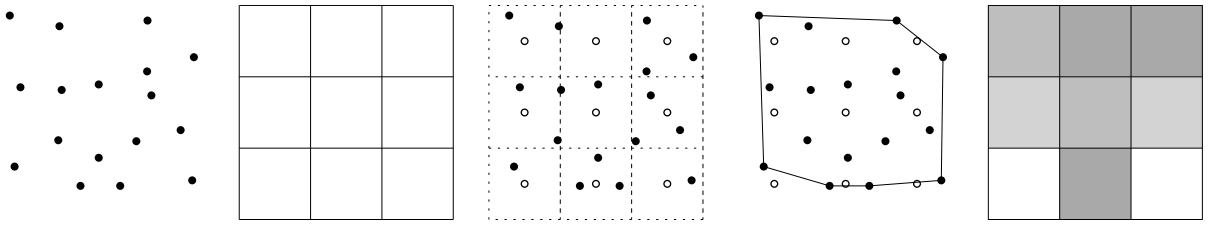


Figure 1: **(a)** input sample points. **(b)** size/location of output raster. **(c)** 9 interpolations must be performed (at locations marked with ○): at the middle of each cell. **(d)** the convex hull of the sample points show that 2 estimations are outside, thus no interpolation. **(e)** the resulting raster.

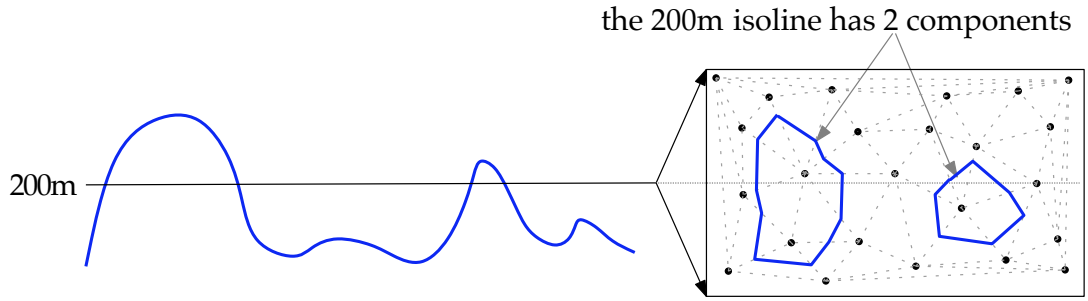


Figure 2: Vertical cross-section of a terrain (left), and a 2D projection of the terrain TIN with the extracted 200m isoline (right).

Algorithm 1: Simple extraction of one isoline

Input: a planar partition E formed of cells (either rectangular or triangular cells); the elevation value z_0

Output: a list of unstructured line segments representing the contour lines at z_0

```
1 segmentList ← [ ] ;
2 for e ∈ E do
3     if z0 intersects e then
          /* See Figures 3 and ??                    */
4         extract intersection χ of z0 with e;
5         add χ to segmentList;
```

'lifted' to their elevation. Each cell of the terrain is thus visited, one after the other, and for each cell if there is an intersection (which forms a line segment) then it is extracted. The resulting set of segment lines forms an approximation of the isoline. This process is then repeated for every level value. Notice that an isoline can have several *components*, for instance when the terrain has more than one peak.

Therefore the number and size of the line segments in the resulting isoline are dependent on the resolution of the data representation.

The basic algorithm for extracting one isoline is shown in Algorithm 1. Note that since the algorithm contours every grid cell or triangle individually and requires only local information, it is very easy to parallelise. It is thus a scalable algorithm. The same idea can be used to extract all the isolines: for each triangle/cell and each level value, extract all the necessary line segments.

2.1 Conversion of raster to isolines

Intersections are computed by linearly interpolating the elevations of the vertex pairs along the edges of this grid. Figure 3 illustrates the different possible configurations. The top-left case indicates the case for which there are no intersections: all vertices are either higher or lower than z_0.

Observe that two vertices are exactly at z_0, then the extraction of these is in theory not necessary because the neighbouring cell could also extract them. However, we do not want to obtain an output with duplicate line segments, and thus a simple solution to this is to only extract such line segments if they are for instance the lower and/or left segments of a given cell.

