Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Theoretische Informatik

Renato Pajarola

Large scale Terrain Visualization using the Restricted Quadtree Triangulation

February 1998

292

Large scale Terrain Visualization using the Restricted Quadtree Triangulation

Renato Pajarola

Institute of Theoretical Computer Science
ETH Zürich, Switzerland
pajarola@inf.ethz.ch

Abstract

Real-time rendering of triangulated surfaces has attracted growing interest in the last few years. However, interactive visualization of very large scale terrain data is still a problem for virtual reality systems. The graphics load has to be controlled by an adaptive surface triangulation and by taking advantage of different levels of detail. Furthermore, the dynamic management of the visible scene requires efficient access to the terrain database. All components must interact smoothly and efficiently to arrive at a high-performance visualization system. We describe triangulation, scene management and data handling that overcome all problems. The triangulation model is based on the restricted quadtree triangulation and extended to meet the different requirements. Moreover, we present new details of the restricted quadtree triangulation not mentioned previously in the literature. These include among others exact error calculation, progressive meshing, performance enhancements and spatial access.

Index terms visualization, multiresolution triangulation, terrain database, geoinformation system

1. Introduction

Interactive visualization of very large scale terrain data brings up a wealth of problems. Most of the problem aspects are subject to current research, and optimal solutions for specific problem contexts exist. In spite of that, the tailoring of the best suitable approach to the conditions of the superior problem, and the integration of the different solutions to form a practical system is still a complex task. A deep understanding of the relevant concepts from the different fields is essential to come up with a viable solution.

The fundamental problems in computer graphics for interactively displaying a large, triangulated surface are rendering efficiency and data management. The best way to take full advantage of a given rendering performance is to reduce the complexity and number of geometries used to display a scene as much as possible. However, the simplification of the scene complexity, or the reduction of geometric primitives should not lead to an inferior visual representation on screen. Therefore, the simplification must be controlled by an error measure. This means that simplification is only performed as long as the approximation error is below a specified threshold. Another way to increase rendering efficiency is the use of the *level of detail* (LOD) concept. This concept corresponds nicely to the human's visual system which can perceive objects at different resolutions based on distance and angle of view. Accordingly, instead of showing everything in full detail, objects or parts of a scene are displayed in lower resolutions – with higher approximation errors – the farther away they are from the viewpoint and view-direction.

Large-scale terrain databases are usually too large to be displayed as a whole, even when using multiple LODs. For instance, the digital elevation model of Switzerland at 25 meter grid-resolution consists of more than 60 million points, or over 120 million triangles. Therefore, only a fraction of such an extensive model can be rendered at an interactive frame-rate. This partial scene, however, must be updated dynamically according to changes of the viewpoint and view-direction. Maintaining such a triangulated scene which dynamically changes with time involves a *dynamic scene update* mechanism. This mechanism refers to the periodic reloading and discarding of partial regions of the visible scene. A *scene manager* has to decide upon certain parameters when and which parts of a scene will be discarded, updated (refined) or newly loaded from the database.

Therefore, the database or data structure holding the terrain data must support spatial access. Most dynamical scene updates involve spatial range queries on the terrain database to reload new or update currently visible parts of the scene. Moreover, an approximation error parameter which specifies the correct LOD for the requested region is attached to each spatial range query. The database must not only be able to extract a given region at variable resolutions, it must as well accept LOD-range selection to support incremental refinement of partially loaded regions. Different resolutions of the triangulated terrain should be maintained without much redundancy of data, and the representation of the topology of the triangulation must be as compact as possible. Efficient storage and compact topology representation is not only required to realize a small database, it is also essential for fast network transmission.

The triangulation model is the core structure for every terrain surface visualization system. It must carefully be adapted and integrated into the application environment. In order to provide a concise representation, the triangulation should only use as many triangles as absolutely necessary. To do this without negative impact on the accuracy of the surface approximation, the triangulation must be adaptive to the terrain structure. This means that high frequency elevation changes in the surface are modeled with more triangles per area unit than low frequency surface regions. Furthermore, this triangulation model must also provide means to extract surface representations at variable precisions. This is needed to enable visualization using multiple LODs. Such a model is also called a multiresolution triangulation.

