Multiresolution Visualization and Compression of Global Topographic Data

Thomas Gerstner
Department for Applied Mathematics,
University of Bonn, Germany
e-mail: gerstner@iam.uni-bonn.de

Abstract

We present a multiresolution model for surfaces which is able to handle large–scale global topographic data. It is based on a hierarchical decomposition of the sphere by a recursive bisection triangulation in geographic coordinates. Error indicators allow the representation of the data at various levels of detail and enable data compression by local omission of data values. The resulting hierarchical triangulation is stored using a bit code of the underlying binary tree and, additionally, relative pointers which allow an adaptive tree traversal. This way, it is possible to work directly on the compressed data. We show that significant compression rates can be obtained already for small threshold values. In a visualization application, adaptive triangulations which consist of hundreds of thousands of shaded triangles are extracted and drawn at interactive rates.

Classification: 65D05, 65N50, 65Y25, 68U05

Keywords: multiscale modeling, recursive bisection triangulation, error indicators, space filling curves

1 Introduction

The handling of large–scale topographic data is a challenging task. The amount of data e.g. available from satellite measurements can easily require gigabytes of memory. The user usually faces the serious problem how to store and process the data in a reasonable way. To this end, several strategies have been derived to handle such vast amounts of data. Hierarchical structuring allows fast data access and search operations. Adaptive methods enable concentration on the interesting parts of the data. Finally, data compression allows the efficient storage of the data on background media or in memory.

Our goal is to show that these approaches can be combined to enable the interactive processing of global topographic data. Typically, the amount of data which can be processed interactively is much smaller than the data which can be stored in main memory. Memory size in turn might be too small for all the data available. Therefore, our strategy will be to compress the complete data set to make it fit into memory or simply to make it more manageable. This is done in such a way that interactive processing of the compressed data is possible.

The main ingredients of our approach are multiresolution digital elevation models (DEMs) based on a recursive bisection triangulation of the unit sphere in geographic coordinates (i.e. longitude and latitude). An error indicator value is assigned to each triangle in the hierarchical triangulation. Then, by choosing a threshold value ε and selecting all triangles whose indicator value is larger than ε, it is possible to construct different approximations of the topographic data at varying levels of detail.

In order to achieve compression, we select a threshold ε_0 and construct a fine resolution adaptive hierarchical triangulation from a given input DEM. The threshold ε_0 can be related to the data error of the input or it might simply be chosen to meet the user's memory requirements. If the resulting approximation error is smaller than the input error, the compression can be considered lossless; larger ε_0 will result in a lossy compression of the data. The hierarchical triangulation is stored using a bit code of the underlying binary tree and relative branch pointers for an adaptive tree traversal. Numerical results for a 30 arc second global topographic data set show that significant compression rates can be achieved already for small ε_0.

Our main application is the visualization of the data using color shaded triangles. A second threshold $\varepsilon_1 \geq \varepsilon_0$ controls the visual approximation of the compressed data. We show that hundreds of thousands of triangles can be extracted from the multiresolution model and drawn interactively on modern graphics workstations. Hereby, the hierarchical structure allows arbitrary moving and zooming through the data set at consistent frame rates.

Furthermore, the hierarchical nature of the multiresolution model allows a very simple addition of local data sets to the global model. We show this with the help of two additional nested local data sets, one covering the lower Rhine valley and another one covering the city of Bonn. For the additional data sets separate hierarchical triangulations are built similar to the construction for the global data set. The new data sets fit smoothly into the global model by a simple merging procedure of the corresponding binary trees.

The remainder of this paper is organized as follows. In Section 2 different types of multiresolution DEMs are compared with respect to their storage requirements. From the great variety of models we select a well known adaptive hierarchical triangulation scheme whose construction is explained in more detail in Section 3. Some basic properties of these triangulations and their relation to space–filling curves are then explained in Section 4. In Section 5, these properties are used to define a compact storage scheme for adaptive triangulations and to construct corresponding algorithms working on the compressed data. In Section 6 it is then shown how these techniques can be applied to the visualization of global topographic data. Numerical results concerning compression and visualization of several data sets are presented in Section 7. The paper concludes in Section 8 with some remarks about further extensions and applications of the introduced methods.

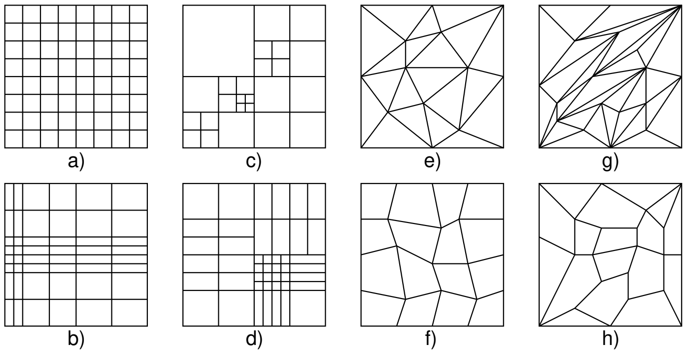

Figure 1: Examples for spatial decompositions used in DEMs.

2 Related Work

There are many ways to construct a multiresolution hierarchy on a DEM. Since one of our primary concerns is data compression, we will distinguish between them with respect to their storage requirements.

A DEM is defined as a set of points $(x_i, y_i, h(x_i, y_i)), 1 \leq i \leq N$, and interpolation rules to derive height values in between. An interpolation rule usually incorporates information on how (e.g. a polynomial degree) and from where (e.g. a local mesh topology) data is interpolated. The height values $h(x_i, y_i)$ are always fixed given data and have to be stored for every DEM. In this context, DEMs can roughly be classified according to how much additional storage is required for the coordinates of the points (x_i, y_i) and for the interpolation rules. If the additional storage requirement is of order $O(N)$, the respective set of data is called explicit. If the order is lower, it is called implicit since the additional memory has no practical influence on the total size. This leads us to the following four classes of DEMs with increasing generality, storage requirements and approximation quality:

- coordinates and interpolation implicit: regular grids (Figure 1a), graded meshes (Figure 1b),
- coordinates implicit, interpolation explicit: quadtrees (Figure 1c), adaptive tensor product grids (Figure 1d),
- coordinates explicit, interpolation implicit: TINs, Delaunay triangulations (Figure 1e), relocated regular grids (Figure 1f),
- coordinates and interpolation explicit: data–dependent triangulations (Figure 1g), general tesselations (Figure 1h).

Note that global transformations and block–structuring usually do not change the class. Clearly, more explicit schemes allow a greater flexibility than less explicit ones. But typically also the complexity of corresponding algorithms and supporting data structures grow. Therefore, it is not clear which model is most advantageous with respect to the ratio of overall cost to flexibility.

The first step in order to build a multiresolution DEM is to construct a series of approximations of a given input DEM. There is a great variety of methods which achieve this, usually classified as top–down (refinement) and bottom–up (decimation) methods [13, 21]. In the first approach starting with a coarse approximate DEM, single elements (such as points, edges or triangles) are inserted until the input DEM is reached. The second approach starts with the input DEM and removes elements successively. In some cases, the resulting meshes are optimized.

There exist three basic types (see [1]) of multiresolution DEMs, namely pyramidal (layered), incremental (evolutionary, historical) and tree–structured models. Pyramidal models consist of a small number of approximate DEMs with different resolutions. The memory overhead is limited by a constant factor if the number of points in the different approximate DEMs form a geometric series. Incremental models code the single insertion steps of a refinement method or the inverted removal steps of a decimation method. Thereby, a much larger number of possible approximations of the input DEM can be obtained. An incremental model can be turned into a tree–structured model by the identification of hierarchical independencies in the single incremental steps. While the first two methods allow only global representations, tree–structured models also enable local refinement.