The most interesting case is the bottom-left one in Figure 3, it occurs when the two pairs

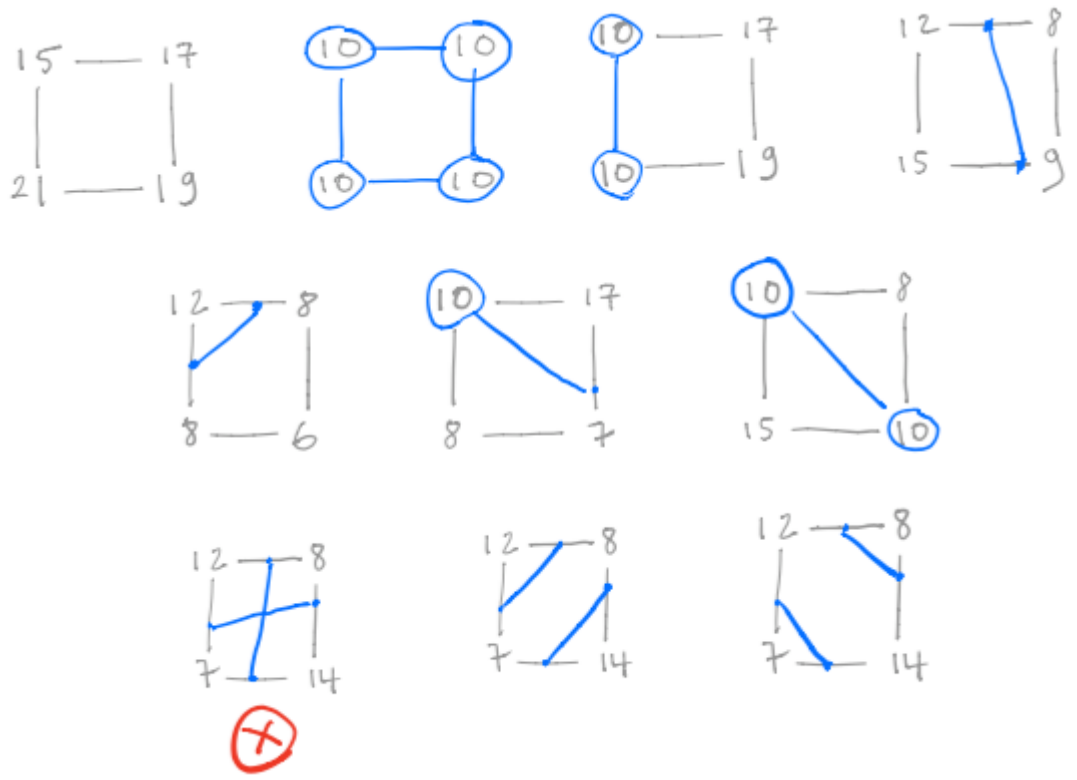


Figure 3: Different cases when extracting an isoline at elevation 10m for a regular grid. The grey values are the elevation of the vertices forming one regular cell, and the blue lines and vertices are the ones extracted for that cell.

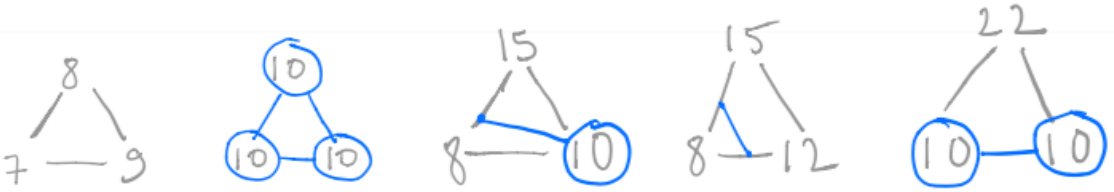


Figure 4: Different cases when extracting an isoline at elevation 10m for a TIN. The grey values are the elevation of the vertices forming one triangle, and the blue lines and vertices are the ones extracted for that triangle.

of opposing points are respectively higher and lower than z_0. This forms a saddle point. The ambiguity arises here since there are two ways to extract a valid pair of contour line segments (only one of the 2 options must be extracted). This can be resolved by simply picking a random option or consistently choose one geometric orientation.

2.2 Conversion TIN to isolines

Since a triangle has one fewer vertex/edge than a square grid cell, there are less possible intersection cases (Figure 4) and, more importantly, there is no ambiguous case. The worst case to handle is when there are horizontal triangles at exactly the height of the isoline. Otherwise, the intersection cases are quite similar to the raster situation and they can be easily implemented.

To avoid extracting twice the same line segment when 2 vertices are at z_0 (case on the right in Figure 4), then we can simply look at the normal of the edge segment: if its y-component is positive then it can be added, if $y = 0$ then only add if the x-component is positive.

