Visualization of TINs*

Mark de Berg

Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract

Geographic information systems often represent terrains, or other height fields, by triangulated irregular networks (TINs). The visualization of TINs is therefore a task any GIS has to be able to perform. This survey paper discusses two aspects of this task: the hidden-surface removal problem, which is to determine which parts of the TIN are visible from a given view point, and the computation of data structures that allow the extraction of representations of the TIN at different levels of detail.

1 Introduction

One of the fundamental tasks any GIS has to perform is the visualization of geographic data. Often the data will be elevation data describing a terrain. How to visualize such data depends on the representation used for the terrain, that is, on the digital elevation model used. Three of the most popular models are the regular square grid, the contour-line model, and the triangulated irregular network. In this survey paper we shall concentrate on the visualization of triangulated irregular networks, or TINs for short. In this model a triangulation of a finite set of 2-dimensional data points is stored, together with the elevation of each data point. For our purposes we can regard a TIN as a collection of triangles in 3-dimensional space forming a connected surface which is z-monotone, that is, which is such that no two triangles intersect when projected vertically onto the xy-plane. Fig. 1 shows a TIN. (The example TIN is fairly regular—the triangles do not differ too much in size—but this need not be the case.)

We will start in Section 2 with an overview of the basics of visualization. This section also introduces the notational conventions we shall use throughout the paper.

The next two sections form the main part of this paper. They discuss two important topics that arise in the visualization of TINs.

The first is the *hidden-surface-removal problem*: given a TIN and a view point (or a viewing direction), we want to compute what we see of the TIN when we look at it from the given view point (or in the given viewing direction). Section 3 describes various approaches to the hidden-surface-removal problem.

The second topic, addressed in Section 4, concerns the following. The number of triangles in a TIN is often so large that it becomes infeasible to visualize all triangles in a reasonable amount of time. Fortunately this is not necessary, as large parts of the TIN will be far from the view point; these parts can be visualized at a lower level of detail, using less triangles. To

*This work was supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

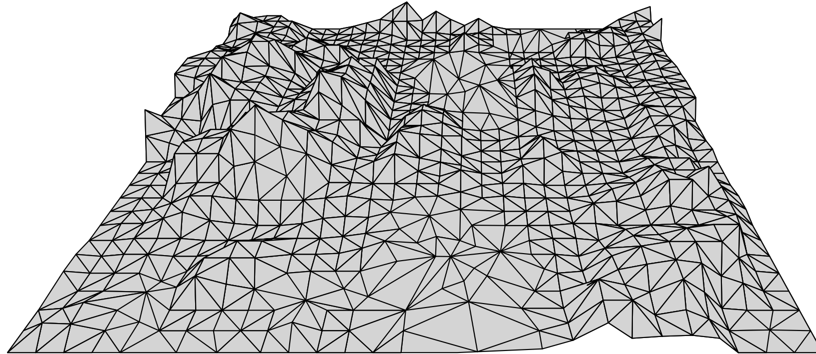

Figure 1: A triangulated irregular network (TIN).

make this work, one should pre-compute representations of the same TIN at various *levels of detail (LOD)*.

We close the paper in Section 5 with a short discussion of some topics that arise in the visualization of TINs and that we did not consider.

2 The basics

Below we briefly describe some of the most basic steps involved in *rendering* (that is, generating a picture of) 2- and 3-dimensional scenes. A more extensive treatment of these and many more topics in computer graphics can be found in the book by Foley et al. [15], or any other good graphics textbook.

A computer screen is composed of a large number of small dots, called *pixels*, which are arranged in a regular grid. The grid-size, or *resolution*, is typically about 1000×1000, resulting in a total of about $1,000,000$ pixels. Each pixel can be assigned a color. The colors of the pixels are stored in the *frame buffer*, an array with one entry per pixel. We denote the frame buffer by FrameBuf; thus FrameBuf$[x,y]$ stores the color of the pixel (x,y). Drawing a picture of a given scene now amounts to computing the correct colors for each pixel or, in other words, to filling in the frame buffer.

Now suppose we want to render a set of primitive objects—line segments, circles, triangles, and so on—in the plane. First we have to identify a rectangular area in the plane, the *window*, which delimits the region to be displayed on the screen. The objects are then *clipped* to the window: (the parts of) the objects that fall outside the window are discarded. Next we have to determine for each object a collection of pixels representing it, and set the entries in the frame buffer that correspond to these pixels to the color of their respective object. This process is called *scan-conversion*. Fig. 2 illustrates these processes.

To visualize a 3-dimensional scene we need some more steps. As in the 2-dimensional case, we need to specify a region of interest. This so-called *viewing volume* is now a 3-dimensional region, which is a rectangular block for parallel projections and a truncated piramid for perspective projections. The objects are clipped to the viewing volume, and then projected onto the *viewing plane*. The projection that is used depends on the type of view: if a view point is given then a perspective projection is used with the view point as the center of projection, and if a viewing direction is given then a parallel projection in the viewing

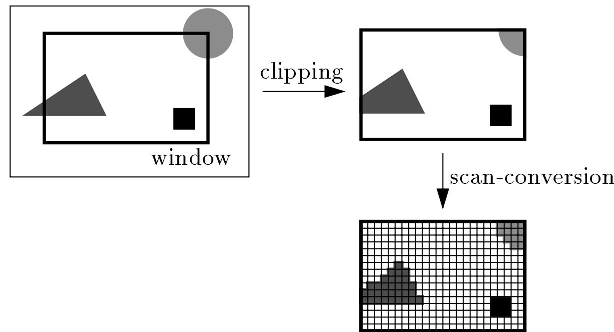

Figure 2: Clipping and scan-converting 2-dimensional objects.

direction is used. Finally, the projected objects are scan-converted: the pixels covered by the projected objects are determined and the corresponding frame-buffer entries are assigned the color of the respective objects.