Summing up, to arrive at an efficient visualization of large, triangulated terrain surfaces, several requirements have to be fulfilled. The following list recalls briefly the demands for the intended visualization system, and in particularly for the core triangulation model. The list is not sorted by priority as there is no clear distinction between more or less significant requirements, it is rather sorted by related topics.

1. fast access to large scene database
2. effective database storage management
3. variable resolution database access
4. terse topology representation
5. quick adaptive triangulation
6. dynamic visible scene management
7. multiresolution visualization
8. high-performance geometry rendering
9. continuous level of detail rendering

The following sections will first introduce a hierarchical multiresolution triangulation model which we modified to comply with our complex problem requirements. Then, for the different problems, solutions and their integration with the triangulation model are presented. The chapter ends with a comparison to related work.

2. Restricted quadtree triangulation

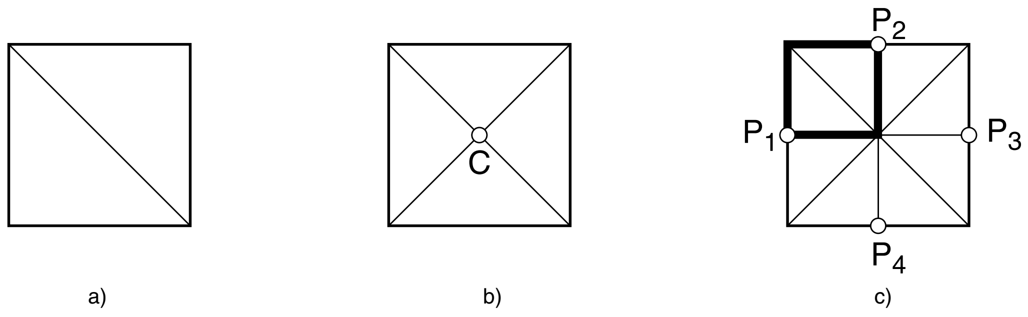
The *restricted quadtree triangulation* (RQT) is an adaptive, hierarchical triangulation for height fields. A height field is a matrix of points that are distributed regularly on a two-dimensional grid with constant distances in both dimension between the grid lines. The hierarchical decomposition of space of the restricted quadtree is derived from the quadtree data structure in [15]. However, an extra restriction is added such that adjacent quadrants, or quadtree blocks, may only differ by one level in the quadtree hierarchy. In [20] the restricted quadtree was used to triangulate a parametric surface. They decomposed the parameter space using a restricted quadtree with appropriate subdivision criteria. Consequently, the triangulation of the surface was done using only the previously selected points in the parameter space. However, given an unrestricted quadtree which satisfies the approximation tolerance, no algorithm was provided in [20] to efficiently build the restricted one from a plain quadtree. In addition to the algorithms presented in [16], we will provide two different algorithms to build a restricted quadtree, one theoretically optimal in the sense of [14] and an implementation-efficient one.

When using the restricted quadtree for terrain triangulation, the quadtree decomposition can directly be applied to the height field data in the object space, instead of using an intermediate parameter space. In [11] a screen-space error function was used as subdivision criterion for the quadtree decomposition. We propose an error measure in the object space to exactly control the accuracy of the surface approximation. Furthermore, in contrast to [20] and [11] we do not retriangulate the surface for each rendered frame using the restricted quadtree. In our model, the surface triangulation is recomputed only at irregular intervals depending on an approximation deviation threshold. Whenever the deviation of the current approximation accuracy from a specified value exceeds a certain threshold, the surface is retriangulated. See also Section 5 for more details.

2.1. Quadtree hierarchy and triangulation

Unlike in the original quadtree, the subdivision of the restricted quadtree has two stages because it is tightly coupled with the triangulation. Figure 1 shows the two stages of the recursive subdivision of a quadtree block, and at the same time the hierarchical triangulation. The basic quadrant consists of four points, or vertices and two triangles as shown in Figure 1 a), or its 90° rotated equivalent. The intermediate stage of the refinement is achieved by adding the mid-point C of the quadtree block. This step splits the initial two triangles into four at that same vertex C in the middle of their base lines, see Figure 1 b). The hierarchical refinement is completed by adding the middle points of the quadrant's boundary edges and splitting the triangles accordingly. Note that two new vertices on adjacent edges, i.e. P_1 and P_2 of Figure 1 c), together with a corner- and the mid-point C form a new basic quadtree block. Note that all new vertices lie on the next higher resolution of the grid with halved distance between grid lines. This two stage scheme will be used later in this section to show how to prevent cracks when constructing the triangulated surface.