Observe that since the algorithm is simpler than that for a raster dataset, one way to extract isolines from a raster dataset is by first triangulating it: each square cell is subdivided into 2 triangles (simply ensure that the diagonal is consistent, eg from lower-left to top-right).

2.3 Structuring the output

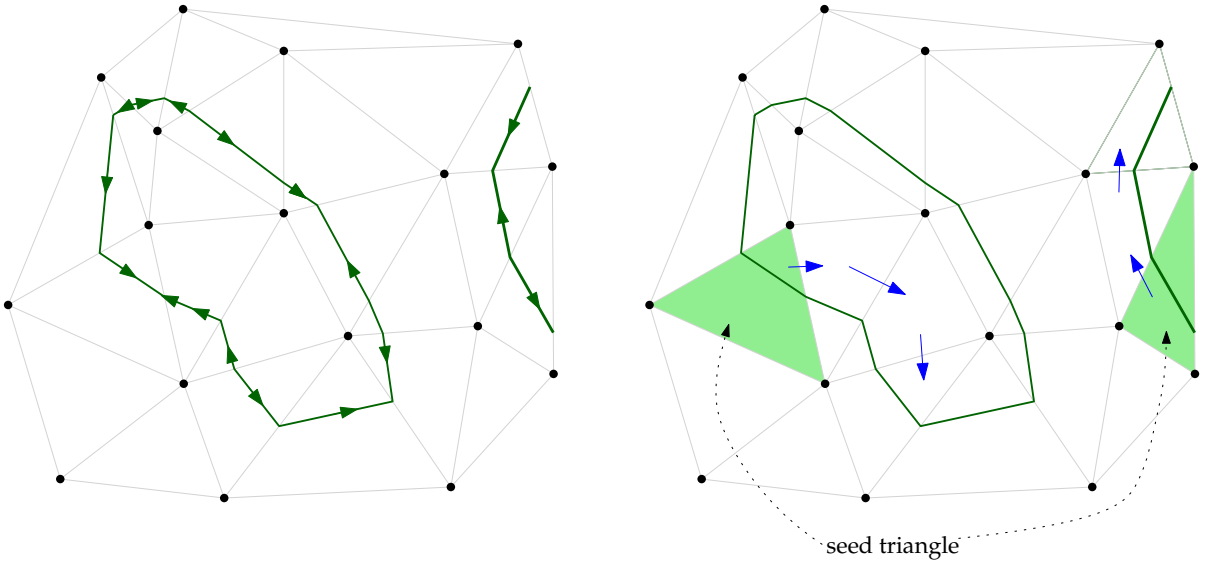
The line segments obtained from the simple algorithms above are not structured, ie they are in an arbitrary order (the order in which we visited the triangles/cells) and are not connected. Furthermore, the set of line segments can form more than one *component*, a set of segments forming a closed polygon (unless they are at the border of the dataset). Perhaps the only application where having unstructured line segments is fine is for visualisation of the lines. For most other applications this can be problematic, for instance:

1. if one wants to know how many peaks above 1000m there are in a given area;
2. if smoothing of the isolines is necessary, with the Douglas-Peucker algorithm for instance;
3. if a GIS format requires that the isolines be closed polylines oriented such that the higher terrain is on the left for instance, such as for colouring the area enclosed by an isoline.

To obtain structured segments, the simplest solution is to merge, as post-processing, the line segments based on their start and end vertices. Observe that the line segments will not be consistently oriented to form one polygon (see Figure 5a), that is the orientation of the segments might need to be swapped. This can be done by simply starting with a segment ab, and searching for the other segment having b as either start or end vertex, and continue until a component is formed (a polygon is formed), or until no segment can be found (the border of the dataset is reached, as shown in Figure 5a).

As shown in Figure 5b, another solution is to find *one* cell τ_0 intersecting the isoline at a given elevation, 'tracing' the isoline by navigating from τ_0 to the adjacent cell, and continuing until τ_0 is visited again (or the border of the dataset is reached). To navigate to the adjacent cell, it suffices to identify the edge ϵ intersecting the isoline, and then navigating to the triangle/cell that is incident to ϵ. It is possible that there is no adjacent cell, if the boundary of the convex hull is reached in a TIN for instance. This requires that the TIN be stored in a topological data structure in which the adjacency between the triangles is available (for a grid this is implied).