Tree-structured models are best suited for our visualization application since often only a small subset of the whole data set can be processed and displayed interactively. Also, these models allow the definition of areas of greater interest, using e.g. magnifying glasses [10] or level of detail depending on the viewer's position and viewing direction [2, 15, 19].

3 Hierarchical Triangulation

In the following, we will consider DEMs based on a particular kind of adaptive hierarchical triangulations. In light of the previous section, they belong to the class of DEMs with implicit coordinates and explicit interpolation. As we will see, these DEMs allow very efficient algorithms and have a good ratio of storage requirement to accuracy. In this section we describe the construction of adaptive triangulations with the help of error indicators.

3.1 Recursive Bisection

We consider an approach here which is usually called longest edge bisection or split newest vertex triangulation and which is used by many authors for adaptive grid refinement in the numerical solution of partial differential equations [16,

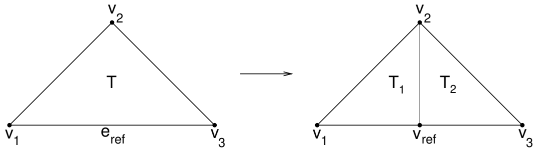

Figure 2: Triangle T is bisected into two subtriangles T_1 and T_2.

17, 23]. Basically the same method has been successfully applied also for the representation of terrain data [2, 4, 5, 6, 15, 20, 22]. It is closely related to restricted quadtrees [19, 25]. But triangulations generated by recursive bisection are more flexible than those generated by a triangulation of the leaves of a quadtree.

The main idea is to start with a coarse triangulation $\mathcal{T}^0$ and to recursively construct finer triangulations $\mathcal{T}^{l+1}$ by splitting each triangle $T \in \mathcal{T}^l$ in two. Here, we use isosceles triangles $T = (v_1 v_2 v_3)$ with a right angle at v_2. By the selection of the midpoint of the longest edge $e_{ref}(T) = (v_1 v_3)$ as the refinement vertex $v_{ref}(T)$, two new triangles $T_1 = (v_2 v_{ref} v_1)$ and $T_2 = (v_3 v_{ref} v_2)$ are generated (Figure 2).

Let us now consider a square regular gridded DEM with $2^k + 1$ height values $h(x_i, y_i)$ in every direction. If the initial triangulation consists of two triangles covering the square, all refinement vertices up to the finest resolution fall onto grid points of the DEM (Figure 3). This way, a piecewise linear surface over the triangulation can be defined. Of course, this construction can be adapted to more general settings. But a restriction to square dyadic grids simplifies subsequent algorithms greatly, since no mapping from the grid points to the vertices of the hierarchical triangulation is necessary.

Clearly, by this refinement procedure a binary tree hierarchy is inferred on the triangles. However, all refinement vertices are shared by two triangles except on the boundary. For example, the common refinement vertex of the two initial triangles is the center of the square (see also Figure 3). Therefore, no tree structure is induced on the set of vertices. Vertices on the boundary have one direct ancestor while vertices in the interior of the domain have two.

3.2 Error Indicators

Now, an adaptive triangulation can simply be defined through selection of a subtree of the triangle binary tree. However, such triangulations can contain hanging nodes which occur if two triangles sharing a refinement vertex are not refined conformingly. Hanging nodes are undesirable because they will lead to cracks in the DEM since the surface defined by the triangulation is no longer continuous. There are several ways to avoid this problem whereby the most commonly used ones are remeshing, filling and projection [16, 17, 18].

We want to solve the problem with the help of error indicators. To this end, a suited error indicator value $\eta(v_{ref})$ is assigned to every refinement vertex

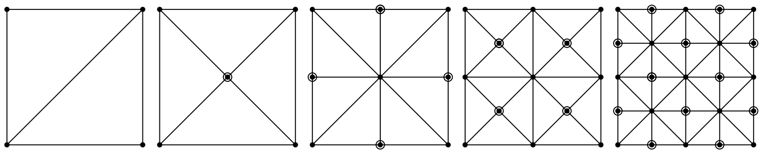

Figure 3: Triangle hierarchy generated by recursive bisection. Refinement vertices are marked by a circle.

v_{ref}. Now, if the error indicator values are saturated, a valid triangulation can be generated in a top–down traversal of the binary tree of triangles. The saturation condition [18] simply states that the indicator value of every triangle is larger than or equal to the indicator values of all its subtriangles, i.e.

$$\eta(v_{ref}(T)) \geq \eta(v_{ref}(T_j)), \; j = 1, 2, \text{ for all } T \in \mathcal{T}^l, l < l_{max}.$$

Then, all triangles T are selected whose refinement vertex $v_{ref}(T)$ has an error indicator value $\eta(v_{ref}(T))$ larger than a user–prescribed threshold ε. The union of all triangles on the locally finest levels form a valid triangular mesh. No hanging nodes can occur since, once a given triangle is refined, the triangle sharing the refinement vertex will automatically be refined as well.

There are many ways to define such error indicators, depending on the type of application in mind [8, 15]. Let $h_{\mathcal{T}^l}(x, y)$ be the continuous elevation surface spanned by the vertices of the triangulation $\mathcal{T}^l$. We will here choose the L_1–norm of the one–level look–ahead difference DEM

$$\|h|_{\mathcal{T}^{l+1}} - h|_{\mathcal{T}^l}\|_1 := \int\int |h|_{\mathcal{T}^{l+1}}(x, y) - h|_{\mathcal{T}^l}(x, y)| \; dxdy$$

as a graphical error indicator since it represents the net change of pixel intensities when a given pair of triangles is refined when viewed from above. The error indicator $\eta(v_{ref}(T))$ can easily be computed locally using the height values on the refinement edge of T by the formula

$$\eta(v_{ref}(T)) := 2^{-l} \cdot \left| -\frac{1}{2}h(v_1) + h(v_{ref}) - \frac{1}{2}h(v_3) \right|.$$

The first factor in this formula is just the volume of the pyramidal hierarchical basis function centered at the grid point $v_{ref}(T)$, up to a constant (see [6]). This constant can be neglected since we are just interested in the ordering of the nodes in the tree. The second factor of the above formula is often called wavelet coefficient or hierarchical surplus and corresponds to the second local variation of h.