FIGURE 1. Restricted quadtree subdivision



The assignment of points on the grid to levels in the quadtree hierarchy is shown in Figure 2. The top level in the hierarchy is level 0, and it denotes the root of the quadtree. Given the dimension $n = 2^k + 1$ of the quadratic grid and all the points $P_{i,j}; i, j = 0..n-1$, then the points on lower levels $l \geq 0$ are given by $L_l = \{P_{i,j} | i, j = h_l d_l \wedge P_{i,j} \notin L_{l-1}\}$. Level dependent is the factor $h_l = 0..2^l$, and the distance d_l in grid-units between points on level *l*, which is $d_l = (n-1)/2^l$. By the center-vertices L_l^{center} we denote the points of L_l which are located in the center of a quadtree block on level *l-1*. In Figure 2 the center-vertices are drawn as *white-filled* circles. Let $\bigcup L_l = \bigcup_{k=0}^{l} L_k$ be all points down to level *l*. Note that in an unbalanced restricted quadtree *Q* the particular set $L_l(Q)$ does not have to include all points of L_l but only a subset of it. The indices *i* and *j* of vertices on level *l* can be generated in lexicographical ordering in turn to visit all points of one level. Also separately the vertices of L_l^{center}. Moreover, the level *l* of a point $P_{i,j}$ is also determined by the indices *i, j* and can be calculated in *O(1)* time in terms of computer operations.

Theorem 2.1 *Of a point $P_{i,j}$ on the grid, the level l in the quadtree hierarchy can be computed in O(1) time in terms of machine operations given the indices i and j.*

Proof The level *l* is determined by the largest quadtree block size where $P_{i,j}$ is a corner-vertex. This largest block size $d_l = (n-1)/2^l$ with $n = 2^k + 1$ has to be a power of 2 because of the hierarchical quadtree decomposition. Because of the relation $i = h_l d_l$ with $h_l = 0..2^l$ and $2^l d_l = n - 1$, the distance d_l in the dimension of *i* is the greatest common divisor $\gcd(i, n-1)$. Therefore, in binary representation d_l of *i* equals to the value of the least significant bit of *i* which is not zero. The least significant and non-null bit of *i* can be obtained by:

$$t(i) = (2^{\lceil \log_2 i \rceil} - i) \oplus i$$
$$d_l(i) = (t \oplus 2^{\lceil \log_2 t \rceil}) \otimes (2t) \qquad \textbf{(EQ 1)}$$

Where $\oplus$ denotes bitwise or combination (disjunction) and $\otimes$ the bitwise exclusive or. Thus with the indices *i* and *j* in two dimensions, the level *l* can be obtained in constant time by:

$$l = \max\left(\log_2\left(\frac{n-1}{d_l(i)}\right), \log_2\left(\frac{n-1}{d_l(j)}\right)\right) \qquad \square \text{ (EQ 2)}$$

FIGURE 2. Hierarchy levels

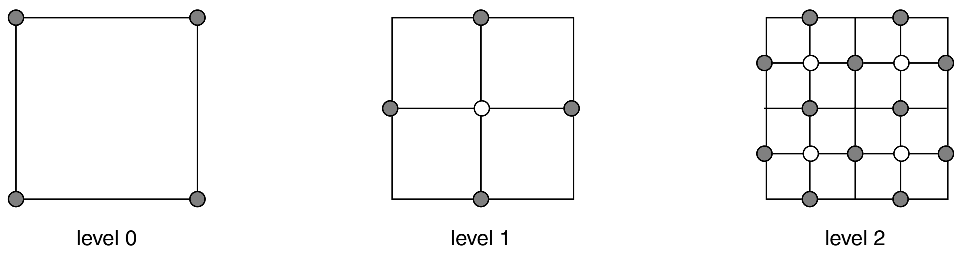


Figure 3 shows all possible basic triangulation variations of a quadtree block. Other alternatives can be realized by rotating the basic variations by a multiple of 90° clock- or counterclockwise. Furthermore, every reoccurrence of the initial two-triangle block pattern at a smaller scale as in Figure 1 c), can recursively be replaced by one of the other triangulation variations to form the hierarchical quadtree triangulation. An example of such a recursive, hierarchical triangulation is also shown on the left side of Figure 4. However, so far the hierarchical triangulation does not prevent *cracks* in the corresponding three-dimensional surface as shown on the right of Figure 4.