The main issue is finding the starting cells (let us call them seed triangles). Obviously, it suffices to have one seed for each of the component of the isolines (there would be 2 seeds in Figure 5b). An easy algorithm to extract all the components of an isoline requires visiting all



(a) (b)

Figure 5: **(a)** The isoline segments extracted with Algorithm 1 do not have a consistent orientation. **(b)** Algorithm 1 can be sped up by starting at a seed triangle and 'tracing' the isoline; the order is shown by the blue arrows.

the cells in a terrain, and keeping track of which triangles have been visited (simply store an Boolean attribute for each triangle, which is called a *mark bit*). Simply visit triangle sequentially and mark them as 'visited', when one triangle has an intersection then start the tracing operation, marking triangles as visited as you trace.

2.4 Smoothness of the contours

The mathematical concept of the *Implicit Function Theorem* states that a contour line extracted from a field f will be no less smooth than f itself. In other words, obtaining smooth contour lines can be achieved by smoothing the field itself. Robin Sibson[1] goes further in stating that:

> *'The eye is very good at detecting gaps and corners, but very bad at detecting discontinuities in derivatives higher than the first. For contour lines to be accepted by the eye as a description of a function however smooth, they need to have continuously turning tangents, but higher order continuity of the supposed contours is not needed for them to be visually convincing.'*

In brief, in practice we should use interpolant functions whose first derivative is continuous (ie C^1) if we want to obtain smooth contours. C^0 interpolants are not enough, and C^2 ones are not necessary.

3 Simplification of a TIN

The TIN simplification problem is:

> Given a TIN formed by the Delaunay triangulation of a set S of points, the aim is to find a subset R of S which will approximate the surface of the TIN as accurately as possible, using as few points as possible. The subset R will contain the 'important'

[1] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.63

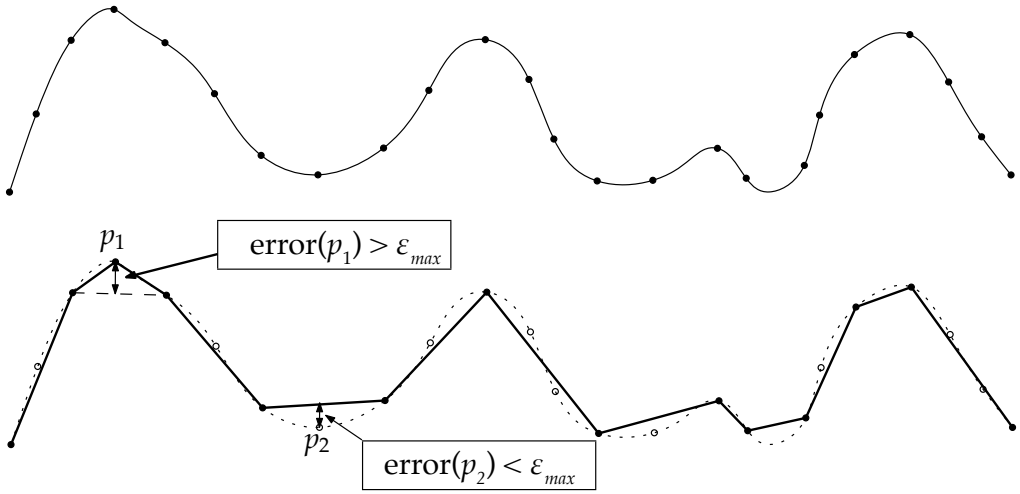


Figure 6: The importance measure of a point can be expressed by its vertical error. When this error is greater than a given threshold ϵ_{max}, the point is kept (p_1), else it is discarded (p_2).

points of S, ie a point p is important when the elevation at location p can not be accurately estimated by using the neighbours of p.

The overarching goal of TIN simplification is always to (smartly) reduce the number of points in the TIN. This reduces memory and storage requirements, and speeds up TIN analysis algorithms.

Observe that the simplification of a TIN can be used to simplify a raster terrain: we can first obtain the triangulation of the middle points of each cell, and then simplify this TIN to obtain a simplified terrain.

3.1 The importance of a point

The importance of a point is a measure that indicates the error in the TIN when that point would not be part of it. Imagine for instance a large flat area in a terrain. This area can be accurately approximated with only a few large triangles, and inserting points in the middle of such an area does not make the TIN more accurate. An area with a lot of relief on the other hand can only be accurately modelled with many small triangles. We can therefore say that the points in the middle of the flat area are less important than the points in the area with relief.

The importance of a point—or importance measure—can be expressed in several ways, eg based on an elevation difference or the curvature of the point. Here we focus on the *vertical error* which has proven to be effective in practice.