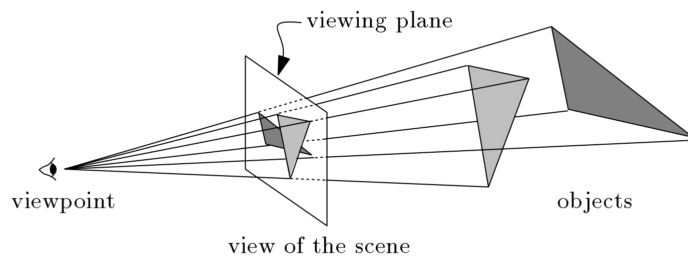

Figure 3: Visualizing a 3D scene: projection and hidden surface removal.

There is one important step in the generation of images of 3-dimensional scenes that we have ignored in the description above: objects can completely or partly be hidden behind other objects. Somehow we have to ensure that pixels that are covered by more than one projected object are assigned the color of the right object—the one closest to the view point. This is illustrated in Fig. 3, where the light grey triangle hides part of the dark grey triangle. Computing what is visible of a given scene and what is hidden is called *hidden-surface removal*.

3 Hidden-surface removal

There are two major approaches to perform hidden-surface removal [31].

One is to first determine which part of each object is visible, and then project and scan-convert only the visible parts. Algorithms using this approach are called *object-space algorithms*. We shall discuss some of these algorithms in Section 3.2.

The other possibility is to first project the objects, and decide during the scan-conversion for each individual pixel which object is visible at that pixel. Algorithms following this approach are called *image-space algorithms*. The Z-buffer algorithm, which is the algorithm most commonly used to perform hidden-surface removal, uses this approach. It works as follows.

Assume for simplicity that we wish to compute a parallel view of the scene. First a transformation is applied to the scene that maps the viewing direction to the positive z-direction. The algorithm needs, besides the frame buffer which stores for each pixel its color, a z-buffer. This is a 2-dimensional array ZBuf, where ZBuf$[x, y]$ stores a z-coordinate for pixel (x, y). The objects in the scene are clipped to the viewing volume, projected, and scan-converted in arbitrary order. The z-buffer stores for each pixel the z-coordinate of the object currently visible at that pixel—the object visible among the ones processed so far—and the frame buffer stores for each pixel the color of the currently visible object. The scan-conversion process is now augmented with a visibility test, as follows. When we scan-convert a (clipped and projected) object t and we discover that a pixel (x, y) is covered by t, we do not automatically write t's color into FrameBuf$[x, y]$. Instead we first check whether t is behind one of the already processed triangles at position (x, y); this is done by comparing $z_t(x, y)$, the z-coordinate of t at (x, y), to ZBuf$[x, y]$. If $z_t(x, y) <$ ZBuf$[x, y]$, then t is in front of the currently visible object, so we set FrameBuf$[x, y] :=$ color$_t$, where color$_t$ denotes the color of t, and we set ZBuf$[x, y] := z_t(x, y)$. (The color of t need not be uniform, so we should actually write color$_t(x, y)$ instead of just color$_t$.) If $z_t(x, y) \geq$ ZBuf$[x, y]$, we leave FrameBuf$[x, y]$ and ZBuf$[x, y]$ unchanged.

The Z-buffer algorithm is easy to implement, and any graphics workstation provides it, often in hardware. Nevertheless, there are situations where other approaches can be superior. In the next two subsections we describe two such approaches.

3.1 Depth-sorting methods

Depth-sorting methods [23] for hidden-surface removal scan-convert the objects in a back-to-front order, instead of in arbitrary order as the z-buffer algorithm does. This means that whenever an object is scan-converted, we know it is in front of all objects scan-converted thusfar. Hence, there is no need for the visibility test (the test on z-coordinate) anymore. This speeds up the algorithm, and it saves memory because the array ZBuf is no longer needed. Fig. 4 illustrates the approach. Because the algorithm resembles the way in which a

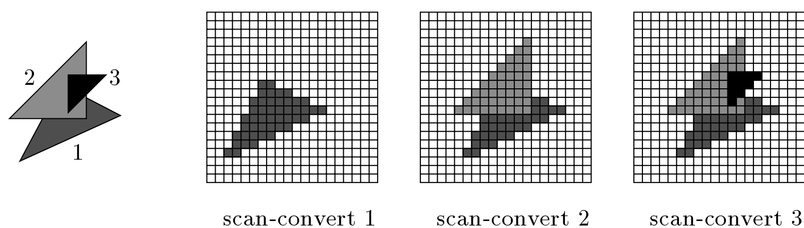

Figure 4: The painter's algorithm.

painter works—objects in the foreground are drawn 'on top of' objects in the background—, it is often called the *painter's algorithm.*

Although the painter's algorithm avoids the visibility test in the scan-conversion phase, it adds an extra preprocessing step: a back-to-front order, or *depth order*, must be computed. This is not always easy. Even worse, it is not always possible. Fig. 5 shows three triangles that cyclically overlap each other; no ordering of the triangles will result in a correct picture of the scene. In general it is not easy to detect and resolve such situations. (Resolving the

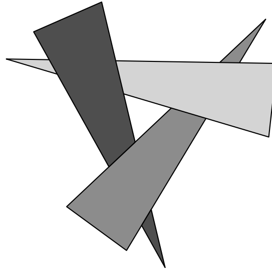

Figure 5: Three triangles that cyclically overlap each other.

situation is done by cutting one or several of the objects into pieces, and computing a depth order for the resulting collection of objects and object pieces.) But for TINs the problem of cyclic overlap does not occur—at least not for parallel views.

We now describe how to compute a depth order for the triangles in a TIN [2]. Let's first define more precisely what a depth order is. Assume that we want to compute a parallel view of the TIN, and let $\vec{d}$ denote the viewing direction. We say that a triangle t is *in front of* a triangle t' if there is line with direction $\vec{d}$ that first intersects t and then intersects t'. In other words, there is a point on t that hides some point on t' from the view. If t is in front of t', we write $t \prec t'$. A depth order for a collection T of triangles is an ordering $t_1, t_2, \ldots, t_n$ of the triangles in T such that $t_i \prec t_j$ implies $i > j$. This means that a triangle that is in front of another triangles should come later in the ordering.