FIGURE 3. Triangulation variations

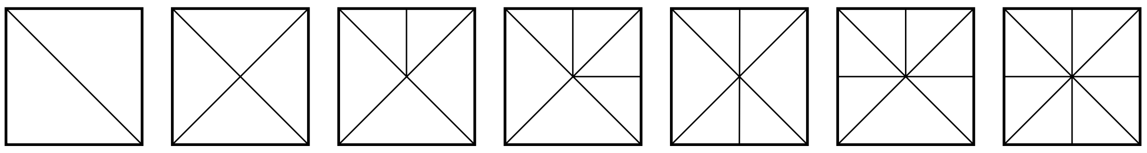
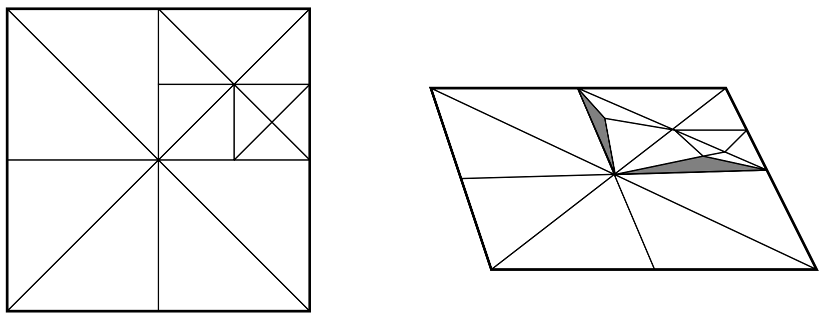


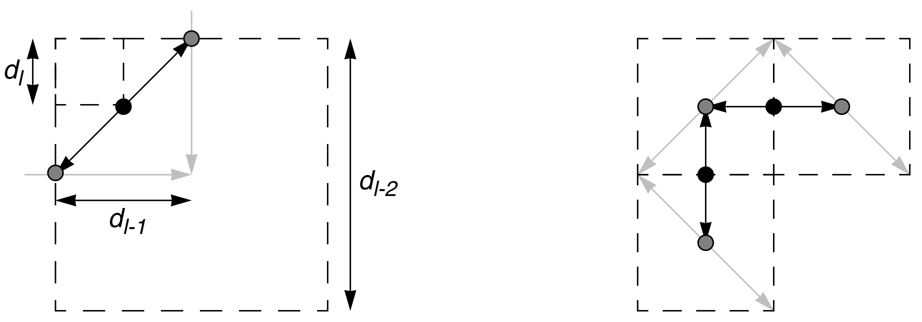
FIGURE 4. Nonrestricted quadtree triangulation



To guarantee a matching triangulation, cracks have to be avoided. In [20] this was granted by using a triangulation rule combined with the restricted quadtree requirement. The restricted quadtree requirement says that adjacent quadtree blocks differ by at most one level in the quadtree hierarchy. The triangulation rule is that every quadtree block is triangulated by eight triangles, or two triangles per boundary edge, unless the edge borders a larger block. However, this method may produce unneeded extra triangles in some cases. Furthermore, the two constraints of [20] provide no deterministic algorithmic handle of how a plain quadtree, or even only a set of points on the grid can be turned into a RQT. The two step subdivision of Figure 1 leads to a dependency graph on the vertices of the grid, see also [11]. This dependency graph can be used in a vertex selection algorithm to create a restricted quadtree from any given set of points on the grid.

The rule which defines the dependency graph of the restricted quadtree, that guarantees a matching triangulation, is shown in Figure 5. Every vertex depends on two other vertices of the same or the next higher level in the quadtree hierarchy. This means that if the vertex is selected for triangulation then the related dependencies must be selected too. As this dependency graph propagates upwards through the quadtree, the center-vertices of the lowest blocks in the hierarchy, lying on the finest grid resolution, are not dependents from other ones. And at the top level, the four corner-vertices of the quadtree root block are mutually dependent on each other. A center-vertex on level l depends on two diagonally opposite and adjacent vertices of level $l-1$, shown in Figure 5 on the left side. These two vertices are located in the middle of the border of a quadtree block of size d_{l-2}. The other vertices on level l depend on two vertically or horizintally adjacent points of the same level. These points are themselves center-vertices of level l, shown on the right hand of Figure 5. Remember that a point on level l is a center-vertex if it is located in the center of the next larger enclosing quadtree block of level $l-1$.