The vertical error of a point p is the elevation difference between p itself and the interpolated elevation in the TIN $\mathcal{T}$ at the (x, y) coordinates of p (see Figure 6). Notice that $\mathcal{T}$ does not contain p as a vertex.

3.2 TIN simplification algorithms

There are two main approaches to TIN simplification: decimation and refinement. In a decimation algorithm, we start with a TIN that contains all the input points, and gradually remove points that are not important. In a refinement algorithm, we do the opposite: we start with a very simple TIN, and we gradually refine it by adding the important points.

Algorithm 2: TIN simplification by refinement

Input: A set of input points S, and the simplification threshold ϵ_{max}
Output: A triangulation $\mathcal{T}$ that consists of a subset of S and that satisfies ϵ_{max}

```
1  Construct an initial triangulation T that covers the 2D bounding box of S ;
2  ε ← ∞ ;
3  while ε > ε_max do
4      ε ← 0 ;
5      q ← nil ;
6      for all p ∈ S do
7          τ ← the triangle in T that contains p ;
8          ε_τ ← the vertical error of p with respect to τ ;
9          if ε_τ > ε then
10             ε ← ε_τ ;
11             q ← p ;
       /* insert the point q that has the largest error:                    */
12     insert into T the point q ;
13     remove q from S ;
```

3.2.1 TIN simplification by refinement

Here we describe an iterative refinement algorithm based on greedy insertion[2]. It begins with a simple triangulation of the spatial extent and, at each iteration, finds the input point with highest importance—the highest vertical error—in the current TIN and inserts it as a new vertex in the triangulation. The algorithm stops when the highest error of the remaining input points with respect to the current TIN is below a user-defined threshold ϵ_{max}. Algorithm 2 shows the pseudo-code. It is also possible to insert only a certain percentage of the number of input points, eg we might want to keep only 10% of them.

3.2.2 TIN simplification by decimation

The implementation of the decimation algorithm is similar to the refinement algorithm. The main differences are

1. we start with a full triangulation of all the input points, instead of an empty triangulation;
2. instead of iteratively adding the point with the highest importance, we iteratively remove the point with the lowest importance, and
3. in order to compute the importance of a point we actually need to remove it *temporarily* from the triangulation before we can decide if it should be permanently removed. In other words: we need to verify what the vertical error would be if the point was not present.

Algorithm 3 shows the pseudo-code for the TIN decimation algorithm. It should be noticed that the implementation of this algorithm requires a method to delete/remove a vertex from a (Delaunay) triangulation, and that many libraries do not have one[3].

[2] A greedy algorithm is one that divides a complex problem into a series of easier steps, then solves it by making the locally optimal choice at each step. In our case the heuristic is the importance measure, ie the vertical error. See `https://en.wikipedia.org/wiki/Greedy_algorithm`.

[3] The implementation in SciPy does not allow to remove one vertex, but CGAL does (`www.cgal.org`)

Algorithm 3: TIN simplification by decimation

Input: A set of input points S, and the simplification threshold ϵ_{max}
Output: A triangulation $\mathcal{T}$ that consists of a subset of S and satisfies ϵ_{max}

```
1  T ← a triangulation of S ;
2  ε ← 0 ;
3  while ε < ε_max do
4      ε ← ∞ ;
5      q ← nil ;
6      for all p ∈ T do
7          remove p from T ;
8          τ ← the triangle in T that contains p ;
9          ε_τ ← the vertical error of p with respect to τ ;
10         if ε_τ < ε then
11             ε ← ε_τ ;
12             q ← p ;
13         put back p into T ;
       /* remove the point q that has the smallest error:                */
14     remove from T the point q ;
```

Observe that the Algorithms 2 and 3 both state that the importance of the points must be completely recomputed after each iteration of the algorithms (either one removal or one insertion), but that in practice several of these will not have changed. As can be seen in Lesson 03, the insertion/deletion of a single point/vertex will only *locally* modify the triangulation, and it is thus faster from a computational point of view to flag the vertices incident to the modified triangles, and only update these.