The crucial observation that makes an efficient computation of a depth order possible for a TIN is that we can compute the order in the projection: we can project all triangles of the TIN onto the xy-plane, project the viewing direction as well, and compute a depth order for the resulting planar scene. This is possible because if a directed line in 3-dimensional space intersects two triangles, then the projected line intersects the projected triangles in the same order, provided that the projected triangles do not overlap. Fig. 6 illustrates this. Because the

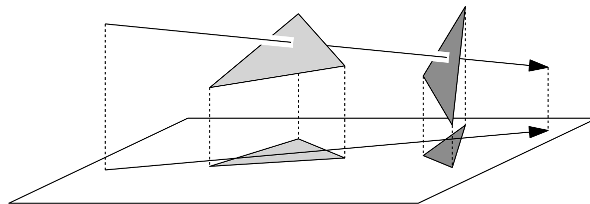

Figure 6: The order of intersection does not change by projection.

triangles form a TIN, their projections do not overlap. From now on, we let T refer to the set of projected triangles, and $\vec{d}$ to the projected viewing direction. The in-front-of relation will also refer to the projected triangles: a projected triangle t is in front of a projected triangle t' if there is a line in the plane with direction $\vec{d}$ that first intersects t and then t'. A depth order can now be computed as follows.

Let $\mathcal{G}_T$ denote the dual graph of the TIN. This graph has a node for every triangle, and there is an arc between two nodes if the corresponding triangles are adjacent. Assuming the TIN is stored in a suitable topological structure—a doubly-connected edge list [4, 26] for

instance—this graph is readily available. We turn $\mathcal{G}_T$ into a directed graph $\mathcal{G}_T(\vec{d})$ as follows: the arc connecting triangles t and t' is directed from t to t' if t is in front of t', it is directed from t' to t if t' is in front of t, and it is deleted if neither of the two is the case—see Fig. 7. (The third case can only occur if the common edge of the two triangles is parallel to the

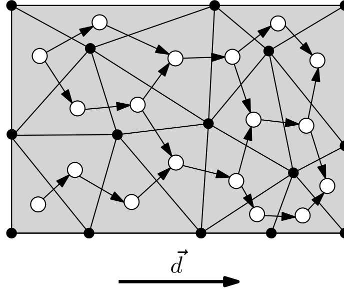

Figure 7: The directed graph $\mathcal{G}_T(\vec{d})$.

direction $\vec{d}$.) One can prove that $\mathcal{G}_T(\vec{d})$ is acyclic.

A topological order on a directed graph is an ordering of the nodes such that if there is an arc from node ν to node ν', then ν comes before ν' in the ordering. This implies that if there is a directed path in the graph from a node ν to a node ν', then ν must come first in the topological order. Now consider two triangles t and t' in T, and assume that t is in front of t'. This means that there is a line with direction $\vec{d}$ that first intersects t and then t'. If we follow this line from t to t', it intersects a number (possible zero) adjacent triangles on the way. Hence, there is a directed path from t to t' in $\mathcal{G}_T(\vec{d})$. This implies that a topological order for $\mathcal{G}_T(\vec{d})$ corresponds to a depth order for T.

Computing a topological order on a directed acyclic graph can be done in $O(V+E)$ time, where V and E are the number of nodes and arcs of the graph, respectively [6]. In our case the number of nodes equals the number of triangles, and the number of arcs is at most three times this number, because a triangle is adjacent to at most three other triangles. We conclude that it is possible to compute a depth order for a TIN consisting of n triangles in $O(n)$ time.

So far we assumed that we are given a viewing direction. Now let's see what happens if we want to compute a perspective view from a view point p_{view}. In this case we say that a triangle t is in front of another triangle t' if there is a ray starting at p_{view} that first intersects t and then t'. A depth order is now defined as above: it is an order consistent with the in-front-of relation. We can use the same approach as before, namely project the triangles and the view point p_{view}, and compute a depth order in the projection by determining a topological ordering on the directed dual graph $\mathcal{G}_T(p_{\text{view}})$. (The graph $\mathcal{G}_T(p_{\text{view}})$ is defined in the obvious way: its nodes correspond to the projected triangles and its arcs reflect the in-front-of relation between adjacent triangles.) There is one problem with this approach: we are no longer guaranteed that $\mathcal{G}_T(p_{\text{view}})$ is acyclic, so a topological order may not exist. Fig. 8 shows this phenomenon: the directed graph obtained for the indicated view point and the shaded triangles contains a cycle. (The dotted rays show the order in which pairs of triangles are intersected, which is reflected in the direction of the arcs in the dual graph.) This problem can be solved by splitting the TIN into two parts with a line through p_{view} and treating the two halves separately. If the triangles in the TIN form a Delaunay triangulation, then one

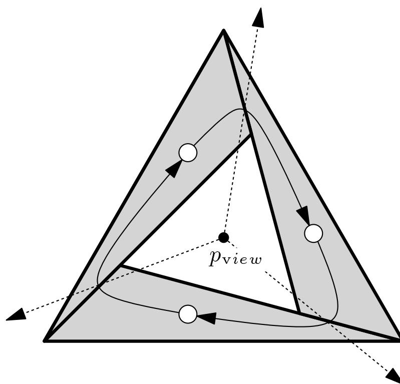

Figure 8: Three triangles and a view point leading to a cycle in the dual graph.

can prove that there are no cycles [8].

The approach we sketched above does not work for arbitrary 3-dimensional scenes. The problem is that the projections of the objects in an arbitrary scene are not necessarily disjoint, unlike for TINs.

An elegant way to implement the painter's algorithm for arbitrary scenes is to use BSP trees. A depth order for the scene can now be obtained by a traversal of the BSP tree [4, 17].