Note that the error indicator η does not necessarily fulfil the saturation condition. However, a minimal saturated error indicator $\bar{\eta}$ can be constructed in a level–wise bottom–up traversal of the binary tree by the recursive formula

$$\bar{\eta}(v_{ref}(T)) := \max\{\; \eta(v_{ref}(T)), \bar{\eta}(v_{ref}(T_1)), \bar{\eta}(v_{ref}(T_2)) \;\},$$

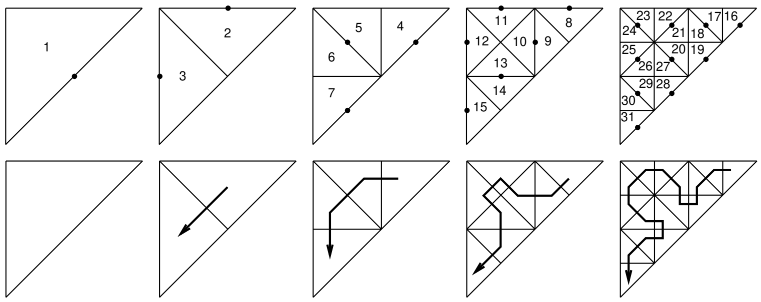

Figure 4: Triangle numbering and corresponding Sierpiński curve.

with $\bar{\eta}(v_{ref}(T)) = \eta(v_{ref}(T))$ on the finest level $l = l_{max}$. Note that a depth–first tree traversal would not be sufficient. This indicator can be efficiently computed and performs very well in practice. Alternatively, it is possible to construct a saturated error indicator η^+ by the recursive formula

$$\eta^+(v_{ref}(T)) := \eta(v_{ref}(T)) + \max\left\{\ \eta^+(v_{ref}(T_1)), \eta^+(v_{ref}(T_2))\ \right\}.$$

This representation allows the computation of robust bounds of the norm used for the computation of η. For example, data bounds for hierarchical isoline extraction or multilevel backface culling can be derived from error indicators of this type [7]. For our numerical computations, however, we will use the minimal saturated error indicator $\bar{\eta}$.

If we would not be concerned with data compression the whole approach would require no special data structures except two arrays, one for the height values and one for the error indicators. The tree traversal could be done by simple index arithmetic. However, in order to employ efficient data compression, we have to take a closer look at the structure of the binary tree. This is subject of the next section.

4 Tree Structure and Space Filling Curves

In the following section, we restrict ourselves to one single triangle with vertices $v_1 = (0,0), v_2 = (0,1)$ and $v_3 = (1,1)$. We define a useful enumeration procedure for the triangles which is closely related to space–filling curves (see [11, 12]). We indicate some basic properties of binary trees which correspond to adaptive triangulations and show how to code the tree structure efficiently.

4.1 Numbering

During the refinement procedure, each triangle T is split into a left and a right subtriangle relative to its refinement vertex $v_{ref}(T)$. The triangles can be numbered recursively: given a number n for a parent triangle, its left subtriangle is assigned number $2n$ and its right subtriangle gets number $2n+1$ if the level of the parent triangle is odd and vice versa if the level is even (see Figure 4). The recursive traversal of the tree of triangles starting with $l = 0$ and $n = 1$ can be sketched in pseudo–code as follows:

```
recursive_descent(Coord v1, v2, v3; Level l, Number n) {
    Coord vref=(v1+v3)/2;
    if (est[vref] < eps) extract_triangle(v1, v2, v3);
    else if (level & 1) {
        recursive_descent(v2, vref, v1, l+1, 2*n);
        recursive_descent(v3, vref, v2, l+1, 2*n+1);
    } else {
        recursive_descent(v3, vref, v2, l+1, 2*n);
        recursive_descent(v2, vref, v1, l+1, 2*n+1);
    }
}
```

Thereby n is the number of the current triangle. Note that this numbering corresponds to the path of the Sierpiński space–filling curve [24, 26] (compare also Figure 4). Following the path of the curve, all triangles on a given level are traversed while any two consecutive triangles share a common edge. This property remains also true for adaptive triangulations over different levels and can e.g. speed up the visualization of triangle meshes since common vertices need not be multiply processed by the graphics engine.

4.2 Boundary triangles

First, we want to identify triangles whose refinement vertex is located on the boundary of the initial triangle. Identification of these triangles is necessary to avoid duplicate storage of vertices lying on the boundaries of the initial triangulation. Boundary vertices fall in two classes. On even levels, they are located on the long diagonal edge (hypotenuse) of the initial triangle, on odd levels on the short horizontal and vertical edges (see Figure 5, left). The bit codes of boundary triangles starts with a 1 (even levels) or with 10 or 11 (odd levels) followed by pairs of 0's or 1's.

This two–level pattern is characteristic for recursive bisection triangulations. In the following proofs, we will always skip one level during induction and step either through the even levels or through the odd ones. Thereby, we have to look at two cases. Let x be the bit code of a triangle. In the first case, if the triangle is located on the boundary, then two of the four child triangles of two levels lower are also located on the boundary. Since the four child triangles form a fan around the refinement vertex of the parent, the bit codes of these triangles are $x00$ and $x11$. In the other case, if the triangle is not located on the boundary, the child triangles won't be located on the boundary as well (see Figure 5, right). The induction starts with the triangles of level 0 and 1 for the even and odd induction, respectively.

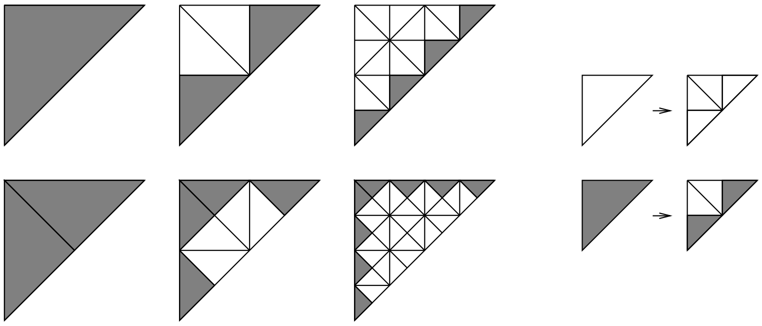

Figure 5: Boundary triangles in even (upper row, left) and odd (lower row, left) levels and respective identification rules (right).

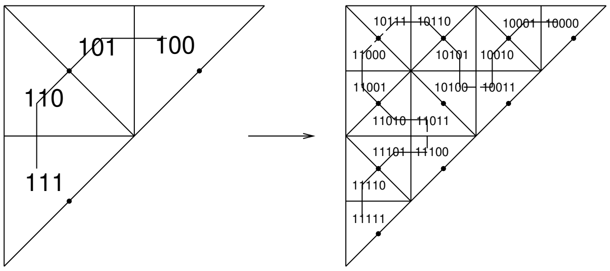

Figure 6: Numbering scheme and triangle fans for the identification of common refinement vertices.

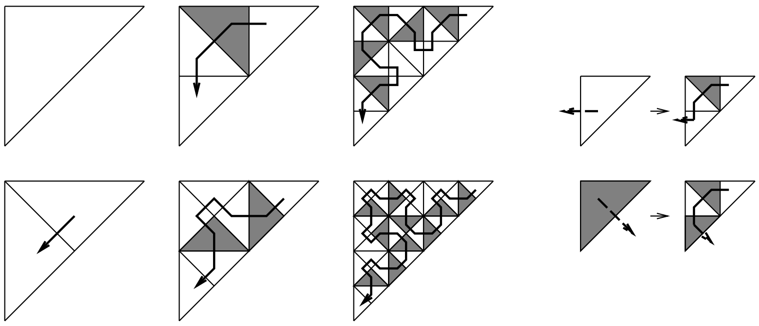

Figure 7: Up–Triangles in even (upper row, left) and odd (lower row, left) levels and respective identification rules (right).

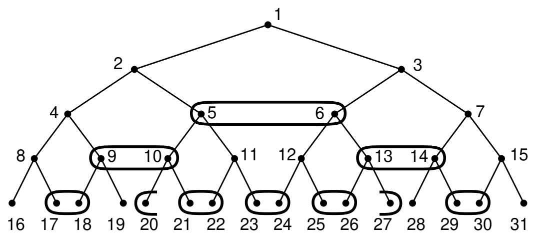

Figure 8: Triangle binary tree with identical refinement vertices tagged.