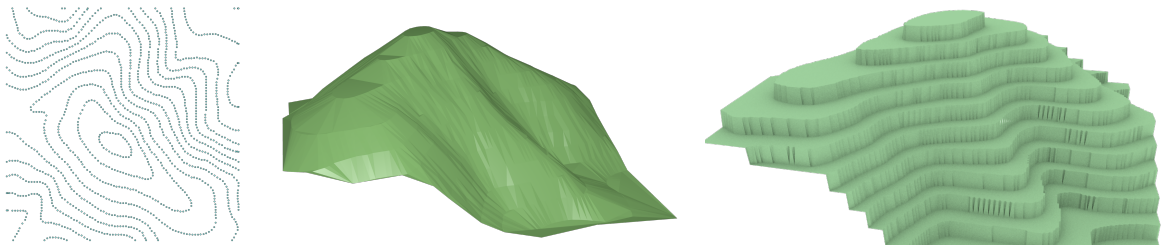
3.3 Comparison: decimation versus refinement

While both methods will allow us to obtain similar results, the properties of the resulting terrain are different. Consider the threshold ϵ_{max} that is used to stop the simplification process. If the refinement method is used, then it is guaranteed that the final surface of the terrain will be at a maximum of ϵ_{max} (vertical distance) to the 'real surface' because all the points of the input are considered. However, with the decimation method, after a vertex is deleted from the TIN, it is never considered again when assessing whether a given vertex has an error larger than ϵ_{max}. It is thus possible that the final surface does not lie within ϵ_{max}, although for normal distribution of points, it should not deviate too much from it.

In practice, refinement is often computationally more efficient than decimation because we do not need to first build a TIN from all input points before removing several of them again. However, decimation could be more efficient when you already have a detailed TIN, stored in a topological data structure, that just needs to be slightly simplified.

4 Conversion isolines to TIN/raster creates the 'wedding cake effect'

If the input is a set of isolines, then the simplest solution is, as shown in Figure 7a, to convert these to points and then use any of the interpolation methods previously discussed. This conversion can be done by either keeping only the vertices of the polylines, or by sampling



(a) (b) (c)

Figure 7: The 'wedding cake' effect. **(a)** The input isolines have been discretised into sample points. **(b)** The TIN of the samples creates several horizontal triangles. **(c)** The surface obtained with nearest-neighbour interpolation.

points at regular intervals long the lines (say every 10m). However, one should be aware that doing so will create terrains having the so-called *wedding cake effect*. Indeed, the TIN obtained with a Delaunay triangulation, as shown in Figure 7b, contains several horizontal triangles; these triangles are formed by 3 vertices from the same isoline. If another interpolation method is used, eg nearest neighbour (Figure 7c), then the results are catastrophic.

Solving this problem requires solutions specifically designed for such inputs. The main ideas for most of them is to add extra vertices between the isolines, to avoid having horizontal triangles. One strategy that has proven to work is to add the new vertices on the *skeleton*, or medial-axis transform, of the isolines, which are located 'halfway' between two isolines. The elevation assigned to these is based on the elevations of the isolines. Lesson 09 explains in details what the medial-axis transform is.

5 Notes & comments

The *Implicit Function Theorem* is further explained in Sibson (1997).

Dakowicz and Gold (2003) describe in details the skeleton-based algorithm to interpolate from isolines, and show the results of using different interpolation methods.

The basic algorithm to extract isolines, which is a brute-force approach, can be slow if for instance only a few isolines are extracted from a very large datasets: all the n triangles/cells are visited, and most will not have any intersections. To avoid this, van Kreveld (1996) build an auxiliary data structure, the *interval tree*, which allows us to find quickly which triangles will intersect a given elevation. It is also possible to build another auxiliary structure, the contour tree, where the triangle seeds are stored (van Kreveld et al., 1997). Such methods require more storage, but can be useful for interactive environment where the user extracts isolines interactively.

Garland and Heckbert (1995) elaborate further on different aspects of TIN simplification, such as different importance measures, the differences between refinement and decimation, and the usefulness of data-dependent triangulations. They also show how Algorithm 2 can be made a lot faster by only recomputing the importance of points in triangles that have been modified.

References & further reading

Dakowicz M and Gold CM (2003). Extracting meaningful slopes from terrain contours. *International Journal of Computational Geometry and Applications*, 13(4):339–357.

Garland M and Heckbert PS (1995). Fast polygonal approximation of terrain and height fields. Technical Report CMU-CS-95-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

Sibson R (1997). Contour mapping: Principles and practice. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.51.63`.

van Kreveld M (1996). Efficient methods for isoline extraction from a TIN. *International Journal of Geographical Information Science*, 10(5):523–540.

van Kreveld M, van Oostrum R, Bajaj C, Pascucci V, and Schikore D (1997). Contour trees and small seed sets for isosurface traversal. In *Proceedings 13th Annual Symposium on Computational Geometry*, pages 212–219. ACM Press, Nice, France.