3.2 Object-space methods

Image-space methods compute the view of a scene pixel by pixel. This means that the 'structure' of the view is lost. Object-space algorithms compute a combinatorial representation of the view. Let's be more precise about this. The view of a scene is a subdivision of the viewing plane into maximal connected regions in each of which (some portion of) a single object can be seen, or no object is seen. In Fig. 3, for example, this subdivision consists of four regions; the light grey triangle is visible in one of them, the dark grey triangle is visible in two regions, and no object is seen in the fourth region. Object-space algorithms compute this subdivision—the *visibility map* of the given set of objects—as a collection of (polygonal) faces. In other words, object-space algorithms compute exactly which part of each object is visible. After that the visible parts can be projected and displayed without difficulty, because any pixel will be covered by at most one projected part.

In general, object-space hidden-surface-removal methods tend to be slower than image-space methods such as the z-buffer algorithm, because they cannot be implemented in hardware very well. However, object-space algorithms have certain advantages over image-space methods. Suppose we want to display the hidden lines in a scene dashed, instead of making them invisible. Object space algorithms compute exactly which parts of each line are visible and which parts are not, thus making it easy to display the invisible parts dashed. Image-space algorithms do not provide the information necessary to achieve this. Another weak point of image-space algorithms comes up when one wants to print the view of a scene on paper, instead of displaying it on the screen of a computer terminal. When hidden-surface removal has been done in image space, the only thing one can do is to plot every pixel separately. But this method fails to take advantage of the fact that the resolution of modern laser printers is much higher than the resolution of computer screens. If hidden-surface removal

has been performed in object space then the visibility map can be processed directly, resulting in a picture of higher quality. Of course it is possible to use an image-space hidden-surface removal algorithm at the printer's resolution, but due to the very large number of pixels this tends to be slow. A third advantage of object-space algorithms is that they can be used to compute shadows in a scene, because the part of a scene lit by a light source is exactly the same as what can be seen by an observer standing at the light source. Finally, the fact that an object-space method computes exactly what is visible from a given view point means that its output is more suitable to perform *viewshed analysis* (that is, to analyze the view), which is useful in several GIS applications.

In theory, the complexity of the visibility map—the total number of vertices of the visible pieces—of a set of n triangles can be as high as $\Theta(n^2)$, even when the triangles form a TIN. This is illustrated in Fig. 9, where each of the long and 'horizontal' triangles of the hill in the back is cut into a linear number of visible pieces by the spikes in the front. This implies

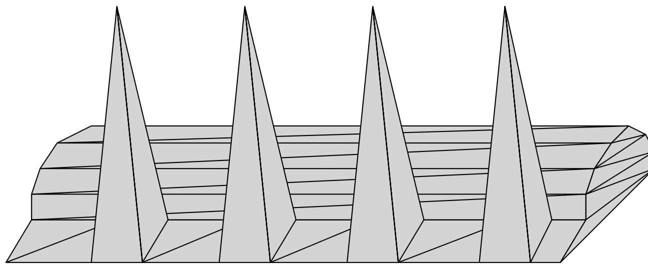

Figure 9: A view of a TIN giving rise to a quadratic-complexity visibility map.

that any object-space hidden-surface-removal algorithm must take $\Omega(n^2)$ time in the worst case. There are algorithms with an $O(n^2)$ running time [12, 22], which are thus optimal in the worst case. But obviously the worst case (a quadratic-size visibility map) usually does not occur in practice. Hence, it is useful to try and find *output-sensitive algorithms*: algorithms whose running time not only depends on n, the number of input triangles, but also on k, the complexity of the output (the visibility map in our case). Such algorithms will be faster when k is small. Dorward [13] gives an extensive overview of output-sensitive hidden-surface-removal algorithms, and de Berg's book [2] also contains an ample discussion of object-space hidden-surface removal. In the following, we shall concentrate on algorithms that are especially efficient for TINs. To simplify the description of the algorithms we assume that we want to compute a parallel view of the TIN. Let $\vec{d}$ be the viewing direction.

One approach to obtain an output-sensitive hidden-surface algorithm is the following. We have seen Section 3.1 that it is always possible to order the triangles of the TIN in a back-to-front order with respect to the viewing direction. The idea is to treat the triangles in the reverse order (that is, a front-to-back order), and to maintain the contour of the triangles—the union of their projection onto the viewing plane—processed thusfar. A triangle t is now handled as follows. First, the visible portion of t is determined by computing which part of its projection lies outside the current contour. Next, the contour is updated by computing the union of the current contour and t—see Fig. 10. Reif and Sen [27] have shown that the contour can be stored in such a way that both tasks—computing the visible portion of the triangle t and computing the new contour—can be performed in $O((k_t + 1) \log n \log\log n)$

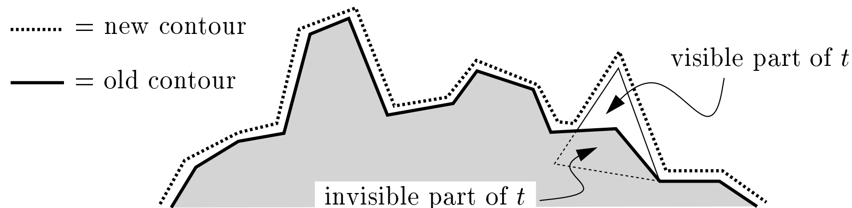

Figure 10: Illustration of the approach of Reif and Sen.

time, where k_t is the complexity of the visible portion of t. This leads to an algorithm with a running time of $O((n+k)\log n \log\log n)$, where k is the complexity of the visibility map.

Another approach is described by Katz et al. [20]. We now describe their method in more detail.

Let T be a collection of triangles in 3-space. In our case the triangles form a TIN, but the method of Katz et al. works for any set of triangles (or other objects) provided that a depth order exists.

The first step in the algorithm is to construct a balanced binary tree $\mathcal{T}$ that stores the triangles in its leaves, where the left-to-right order of the leaves corresponds to a depth order on the triangles. For a node ν of this tree, we let $T(\nu)$ denote the set of triangles stored in the subtree rooted at ν. Thus $T(\text{root}(\mathcal{T})) = T$.