FIGURE 5. Vertex dependencies



The complete dependency graph for the first two relevant levels is depicted in Figure 6, the dependencies of the top level 0 are mentioned above. We will use the notation $G_{dep} = (V_{dep}, E_{dep})$ to refer to the complete dependency graph. It includes all vertices of the height field $V_{dep} = \bigcup L_{\log 2(n-1)}$. The dependency relation between vertices is denoted by the directed edges set $E_{dep} = \{(P_1, P_2) | \text{ dependency from } P_1 \text{ to } P_2\}$. As the the assignment of grid vertices $P_{i,j}$ to a level l in the quadtree, also the dependency relation can be expressed as a function over d_l, l and the point's indices i,j. We will demonstrate that here for the center-vertices which can be determined by the relation

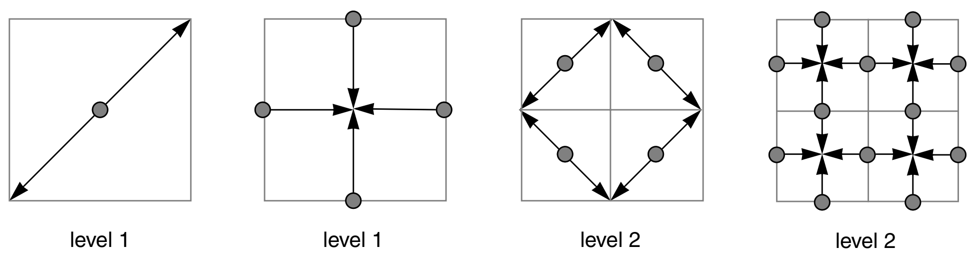
$$P_{i,j} \in L_l^{center} \Leftrightarrow \frac{i}{d_l} \text{ MOD } 2 \neq 0 \wedge \frac{j}{d_l} \text{ MOD } 2 \neq 0, \qquad \textbf{(EQ 3)}$$

after the distance d_l and the level l have been calculated by Equations 1 and 2. The next larger enclosing quadtree block of the center-vertex $P_{i,j}$ is located at $x = (i \text{ DIV } 2d_l)$ and $y = (j \text{ DIV } 2d_l)$. The $x \text{ MOD } 2$ and $y \text{ MOD } 2$ determine the diagonal direction of the dependency arcs from $P_{i,j}$ in comparison to the top-level dependency of the center-vertex in L_1^{center},

$$dir = x \text{ MOD } 2 + y \text{ MOD } 2 = \begin{cases} 0, 2 \Rightarrow \text{same diagonal direction} \\ 1 \Rightarrow \text{opposite diagonal direction} \end{cases}.$$

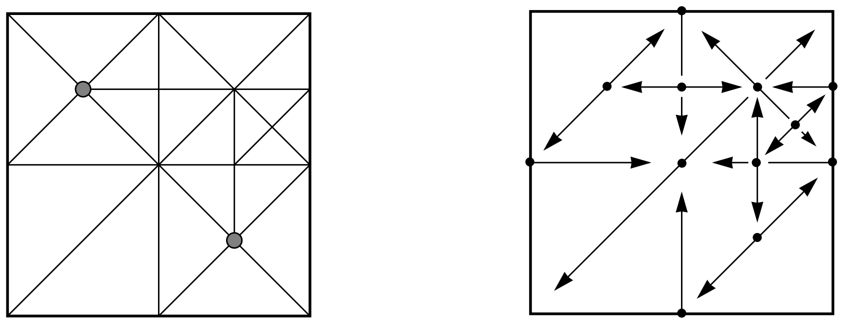
The diagonal distance is $\sqrt{2}d_l$. A similar method can be applied to the other vertices of $L_l \backslash L_l^{center}$ to determine if their dependency edges point to vertically or horizontally adjacent vertices in distance d_l.