4.3 Common refinement vertices

Remember that every refinement vertex (except on the boundary) is shared by two triangles and therefore two numbers are assigned to it. Given a number n, the respective other number can be computed by looking at the bit code of n. The transformation can be defined by adding the symbol \$ to the end of the bit code and by an iterative application of the following ordered set of replacement rules:

$$\{\ 00\$ \to \$11,\ 11\$ \to \$00,\ 01\$ \to 10,\ 10\$ \to 01,\ \$ \to .\ \}.$$

This way, starting from the two least significant bits, all bits are inverted in pairs until a 01 or 10 occurs. For example, the opposite triangle (with respect to the refinement edge) of 13 (1101) is 14 (1110) and the opposite triangle of 27 (11011) is 20 (10100) (see Figure 8). No corresponding neighbour is found, i.e. the triangle is located on the boundary, if the last rule is applied or if the fourth rule is applied to the leftmost two bits of the code.

Again, this property can be proved by induction. In Figure 6 (left) we see that in a fan of four triangles the second and third triangle have a common refinement vertex. The other two triangles have their refinement vertex located on the boundary. The necessary transformation would be either to add 1 to the $x01$ triangle or to subtract 1 from the $x10$ triangle. Equivalently, the last two bits just need to be inverted. After two steps of refinement, the fan is split into four fans (Figure 6, right). Here, in addition to the refinement vertices inside each of the four fans, two refinement vertices between fan two and three are common. The first triangle of the second fan ($x0100$) is adjacent to the last triangle of the third fan ($x1011$) and the last triangle of the second fan ($x0111$) is adjacent to the first triangle of the third fan ($x1000$). Again the transformation is to invert bits starting from the least significant bits. The inversion stops when a 01 or 10 occurs.

This transformation could be used for the hash storage (see [9]) of adaptive triangulations. The hash function just has to map identical refinement vertices to the same address, e.g. by using the smaller of the two numbers. Assuming an equal distribution of numbers, the average complexity of this transformation, i.e. the average number of bits to be inverted, is independent of the depth of the tree. The number of bits which are inverted is 2 with probability $\frac{1}{2}$, 4 with probability $\frac{1}{4}$, 6 with probability $\frac{1}{8}$, and so on, resulting in a total number of

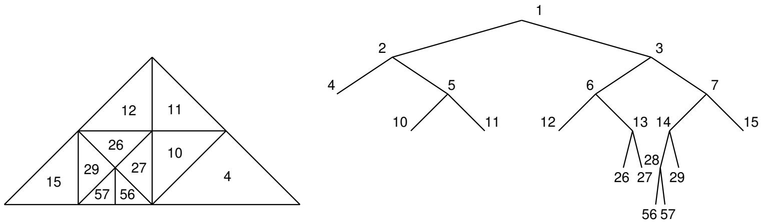

Figure 9: Adaptive triangulation and corresponding binary tree.

$\sum_{l=1}^{l_{max}} 2l \cdot 2^{-l} < 4$. Hashing allows efficient random access to the data, but also requires the storage of hash keys which can require more space than the data itself for large triangulations. For our application, we do not need random access and can therefore apply more efficient schemes, see Section 5.

4.4 Up- and down–triangles

A subtree of the triangle binary tree is defined by a proper subset S of the set of valid triangle numbers $\{1, 2, \ldots, 2^{l_{max}}\}$. Therefore, an adaptive triangulation can be defined by the set $F \subset S$ of all leaves of the subtree. The set F is called the front of the subtree (see Figure 9). If the adaptive triangulation contains no hanging nodes, the underlying binary tree has the following properties:

1. Completeness:
$$\forall\ n \in S, n > 1 \text{ also } \lfloor n/2 \rfloor \in S$$
2. Twins:
$$\forall \text{ even } n \in S \text{ also } n+1 \in S \text{ and } \forall \text{ odd } n \in S \text{ also } n-1 \in S$$
3. Balancing:
$$\forall\ n \in F \text{ either } n+1 \in F \text{ or } \begin{cases} (n+1)/2 \in F, & n \in (0|1)^*01(11)^* \\ (n+1) \cdot 2 \in F, & n \notin (0|1)^*01(11)^* \end{cases}$$

While the first two properties are quite intuitive, the third one needs closer attention. It states that we can traverse the front of the tree from left to right staying on the same level or going one level either up or down. In fact, only two of the three possible cases apply to any given triangle. For one class of triangles (up–triangles) we either stay on the same level or go up one level. Equivalently, the triangle following an up–triangle along the space–filling curve may either be larger than the up–triangle or be of the same size. Consequently, in up–triangles the space–filling curve leaves through the hypotenuse of the triangle. Up–triangles can again be identified by looking at the bit code of their number: their bit code ends with 01 or contains 01 followed by an even number of 1s.

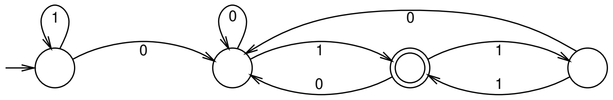

Figure 10: Automaton recognizing up triangles.

The other class of triangles where we either stay on the same level or go down one level (down–triangles) are identified by the respective complement.

Let us now prove this property. In Figure 7 (left) all up–triangles are grey–shaded. Not surprisingly, the pattern can again be defined recursively. In the first case, the second triangle in a triangle fan is always an up–triangle. In the second case, up–triangles also occur as the fourth triangle of a fan, but only if the parent triangle is also an up–triangle. Figure 7 summarizes these two cases by definition of the respective creation rules. Induction again runs over the even and odd levels, respectively. These creation rules can easily be converted into bit codes. This way, up–triangles occur if their bit code ends with 01 (case 1) or ends with 11 (case 2) preceded by 01. In summary, this corresponds to the language $(0|1)^*01(11)^*$.

Note that only approximately 1/3 of all triangles are up–triangles. It is possible to recognize up- or down–triangles during the tree traversal by the four–state automaton of Figure 10. The automaton is initialized to the leftmost state for the inital triangle and is advanced one state for each refinement step (0 for the first and 1 for the second child). If the automaton is in the double–circled state, the current triangle is an up–triangle, otherwise it is a down–triangle.

4.5 Tree and Tree–Front Coding

Now, due to properties 1 and 3 it is possible to code an adaptive hierarchical triangulation using one bit per triangle in the tree front. We code a stay on the same level by 0 and a change of level by 1. In order to start correctly, it is also necessary to store the level of the leftmost triangle. This way, the numbers of the triangles in the tree front F can be computed successively using the number of the current triangle and the next bit in the code. This is done in a single preorder traversal of the tree. For example, the tree–front code for the triangulation in Figure 9 is

2 100101011.

Since the number of triangles in an adaptive triangulation is about twice the number of grid points, approximately two bits per grid point are needed to store the tree.

Alternatively, we can use properties 1 and 2 to code the local tree topology using the lookup table of Figure 11. For a preorder traversal of the binary tree, the tree code for the triangulation in Figure 9 is

11 01 00 11 01 00 10 10 00.

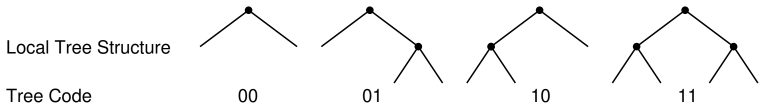

Figure 11: Lookup table for the tree code.

Note that in this way approximately four bits per grid point are needed for the storage of the triangulation.