The next step is to compute for each node ν the sets $U(\nu)$ and $V(\nu)$, which are defined as follows: $U(\nu)$ is the union of the projections of the triangles of $T(\nu)$ onto the viewing plane, and $V(\nu)$ is the visible part of $U(\nu)$. More precisely, $V(\nu)$ is defined as follows. Let $U_{\text{left}}(\nu)$ denote the union of the projections of the triangles stored in leaves to the left of $T(\nu)$. In the depth order, these triangles are the ones that come before the triangles in $\mathcal{T}(\nu)$, and so they may hide them from the view. $V(\nu)$ is now defined as $U(\nu) - U_{\text{left}}(\nu)$. Fig. 11 illustrates these definitions. If we let ν_t denote the leaf of $\mathcal{T}$ that stores triangle $t \in T$, then,

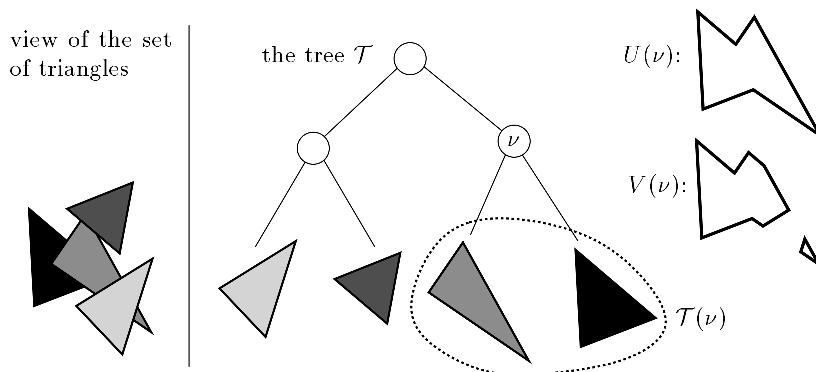

Figure 11: The tree $\mathcal{T}$ and the sets $U(\nu)$ and $V(\nu)$.

by definition, the visible part of t is $V(\nu_t)$. Hence, after having computed the sets $V(\nu)$ for all nodes (including the leaves) in $\mathcal{T}$, we have solved the hidden-surface-removal problem. To compute the regions $V(\nu)$, we first compute the regions $U(\nu)$, using the following relation. Let lchild(ν) and rchild(ν) denote the left and right child of node ν respectively. Then we

have

- $U(\nu) = U(\text{lchild}(\nu)) \cup U(\text{rchild}(\nu))$.

For a leaf ν, the region $U(\nu)$ is simply the projection of the triangle stored at that leaf. Hence, the relation above implies that the sets $U(\nu)$ can be computed by a simple procedure working in a bottom-up manner. The basic operation in this procedure is to compute the union of two polygonal sets (namely the unions of the children of a given node) in the plane. This can easily be done with a plane sweep algorithm.

Once we have computed $U(\nu)$ for each node ν we can compute the sets $V(\nu)$ in a top-down fashion by using the following relations:

- $V(\text{root}(\mathcal{T})) = U(\text{root}(\mathcal{T}))$,
- $V(\text{lchild}(\nu)) = V(\nu) \cap U(\text{lchild}(\nu))$,
- $V(\text{rchild}(\nu)) = V(\nu) - U(\text{lchild}(\nu))$.

The basic operations in the top-down procedure are intersection and difference computations on polygons regions. Again, this can be done by a plane sweep.

For the efficiency of the method it is important that the regions $U(\nu)$, which are computed to aid the computation of the regions $V(\nu)$, are not too complex. If the set of triangles forms a TIN, then one can prove that this is indeed the case: the complexity of the region $U(\nu)$ is basically linear in the number of triangles in $T(\nu)$. To be precise, it is at most $O(n_\nu \alpha(n_\nu))$, where n_ν is the number of triangles in $T(\nu)$ and where $\alpha()$ denotes the extremely slowly growing functional inverse of Ackermann's function [1]. This can be used to show that the entire hidden-surface removal algorithm can be implemented so that on a TIN of n triangles it runs in $O(n\alpha(n) \log n + k \log n)$ time, where k denotes the complexity of the visibility map.

4 Levels of detail

For a realistic representation of a terrain millions of triangles are needed. In applications such as flight simulation a terrain should be rendered at real time, but even with modern technology it is impossible to achieve this when the number of triangles is this large. Fortunately, a realistic image of the terrain is only crucial when one is close to the terrain and only a small part of the terrain is visible; when one is flying high above the terrain a coarse representation suffices. So what is needed is a *multiresolution model*: a hierarchy of representations at various levels of detail. This makes it possible for a given view point to render the terrain at an adequate level of detail. Fig. 12–14 shows an example of a hierarchy consisting of three levels. On the left perspective (and shaded) 3D views of the terrain are shown, and on the right 2D views of the underlying TIN.

One important property required from such a hierarchy is that subsequent levels should not differ too much in appearance: switching to more and more detailed representations when zooming in should not cause disturbing 'jumps' in the image. On the other hand, the reduction of the number of triangles in subsequent levels should not be too small, otherwise too many levels would be needed, resulting in an unacceptable increase in storage.

Another desirable property is the following. It is in general insufficient for rendering purposes to use only one level of detail at a time: although some part of the terrain may be close to the view point, another part—the horizon, for example—can be far away. Hence, one

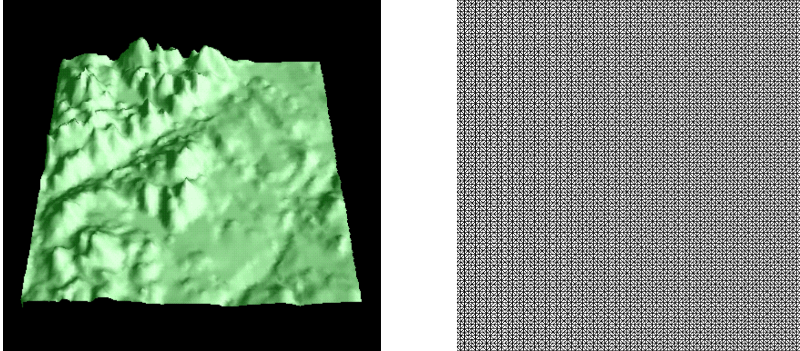

Figure 12: The first, most detailed, level of the hierarchy.