FIGURE 6. Dependency graph



The hierarchical triangulation of Figures 3 and 4 implies that cracks can only occur between adjacent quadtree blocks. Either these blocks are of different levels or have a different triangulation variation at their common edge. Therefore, the center-vertices cannot be the immediate source of cracks as they are not located in the middle of the border edges of quadtree blocks. The dependency graph prevents cracks by consistently restricting the selection of points such that a matching triangulation results between all blocks. Figure 7 shows the restricted version of the quadtree triangulation from Figure 4. The additional vertices are emphasized on the left side, and the dependency graph is shown on the right. Hence, any given set of points on the grid can now decisively be turned into a restricted quadtree using the dependency rules. Furthermore, because the triangulation is hierarchically defined, every node in the quadtree is recursively triangulated using the triangulation variations shown in Figures 3 and 4. Note that this triangulation is computationally very efficient because it is given implicitly. No geometric computations such as in-circle tests need to be preformed. Furthermore, the resulting triangles are well shaped, are all isosceles and have a 90° angle. Moreover, they form a Delaunay triangulation in the 2D projection. However, it is a degenerated one because often 4 points are located on a triangle's circumcircle.

FIGURE 7. Restricted quadtree triangulation



2.2. Restricted quadtree construction

The dependency graph introduced in the previous section is a basic component of the two restricted quadtree construction algorithms presented here. We distinguish between two ways in which a point is selected for the requested triangulation. Either it is selected because of an approximation error criterion, or because the point is needed to create a matching triangulation. We assume that the decision if a point contributes to the triangulation based on an approximation error threshold can be determined by comparing with the threshold, or that the decision can be made locally at low cost. The actual calculation of a suitable approximation error is described in the next subsection. Basically every selection method of points can be used. By collecting all points following the dependency graph any point set can be completed to a restricted quadtree. In contrast to [16] we do not consider the problem of reading the vertices of a height field sequentially row by row. However, we assume an efficient random access based on indices, as it is usual for arrays and matrix structures. The efficient, incremental construction of a region-quadtree [15] is left to the reader. Furthermore, a quadtree data structure on a $(2^k+1) \times (2^k+1)$ grid can be implemented using index operations and recursion instead of using pointers and nodes. We call this an implicit quadtree defined on a grid.

Our first method to construct a restricted quadtree and its triangulation is a recursive *top-down* algorithm that operates on the quadtree hierarchy itself. We assume that the elevation points of the height field are already kept in a region quadtree, and that an approximation error is associated to every point. Additionally, each node knows the maximal approximation error of its own triangulation with respect to all points stored in that subtree. This error is simply the maximum error of the vertices in that node with respect to the error definition of Section 2.3. Furthermore, the dependency relation is already computed for every vertex, and the related points can directly be reached. As an alternative the dependencies and the related points location can be calculated from the grid indices as mentioned earlier. The procedure *SelectVertices()* tests and selects all vertices satisfying the approximation threshold of the current node, and then recursively traverses the given quadtree whenever further subdivision is required. Without the procedure *ResolveDependencies()* this traditional tree traversal would return an unrestricted selection of points. For a given point v, *ResolveDependencies()* adds the related vertices $\{p \mid (v, p) \in E_{dep}\}$ to the current selection, and follows the dependency graph only as long as necessary.

ALGORITHM 1. Recursive restricted quadtree construction

```
PROCEDURE SelectVertices(node: QTree; errmax: INT; VAR set: QSet);
VAR v: Vertex; son: POINTER TO QTree;
BEGIN
   FOREACH v IN node.vertices() DO
      IF v.error >= errmax AND NOT set.contains(v) THEN
         set.insert(v);
         ResolveDependencies(v, set)
      ENDIF
   END;
   FOREACH son IN node.descendents() DO
      IF son.maxerror() >= errmax THEN
         SelectVertices(son, errmax, set)
      ENDIF
```

```
    END
END SelectVertices;

PROCEDURE **ResolveDependencies**(v: Vertex; VAR set: QSet);
VAR dep: Vertex;
BEGIN
    dep := v.left();
    IF NOT set.contains(dep) THEN
        set.insert(dep);
        ResolveDependencies(dep, set)
    ENDIF;
    dep := v.right();
    IF NOT set.contains(dep) THEN
        set.insert(dep);
        ResolveDependencies(dep, set)
    ENDIF
END ResolveDependencies;
```

In the previous two procedures of Algorithm 1, the collection of selected vertices forming the restricted quadtree is extracted and stored in a *QSet*. The *QSet* can be a new (restricted) quadtree or any other data container which can be used for different purposes like transmission over a network. Otherwise, *QSet* can be abstract in the sense that it is not really another data structure but only a selection of vertices in the original quadtree structure. Moreover, this restricted quadtree selection can even just be an implicit quadtree defined on the height field. For that, the procedures *set.insert()* and *set.contains()* can simply be implemented by setting and testing a flag for every point. The restricted quadtree with the appropriate error tolerance then consists of the flagged points in the original quadtree, or it is made up implicitly by the marked points of the height field. The complexity of Algorithm 1 and the respective triangulation is linear in the size of the extracted restricted quadtree.