In either case much less memory is used for the storage of a given triangulation in comparison to more explicit schemes, e.g. Delaunay triangulations. These models need the storage of coordinates which usually require several bytes per grid point. The tree–front code can be used for the storage of DEMs on background media or for applications, such as image compression, which only require the processing of the whole DEM. For the visualization application we need to use the tree code since in an adaptive tree traversal only a very small subset of the whole data is processed for each image.

5 Compression

We will now construct a fine resolution triangulation by choice of a threshold value ε_0 and selection of all triangles T with error indicator $\bar{\eta}(v_{ref}(T)) > \varepsilon_0$. The remaining triangles, respectively the height values at their refinement vertices, are not stored since the height values can be exactly or nearly (up to ε_0) interpolated from their hierarchical neighbours. If ε_0 is set to 0, the compression is lossless, i.e. the original DEM can be reconstructed without error. If ε_0 is chosen such that the error of the resulting approximation is smaller than the data error of the input, the compression is usually also considered lossless. Larger ε_0, however, will result in a lossy compression.

In this section we demonstrate how a multiresolution DEM defined over such a triangulation can be efficiently stored and how the compressed data can be processed. This allows the efficient handling of very large data sets without partial uncompression of the data from disk or other background storage media. To be able to work on compressed data is an important prerequisite for real–time applications. The concepts are similar to pointer–less (e.g. so–called linear) quadtrees [14], but for triangulations the relation triangles – vertices (see Section 3.1) requires special care.

5.1 Node Stack

Let us first be concerned with the storage and the processing of the single DEM defined over an adaptive (fine resolution) triangulation. As seen before, it is possible to store the DEM using two one–dimensional arrays, one consisting of the height values in the order they appear in the tree traversal and one consisting of a code of the underlying tree. The size of these arrays is determined by the number of vertices in the triangulation. The location of each array element in the binary tree can be computed based on the code of the tree during the tree traversal. However, since all interior vertices appear twice during the traversal, special care has to be taken to avoid double storage of height values. This can be done in several ways:

Clearly, vertices have to be stored only at their first occurrence. One possibility to achieve this would be to simply mark already processed vertices which requires, however, additional memory. Another solution would be to apply the replacement rules of Section 4.3 and to store only nodes whose number is smaller than the number of the opposite triangle. Alternatively, vertices at their second occurrence are identified by the language $(0|1)^*10(00|11)^*$ and can be recognized by the automaton parsing this language. The construction and proof can be conducted in complete analogy to Section 4.4.

When working with the compressed data, it is possible to recognize vertices which have been stored earlier during the traversal, however, their location in memory is not known. One idea how to solve this problem is to put vertices which will be processed again later onto a stack and to remove them after their second usage. It turns out that this is indeed possible (compare [11]), but several stacks are needed, one for each level. The triangle binary tree can be traversed with the following actions being taken for a given triangle n of level l:

$$n \in \begin{cases} (0|1)^*01(00|11)^* & : \text{ read vertex and push onto stack } l, \\ (0|1)^*10(00|11)^* & : \text{ pop vertex from stack } l, \\ \text{otherwise} & : \text{ read vertex (located on boundary).} \end{cases}$$

Of course, the three different cases can also be identified using the transformation of Section 4.3.

An upper bound for the size of a stack for level l is $2^{\lfloor l/2 \rfloor}$ which is attained for regular triangulations. Therefore the overall memory overhead is at most of order $O(\sqrt{N})$. Note that the size actually needed for the stacks for a given triangulation can be much smaller, up to $O(1)$ for highly adaptive triangulations.

A simple pyramidal multiresolution DEM could consist of several DEMs corresponding to threshold values $\varepsilon_0, 2\varepsilon_0, 4\varepsilon_0, \ldots$, up to a maximum threshold. The multiresolution DEM could be stored using the tree–front codes for all the different thresholds and the height values only for ε_0. However, for our visualization application we need a tree–structured model. The necessary additional information is determined in the next section.

5.2 Branch Pointers

Now, we know how to code and compress an adaptive triangulation and how to traverse the complete (fine resolution) triangulation. In many applications such as visualization, it is necessary to be able to traverse it selectively, i.e. to extract a subtree of the adaptive binary tree. Since the vertices are arranged in one large array, we would have to know the number of vertices which are skipped if the traversal is stopped locally.

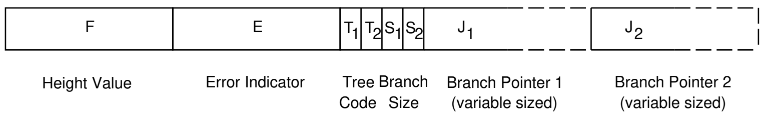

Figure 12: Data type used for adaptive hierarchical triangulations.

One idea to solve this problem is by the additional storage of relative branch pointers. For each node in the tree this pointer is exactly the number of nodes in the subtree below. Thereby the expensive storage of absolute pointers is avoided, which would usually require more memory than the data itself. Some of the relative pointers can get quite large, i.e. the topmost node will contain number $N-1$. But, on most of the nodes the numbers are small, i.e. on the locally finest levels of the adaptive triangulation they are 0 and on the second finest levels they are at most 2.

Therefore we allocate variable sized memory for the pointers and use a bit code for the allocated size. Here we will use 2 bits for the size, i.e. 00 for 1 byte, 01 for 2 bytes, 10 for 4 bytes and 11 for 8 bytes. Note that for a tree code 00 the branch pointer is also 0 and therefore does not need to be stored. Of course, since we use variable sizes, the branch pointer is now not the number of triangles in the subtree, but the number of bytes allocated for all the nodes in the subtree. It can be computed in a bottom–up traversal of the tree. This way, the average storage requirement for a branch pointer is less than one byte independent of the depth of the binary tree.

Since each node has two subtrees (remember the duality triangles – vertices) we are led to the data type of Figure 12. The data values T_1, S_1 and J_1 correspond to the first occurrence of the vertex in the space–filling curve and the data with subscript 2 correspond to the second. For boundary vertices, the latter entries are empty. Remember that the tree code and the branch pointer size require each 2 bits for their storage. Therefore, T_1, T_2, S_1 and S_2 together require exactly one byte. This way, on average only approximately 3 bytes per vertex are needed for the coding of the complete tree structure.

This scheme allows a very fast and efficient adaptive tree traversal since only a few bit operations are necessary to find subsequent data values.

6 Visualization of Global Data

Now, our main application will be the interactive visualization of DEMs using color shaded triangles. However, the amount of data to be visualized can easily exceed the amount of triangles that can be drawn interactively by several orders of magnitude. This problem can be resolved by multiresolution models making two observations. First, areas that are not visible, i.e. outside the viewing window or smaller than a screen pixel, should not be processed. Second, less interesting (e.g. flat) areas may be approximated by larger triangles. This way, the overall number of triangles to be drawn can be reduced greatly thus allowing

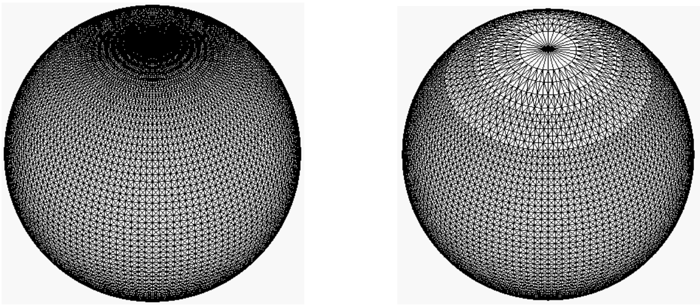

Figure 13: Compensation of pole singularities by the error indicator.

real–time visualization.