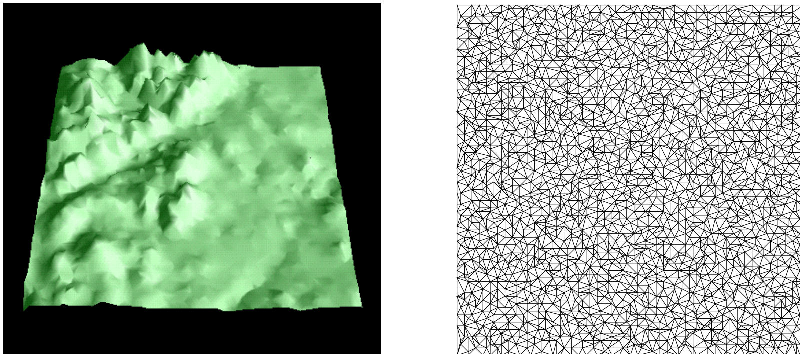

Figure 13: The second level of the hierarchy.

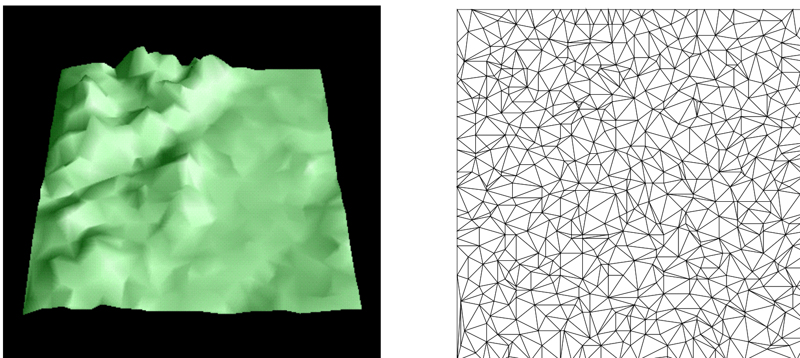

Figure 14: The third level of the hierarchy.

would like to combine parts from different levels into a single *variable-resolution representation* of the terrain, such that each part of the terrain is rendered with appropriate detail. This means that the levels cannot be completely independent, as it should be possible to glue them together smoothly.

The idea of multiresolution models is quite old, and work on it is scattered over literature in graphics, GIS, and other areas. Heckbert and Garland [18] and De Floriani et al. [10] give nice surveys of many of the existing multiresolution techniques. A related problem is the problem of simplifying general surfaces [19, 14] or terrains [16, 5, 21, 24].

The best known hierarchical data structure is probably the quadtree [29, 28]. Although quadtrees are based on regular grids, not on TINs, we discuss them briefly because they are so widely used as multiresolution model. Fig. 15 shows a three-level hierarchy based on a quadtree. The least detailed level simply consists of one square, namely the square corresponding to the root of the quad tree. The next level consists of the four squares corresponding to the children of the root; these squares are exactly the quadrants of the root square. In general, to obtain the next more detailed level from a given level, each square is replaced by its four quadrants. For rendering purposes, each square is split into two triangles with a diagonal. Extracting a representation where different parts of the terrain

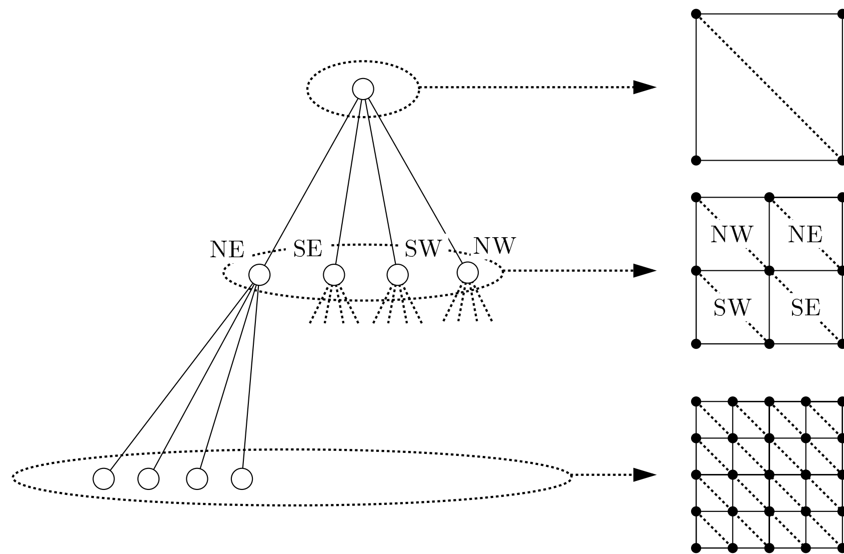

Figure 15: A three-level hierarchy based on a quad tree.

use a different level of detail is quite easy when a quadtree is used. Consider the three-level hierarchy of Fig. 15 and suppose that we want to render the terrain for a view point located close to the SW corner of the terrain. Hence, we want a representation where the part of the terrain close to the SW corner is detailed, and parts further away are less detailed. Fig. 16 shows such a representation. In general, one simply performs a top-down traversal of the quadtree, proceeding at every node as follows: if the square corresponding to that node is detailed enough with respect to the given view point, then (the two triangles of) the square are rendered, otherwise the four children of the square are visited recursively. In the given example, the traversal starts at the root, where it is decided that the corresponding square is not detailed enough. Next, the four children of the root are visited. At three of the children