6.1 Spherical Mapping

In order to construct a hierarchical triangulation for global topographic data it is necessary to cover the unit sphere with triangles. Here we use the well–known transformation from geographic coordinates $(\phi, \psi) \in \Omega = [0, 2\pi] \times [-\pi/2, \pi/2]$ (with longitude ϕ and latitude ψ) into spherical coordinates $(u, v, w) \in \mathbb{R}^3$ by the formula

$$\begin{pmatrix} u \\ v \\ w \end{pmatrix} = (1 + h(\phi, \psi)) \cdot \begin{pmatrix} \cos\phi \cdot \cos\psi \\ -\sin\phi \cdot \cos\psi \\ \sin\psi \end{pmatrix},$$

where $h(\phi, \psi)$ is the normalized height value at the given coordinate (see Figure 13, left). The rectangle Ω is split into two squares with side length π. This way, the coarsest triangulation consists of four triangles each covering a vertical 90 degree slice on the sphere. Note that for the computation of the error indicator values at $\phi = 0$ and 2π have to be updated consistently due to periodicity.

We chose geographic coordinates despite the occurrence of singularities at the poles due to several reasons: First, they allow very intuitive handling of data sets, since the user is accustomed to the coordinate system. Second, coordinate conversion is very fast, especially if the occurring sine and cosine values are precomputed. Third, the singularities at the poles can be compensated for if the error indicator values are multiplicated (prior to saturation) with $\cos\psi$. This scaling factor relates the area of the planar triangle to the area of the spherical triangle varying with ψ. The latter will rapidly become smaller towards the poles. Figure 13 (right) shows this compensation effect.

Of course, it also possible to construct hierarchical triangulations for other sphere approximations, e.g. based on one of the five Platonic solids or the soccer ball tesselation [3]. The only requirements are a valid initial triangulation and the matching of refinement edges of adjacent triangles.

6.2 Visualization

The main ingredient of the drawing algorithm is a top–down traversal of the binary tree of triangles. The traversal is stopped locally if one of the following conditions holds: if the current triangle T is completely outside the viewing window, nothing is drawn. Otherwise, if the error indicator value of the current triangle $\bar{\eta}(v_{ref}(T))$ is below a drawing threshold $\varepsilon_1 \geq \varepsilon_0$, the triangle is drawn. Since we are mainly interested in a top view of the DEM, the first check can be easily done by coordinate clipping.

The drawing threshold ε_1 could be selected by the user. However, if the user zooms into or out of the data set, the drawing threshold should change in order to maintain the current frame rate and the image quality. In order to avoid a continuous readjustment of the threshold by the user, the drawing threshold has to be related to the current zoom factor appropriately. Let us therefore define the zoom factor z as the ratio of the current scale to the global scale. As can easily be seen, the correct way to relate the drawing threshold ε_1 to the zoom factor is given by the formula $\varepsilon_1 = \varepsilon_r \cdot z^{-5/4}$. Here, ε_r is a global reference threshold. By adjusting ε_r instead of ε_1, the user is able to control the overall image quality and the amount of triangles being drawn can be held constant independent of the current zoom factor z.

The error indicator concept can also be used by applications requiring view–dependent level–of–detail. Thereby, areas which are far away from the current viewpoint are considered less important than those close by. View–dependent LOD can be achieved by an appropriate choice of a continuous distance function $\chi(\bar{\eta}, \varepsilon_1)$ as a stopping criterion. The technique is illustrated in detail in [18].

7 Numerical Results

We will now show performance results of our algorithm concerning compression and visualization of the *gtopo30*[1] dataset. The computations were done on an SGI Onyx[2] R10000 (195 MHz). The data set was interpolated onto two 8193×8193 grids resulting in over 134 Mio. elevation values. Each elevation value represents the height value in meters above sea level and is stored using 2 bytes as a signed integer (remember that some areas on earth are several hundred meters below sea level). Ocean areas are marked by *nodata* values.

For the computation of the error indicator at coastlines we set ocean height values artificially to -1000 meters. This way, coastlines are given a higher importance than inland areas which greatly improves the overall visual impression since the eye is particularly sensitive to edges. We also use 2 bytes for the error indicator $\bar{\eta}$ which enables us to extract up to 65536 approximations with different resolutions. The indicator values are scaled such that the lowest possible resolution contains about 500 triangles.

[1] Courtesy of US Geological Survey

ε_0	vertices	nodes in tree	size (bytes) on disk	size (bytes) in memory	rel. error
0	134.250.498	268.435.454	302.055.446	809.697.258	0.0%
2	40.726.357	81.433.161	91.631.895	248.751.126	0.004%
4	30.971.462	61.924.693	69.683.546	190.170.314	0.020%
8	23.299.051	46.580.830	52.420.742	144.005.944	0.064%
16	15.578.706	31.141.001	35.050.073	97.012.493	0.220%
32	8.886.835	17.757.981	19.993.453	55.750.190	0.659%
64	4.468.305	8.921.476	10.051.830	28.175.287	1.513%
128	2.150.101	4.285.449	4.835.918	13.605.848	2.500%
256	1.042.534	2.072.666	2.344.186	6.605.564	3.677%
512	404.078	801.818	908.419	2.561.195	5.153%
1024	156.618	309.955	352.015	992.912	7.023%
2048	61.212	120.714	137.548	388.057	9.500%
4096	23.808	46.984	53.489	148.810	12.968%
8192	9.353	18.209	21.017	59.120	18.315%
16384	3.579	6.887	8.055	22.561	26.649%
32768	1.387	2.613	3.136	8.670	40.761%
65535	502	921	1.154	3.113	60.395%

Table 1: Compression results for the *gtopo30* dataset.

7.1 Compression

The compression results are given in Table 1. For different threshold values ε_0 we compare the number of stored elevation values, the number of triangles in the corresponding triangulation, the number of interior vertices in the binary tree, the relative approximation error $\|h - h_{\varepsilon_0}\|_1/\|h\|_1$, and the memory requirements using the data types of the previous section. Note that additional temporary memory is required for the different stacks needed for the tree traversal, but the amount is negligible.

We see that the number of vertices roughly halves when ε_0 is doubled. This agrees well with asymptotic approximation theory for L_p–type indicators for smooth functions. The number of nodes in the tree corresponds to the algorithmic overhead imposed by the tree traversal. As expected, it is at most twice the number of vertices independent of the threshold value.

Compression results are shown in column four and five. On–disk compression was done using the tree–front code of Section 4.5. Thereby besides the two bytes for the elevation values, approximately two bits per vertex for the storage of the tree are required. The bitstream could be further compressed in a postprocessing step using a lossless data compression method such as Lempel–Ziv or Huffman coding. Experimental results then indicate another gain of a factor of two in the compression ratio. A replacement of the elevation values with their respective wavelet coefficients will not increase the compression ratio significantly here. The in–memory storage requirements using the data type of Section 5.2 is less than seven times the number of vertices. This means that less than three bytes per node are needed on average for the complete tree structure. For a threshold of 0, the memory requirements would be larger than for the simple two–dimensional full array (0.25 GByte). But, already for a threshold of 2 the total memory requirement is less than that required for the original DEM.

The relative approximation error is measured in the L_1–norm in correspondence to the type of error indicator. The approximation error does not depend linearly on ε_0. It increases faster than ε_0 for small values and slower for large values. For a relative error of 5% only about 0.3% of the number of vertices are needed.

7.2 Visualization

Figure 14 shows a subset of the whole data set which is visualized in an interactive application. The triangles are color shaded using a simple geographical colormap. Ocean areas are not drawn, instead a texture containing a sphere is displayed. All height values are exaggerated by a factor of 10. We achieved a drawing rate of 500000 triangles per second independent on the current position and zoom factor. Moderate quality images can be produced at a high frame rate allowing interactive flyovers while high quality images for closer inspection take at most a few seconds.

7.3 Zooming through several data sets

Due to the tree structure of the multiresolution model it is possible to add local data sets at lower scales. This is simply done by the construction of a multiresolution model for the new data set and by replacing a subtree of the original one with the new one. In order to conserve a valid triangulation, possibly some points near the boundary of the new data set, which interpolate the missing height values, have to be inserted in the tree. The number of those points is determined by the local smoothness in the transition zone and the difference in resolution of the data sets. Furthermore, some of the error indicator values might need adjustment to satisfy the saturation condition.

We added two nested data sets to our global model. The first one covers the lower Rhine valley and its surroundings[2] while the second one covers the city of Bonn[3]. Height values are represented in decimeters and centimeters, respectively. Each data set was interpolated onto a 8193×8193 grid covering the next larger dyadic box in geographic coordinates.

Figure 15 shows that the data sets inserted this way fit smoothly into the global model. All three data sets were compressed with a relatively small $\varepsilon_0 = 2$ resulting in a total memory requirement of less than 0.5 GByte. The global reference threshold ε_r was set to 1000. This way, the number of triangles drawn are 83491, 93967, 49323, and 97353 and therefore differ by at most a factor of 2. Note that different colormaps were used for the images in order to improve the visual impression.

[2]Courtesy of SFB 350, University of Bonn
[3]Courtesy of DLR, Köln–Porz

8 Concluding Remarks

In this paper, we have shown that interactive handling of large–scale topographic data is possible with a system having three interwoven constituents: hierarchical structuring, adaptive refinement and compression. The system presented here specifically uses recursive bisection triangulations as the underlying hierarchical structure, saturated error indicators for adaptive tree traversal and tree codes as well as relative branch pointers for efficient in–memory and on–disk storage. Thereby, we have shown that these compressed adaptive hierarchical triangulations allow an interactive visualization of very large DEMs.

We have considered lossy as well as lossless compression of adaptive hierarchical triangulations by local omission and interpolation of insignificant elevation values. Let us emphasize once more that in our scheme the number of bits per grid point needed for the storage of an adaptive triangulation is independent of the total number of triangles or the maximum level of the binary tree. Thereby, the stored elevation values were represented exactly. Further lossy compression could be obtained by quantization of the hierarchical offset and error indicator values.

Furthermore, we have shown that data sets can be quickly added and removed from the model. This is an important prerequisite for interactive global geographic information systems. The tree structure and space–filling curve property could be used for parallelization of the program employing dynamic load balancing (see [7, 9]) which would allow the handling of still larger data sets.

Note that the proposed compression scheme applies not only to the visualization of large data sets. With appropriate modifications it can be used for a variety of further applications such as terrain analysis, feature extraction, the computation of drainage networks or the solution of partial differential and integral equations.

References

[1] De Floriani, L., Puppo, E., Magillo, P.: Geometric Structures and Algorithms for Geographic Information Systems, in J.R. Sack, J. Urrutia (eds.), Handbook of Compuatational Geometry, pp. 333-388, 1999.

[2] Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C., Mineev–Weinstein, M.B.: ROAMing Terrain: Real–time Optimally Adapting Meshes, in *Proc. Visualization '97*, 1997.

[3] Dutton, G.H.: A Hierarchical Coordinate System for Geoprocessing and Cartography, Lecture Notes in Earth Sciences 79, Springer, 1999.

[4] Evans, W., Kirkpatrick, D., Townsend, G.: Right Triangular Irregular Networks, *Technical Report* 97–09, University of Arizona, 1997.

[5] Gerstner, T.: Ein adaptives hierarchisches Verfahren zur Approximation und effizienten Visualisierung von Funktionen und seine Anwendung auf digitale 3–D Höhenmodelle, *Master's thesis*, Institut für Informatik, TU München, 1995.

[6] Gerstner, T.: Adaptive Hierarchical Methods for Landscape Representation and Analysis, in *Process Modelling and Landform Evolution*, S. Hergarten, H.–J. Neugebauer (eds.), Lecture Notes in Earth Sciences 78, Springer, 1998.

[7] Gerstner, T., Rumpf, M.: Multiresolutional Parallel Isosurface Extraction based on Tetrahedral Bisection, in *Proc. VolVis99*, 1999, to appear.

[8] Gerstner, T., Rumpf, M. and Weikard, U.: A Comparison of Error Indicators on Nested Grids for Multilevel Visualization, in Data Visualization '99, E. Gröller, H. Löffelmann, W. Ribarsky (eds.), Springer, 1999.

[9] Griebel, M., Zumbusch. G.: Parallel Multigrid in an Adaptive PDE Solver based on Hashing and Space-Filling Curves, Parallel Computing 25, pp. 827–843, 1999.

[10] Gross, M.H., Staadt, O.G. and Gatti, R.: Efficient Triangular Surface Approximations using Wavelets and Quadtree Data Structures, *IEEE Trans. on Visualization and Computer Graphics 2(2)*, 1996.

[11] Hebert, D.J.: Cyclic Interlaced Quadtree Algorithms for Quincunx Multiresolution, *J. Algorithms* 27:97–129, 1998.

[12] Hebert, D.J., Kim, H.-J.: Image encoding with triangulation wavelets, in Wavelet Applications in Signal and Image Processing III, A.F. Laine, M.A. Unser, M.V. Wickerhauser (eds.), pp. 381–392, 1995.

[13] Heckbert, P.S. and Garland, M.: Survey of Surface Approximation Algorithms, *Carnegie Mellon University Technical Report* CMU–CS–97–, 1997, to appear.

[14] Lee, M., Samet, H.: Navigating through Triangle Meshes Implemented as Linear Quadtrees, report CAR–TR–887, Computer Science Department, University of Maryland, 1998.

[15] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L.F., Faust, N. and Turner, G.: Real–Time, Continuous Level of Detail Rendering of Height Fields, *Computer Graphics (Proc. SIGGRAPH '96)*, 1996.

[16] Maubach, J.: Local bisection refinement for n-simplicial grids generated by reflection, *SIAM J. Sci. Comput.* 16, pp. 210–227, 1995.

[17] Mitchell, W.F.: Adaptive refinement for arbitrary finite element spaces with hierarchical bases, *J. Comp. Appl. Math. 36*, pp. 65-78, 1991.

[18] Ohlberger, M., Rumpf, M.: Adaptive Projection Methods in Multiresolutional Scientific Visualization, *IEEE Transactions on Visualization and Computer Graphics* 4, 1998.

[19] Pajarola, R.: Large scale Terrain Visualization using the Restricted Quadtree Triangulation, in *Proc. IEEE Visualization '98*, pp. 19–24, IEEE Computer Society Press, 1998.

[20] Paul, A., Dobler, K.: Adaptive Realtime Terrain Triangulation, in *Proc. WSCG '97*, University of West Bohemia, Plzen, Czech Republic, 1997.