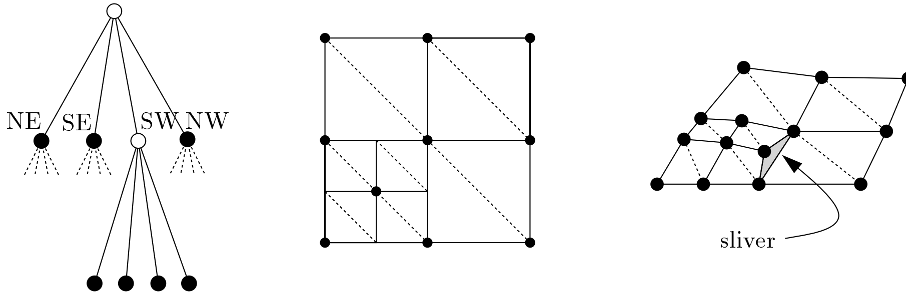

Figure 16: Extracting a variable-resolution representation from a quad tree.

it is decided that the corresponding squares are detailed enough, since they are far away from the view point. The SW square, however, is not detailed enough, since it is close to the view point. Hence, its children are visited recursively. Finally, the squares corresponding to these children are accepted, as they are detailed enough (or perhaps, in this example, because a more detailed representation is not available).

Using a quadtree as a multiresolution representation has two drawbacks. First of all, it assumes that the underlying representation is a regular grid; a quadtree it is not suitable for TINs. Second, although the parts that use different detail levels combine nicely in 2D, the terrain may no longer be continuous when the height of the sample points is taken into account. As a result, *slivers*—small 'cracks' in the terrain—may be visible, as can be seen in Fig. 16.

We now turn our attention to multiresolution models for TINs. These models can be subdivided into two categories.

In the first category one starts with a triangulation of a small subset of the data points [9, 25]. This is the coarsest representation. To obtain the next level, the triangles are refined by adding new data points inside them and retriangulating each triangle with its new interior points. Thus each triangle is replaced by a number of smaller triangles. This process is repeated until all data points have been added, or some precision criterion is met. Such a hierarchy can be modeled as a tree. The nodes in this tree correspond to the triangles in the hierarchy, and there is an arc from the node corresponding to a triangle t to the node corresponding to a triangle t' if the triangles belong to consecutive levels and t' is contained in t. There is also a root node, which is connected to all triangles of the first level. Fig. 17 shows an example of a hierarchy and the corresponding tree. Note the similarity of this model

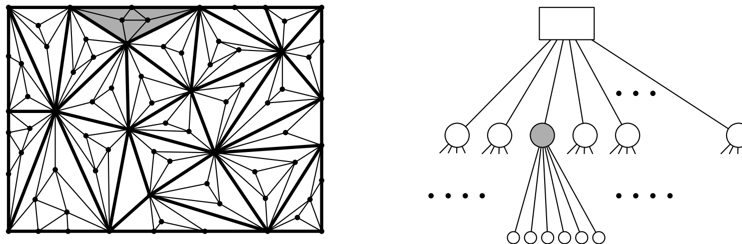

Figure 17: A two-level hierarchy with a tree structure.

with the quad tree: in both cases an 'elementary shape' at a given level (a triangle or, for quadtrees, a square) is replaced by a number of smaller elementary shapes at the next level.

For tree-like hierarchies it is easy to combine different levels into one representation: one can use the same approach as for quadtrees, that is, perform a top-down traversal of the tree accepting the triangle of a node when it is detailed enough and visiting the children of the node otherwise.

Unfortunately, tree-like hierarchies have a serious drawback: the triangles at higher detail levels are very skinny, because the edges of the initial triangulation remain present at more detailed levels. (There are methods that try to avoid this by adding extra points on the edges of the triangles [30, 11]. The problem with this approach is that the introduction of extra vertices on the edges may cause slivers.) This effect is already apparent in the two-level hierarchy of Fig. 17. Skinny triangles can cause robustness and aliasing problems.

The second category uses the Delaunay triangulation [26] of the set of data points at every level. This triangulation has the nice property that it maximizes the minimum angle of the triangles [4]. Thus robustness and aliasing problems are reduced. The hierarchies of this category can be represented by directed acyclic graphs: the nodes correspond to the triangles in the hierarchy, and there is an arc from the node corresponding to a triangle at a certain level to a triangle at the next level if these triangles intersect. An example of such a hierarchy is the *Delaunay pyramid* [7], which is obtained by always adding the data point with the maximal error and retriangulating using the Delaunay criterion.

Fig. 18 shows a two-level hierarchy of this type. Left in this figure is the Delaunay triangulation of a subset of the points, and in the middle is the Delaunay triangulation of the whole set. The corresponding graph structure is shown on the right. For these hierarchies it

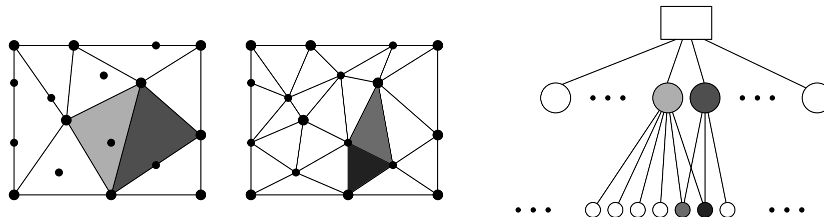

Figure 18: A two-level hierarchy that uses the Delaunay triangulation.

is not as easy to obtain a variable-resolution representation as for tree-like hierarchies. The problem is that one cannot decide whether or not to refine a triangle indepently from the other triangles. Suppose, for instance, that in Fig. 18 we decide that the light grey triangle should be refined, but that the dark grey triangle is detailed enough (because it is slightly further from the view point). Then we have a problem, because the children of the light grey triangle overlap with the dark grey triangle.