[21] Puppo, E., Scopigno, R.: Simplification, LOD and Multiresolution Principles and Applications. *Eurographics '97 Tutorial Notes*, Eurographics Association, 1997.

[22] Röttger, S., Heidrich, W., Slussallek, P., Seidel, H-P.: Real–Time Generation of Continuous Levels of Detail for Height Fields, Proc. 6th Int. Conf. in Central Europe on Computer Graphics and Visualization, pp. 315–322, 1998.

[23] Rivara, M.C.: Algorithms for Refining Triangular Grids Suitable for Adaptive and Multigrid Techniques, *International Journal for Numerical Methods in Engineering* 20, pp. 745–756, 1984.

[24] Sagan, H.: Space–filling Curves, Springer, 1994.

[25] Samet, H.: Data Structures For Quadtree Approximation And Compression, *Comm. ACM 28*, pp. 973-993, 1985.

[26] Sierpiński, W.: Sur une nouvelle courbe continue qui remplit toute une aire plane, *Bull. Acad. Sci. de Cracovie*, pp. 462–478, 1912.

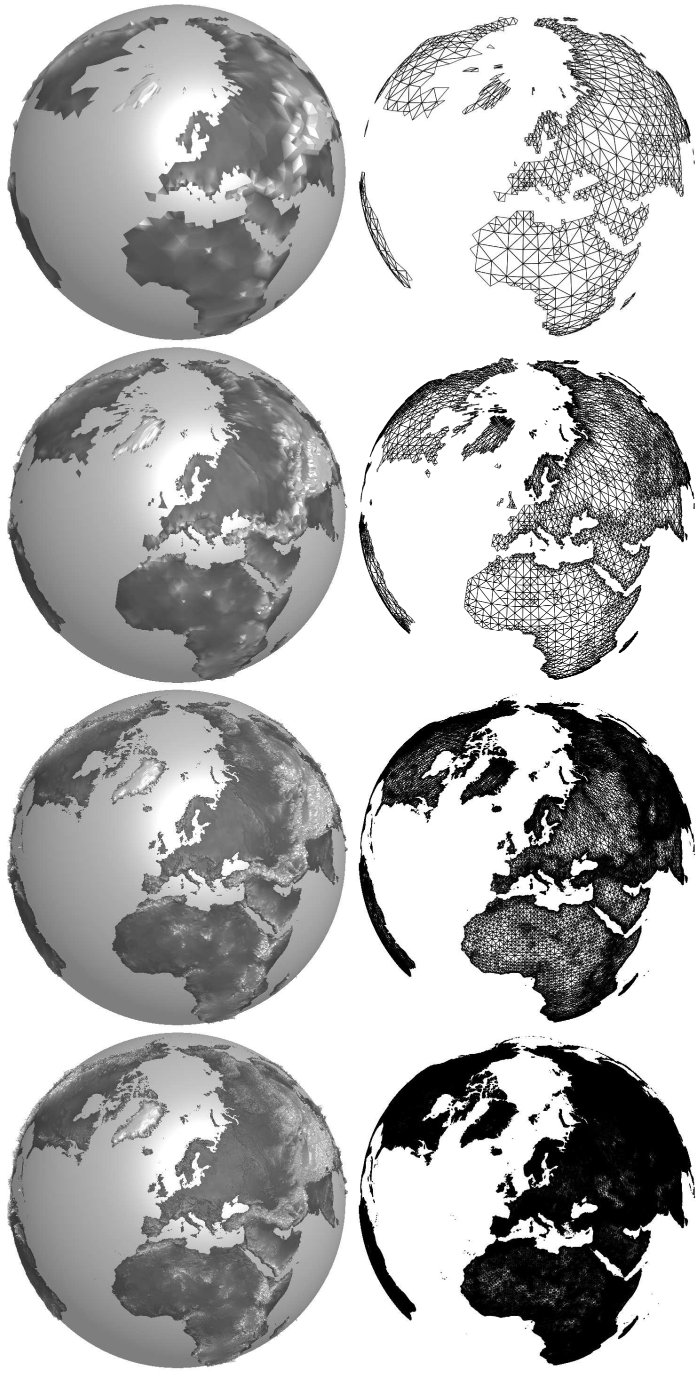

Figure 14: DEMs corresponding to ε_0 values of 16384, 4096, 1024, and 256.

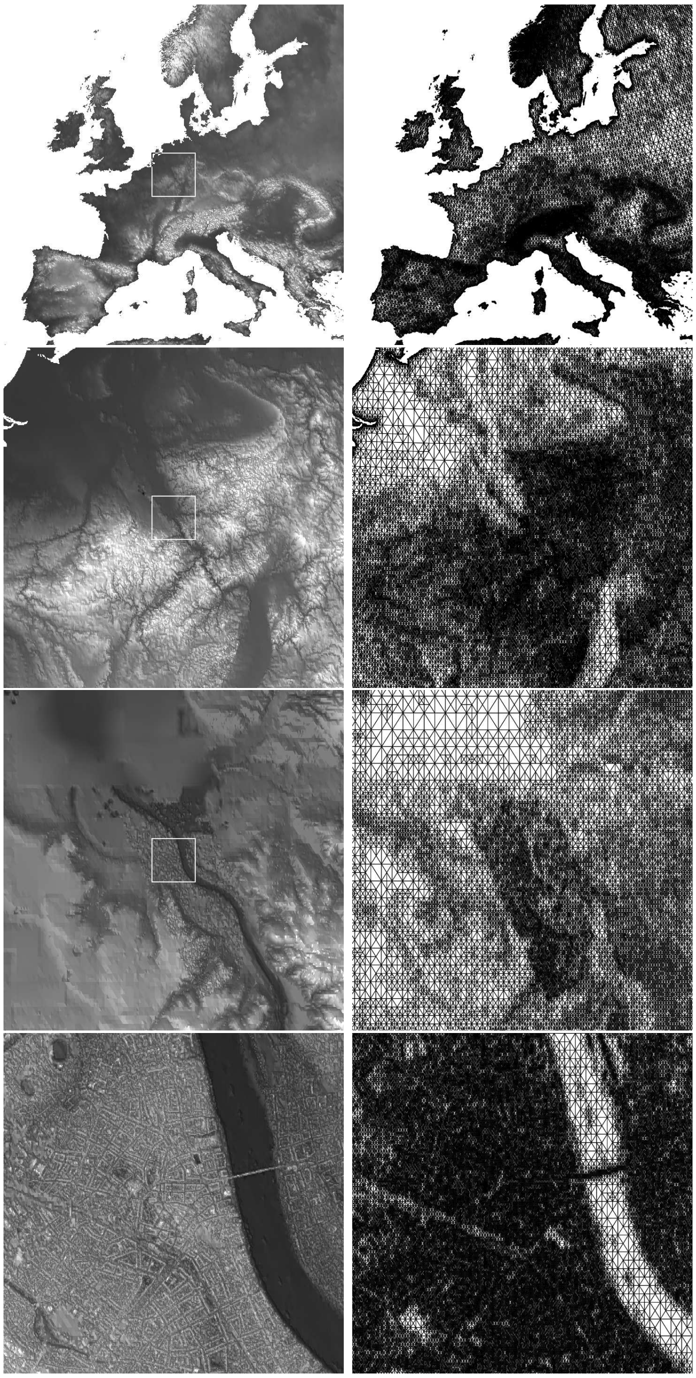

Figure 15: DEMs corresponding to zoom factors of 4, 32, 512, and 4096.