De Berg and Dobrindt [3] have shown that if the hierarchy is constructed in a slightly different manner, then it is still possible to extract a variable-resolution representation. Their idea is to construct the hierarchy as follows. They start with the most detailed level, remove a subset of the vertices and retriangulate. The Delaunay triangulation thus obtained is the one-but-finest level of detail in the hierarchy. To obtain the next level, another subset of vertices is removed, and a triangulation of the remaining vertices is computed, and so on. The vertices to be removed in a single stage must satisfy the following property: no two should be adjacent.

This means that the triangulation at a given level is obtained by replacing several groups of triangles at the previous level with groups of fewer triangles covering the same area—similar to tree-like hierarchies, where a single triangle is replaced by a group of triangles covering the same area. De Berg and Dobrindt show how to extract a variable-resolution representation, based on these groups, from the hierarchy. Fig. 19 shows an example of (a 2D view of) a variable-resolution representation obtained with their algorithm for the indicated viewpoint.

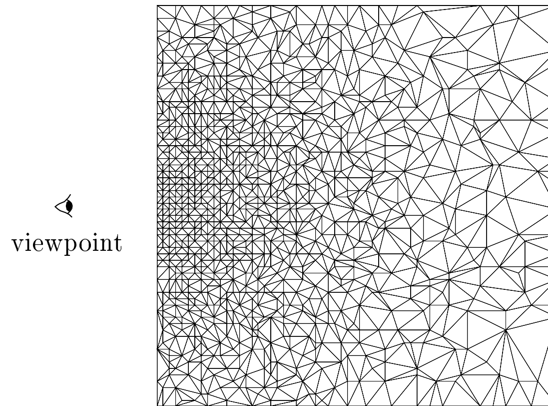

Figure 19: A variable-resolution representation of a terrain.

As already remarked, there are many more papers dealing with multiresolution models. The examples we gave above should give a fairly good idea of the issues involved. Readers who want to know more should consult the survey papers by Heckbert and Garland[18] and by De Floriani et al. [10]. The first paper considers multiresolution models mainly from a rendering point of view and considers not only TINs but arbitrary 3D objects, whereas De Floriani et al. restrict their discussion to multiresolution models for TINs.

5 Other issues

We have discussed two problems arising when one wants to visualize a TIN: hidden-surface removal and multiresolution models. There are many more issues that play a role, especially when one is interested in generating realistic images of a natural terrain. In that case one needs to take lighting considerations into account—which means, among other things, that one needs to compute shadows—, one needs to use so-called texture mapping, or other techniques, to make the terrain look more natural, and so on. Readers interested in these more advanced topics should consult the general graphics literature dealing with realistic rendering—Chapters 14–16 of *Computer Graphics: Principles and Practice* [15] are a good starting point.

References

[1] P. K. Agarwal and M. Sharir. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, Cambridge, UK, 1995.

[2] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal.* Lecture Notes in Computer Science 703, Berlin, 1993.

[3] M. de Berg and K.T. Dobrindt. On levels of detail in terrains. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C26–C27, 1995.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Heidelberg, 1997.

[5] Z. Chen and J. A. Guevara. System selection of very important points (VIP) from digital terrain model for constructing triangular irregular networks. In *Proc. 8th Internat. Sympos. Comput.-Assist. Cartog. (Auto-Carto)*, pages 50–56, 1988.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[7] L. De Floriani. A pyramidal data structure for triangle-based surface representation. *IEEE Comput. Graph. Appl.*, 9:67–78, March 1989.

[8] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. On sorting triangles in a Delaunay tessellation. *Algorithmica*, 6:522–532, 1991.

[9] L. De Floriani, B. Falcidieno, G. Nagy, and C. Pienovi. Hierarchical structure for surface approximation. *Comput. Graph. (UK)*, 8(2):183–193, 1984.

[10] L. De Floriani, P. Marzano, and E. Puppo. Hierarchical terrain models: Survey and formalization. In *Proc. ACM Sympos. Applied Comput.*, 1994.

[11] L. De Floriani and E. Puppo. A hierarchical triangle-based model for terrain description. In *Proc. Internat. Conf. GIS: Theory and Methods of Spatio-temporal Reasoning in Geographic Space*, Lecture Notes in Computer Science, pages 236–251. Springer-Verlag, 1992.

[12] F. Dévai. Quadratic bounds for hidden line elimination. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 269–275, 1986.

[13] S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.

[14] Nira Dyn, David Levin, and Samuel Rippa. Data dependent triangulations for piecewise linear interpolation. *IMA Journal of Numerical Analysis*, 10:137–154, 1990.

[15] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1990.

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

[17] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[18] P. S. Heckbert and M. Garland. Multiresolution modeling for fast rendering. In *Proc. Graphics Interface '94*, pages 43–50. Canadian Inf. Proc. Soc., 1994.

[19] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proc. SIGGRAPH '93*, pages 19–26, 1993.

[20] M. J. Katz, M. H. Overmars, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Comput. Geom. Theory Appl.*, 2:223–234, 1992.

[21] J. Lee. A drop heuristic conversion method for extracting irregular networks for digital elevation models. In *Proc. of GIS/LIS '89*, pages 30–39, 1989.

[22] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Trans. Graph.*, 6:19–28, 1987.

[23] M.E. Newell, R.G. Newell, and T.L. Sancha. A solution to the hidden surface problem. In *Proc. ACM Natl. Conf.*, pages 443–450, 1972.

[24] Michael F. Polis and David M. McKeown, Jr. Issues in iterative TIN generation to support large scale simulations. *Proc. of 11th Intl. Symp. on Computer Assisted Cartography*, 1993.

[25] J. Ponce and O. Faugeras. An object centered hierarchical representation for 3d objects: the prism tree. *Comput. Graphics and Image Proc.*, 38(1):1–28, 1987.

[26] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[27] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithms and its parallelization. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 193–200, 1988.

[28] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, 1990.

[29] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[30] L. Scarlatos and T. Pavlidis. Adaptive hierarchical triangulation. In *Proc. 10th Internat. Sympos. Comput.-Assist. Cartog. (Auto-Carto)*, pages 234–246, 1990.

[31] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, March 1974.