Theorem 2.2 *The number of calls to set.contains() and set.insert() of Algorithm 1 is linear in the size of the resulting restricted quadtree Q.*

Proof A node of the input quadtree is required if the subtree of that node contains a point that does not satisfy the approximation tolerance. All the required nodes are visited once by *SelectVertices()*. The resulting restricted quadtree Q consists at least of all required nodes of the input quadtree. $\Rightarrow \mathrm{O}(|Q|)$ calls to *set.insert()* and *set.contains()* by *SelectVertices()*.

For every selected point in *SelectVertices()*, *ResolveDependencies()* is called once. The recursion in *ResolveDependencies()* visits all vertices of Q at most four times as no more than four arcs of the dependency graph end in one vertex, see also Figure 6. $\Rightarrow \mathrm{O}(|Q|)$ calls to *set.insert()* and *set.contains()* by *ResolveDependencies()*. □

Moreover, the complexity is theoretically optimal in the sens of [14]. The RQT is a multi-triangulation and it can be constructed in linear time of the output size, the number of triangles issued.

Contrarily to the top-down method in [16] we do not predict and interpolate next-level vertices to guide recursive tree traversal. We can use the exact approximation errors of Section 2.3 as subdivision criterion. Also we handle the cascading splits implied by the restricted quadtree constraints differently, using the dependency relation E_{dep}. Furthermore, we do not have to deal with insertion of nodes into the quadtree if we use an implicit quadtree defined on the heigt field.

In contrast to the previous method, Algorithm 2 operates iteratively *bottom-up* on the height field. Starting at the lowest level with the highest resolution of the quadtree hierarchy, see also Figure 2 on page 5, the algorithm iterates over all levels examining the particular points on each. But for the compulsory organization of the points in a matrix or height field, the same presumptions hold as for Algorithm 1. During the inspection of points within a level, the approximation errors and the marks from dependency relations are checked for every point. If the error exceeds the tolerance criterion, or if the point is marked then this vertex is added to the result. Additionally, the two related vertices are marked accordingly. Note that we first iterate over all points $L_l \backslash L_l^{center}$ within a level, and then examine the center-vertices L_l^{center}. Thus, marks are never set on vertices which have already been inspected. And therefore, the marks can be cleared immediately after inspection if not needed further. This eliminates a preceeding cleanup stage to remove all marks before the next restricted quadtree construction.

ALGORITHM 2. Iterative restricted quadtree construction

```
PROCEDURE CollectVertices(field: Matrix; n, errmax: INT; VAR set: QTree);
VAR l: INT; v: Vertex;
BEGIN
   l := $\log_2$(n-1);
   WHILE l >= 0 DO
      FOREACH v IN field.verticesOfLevel(l)
         IF v.error >= errmax OR v.marked THEN
            set.insert(v);
            v.left().marked := TRUE;
            v.right().marked := TRUE
         ENDIF;
         v.marked := FALSE
      END;
      DEC(l)
   END
END CollectVertices;
```

In contrast to the first algorithm, this bottom-up approach visits vertices only once. However, the marking mechanism has to be studied carefully such that no dependencies might be left out in any case. Visiting the center-vertices last on each level guarantees that only points are marked which will be visited afterwards in the same level, or during the next iteration on the next higher level, see also Figure 6. Hence, no dependencies will be lost, and marks can be cleared after visiting the corresponding point. As a consequence, no recursive resolution of the dependency graph has to be performed to construct the restricted quadtree. As well, no set-inclusion tests are needed to stop recursive selection of vertices. Similar to Algorithm 1, the selected vertices, building the restricted quadtree, have to be collected and stored in a query answer data structure, or form an implicit quadtree on the grid. The efficiency of Algorithm 2 results overall from a low number of simple operations computed per input vertex.

Theorem 2.3 *Algorithm 2 runs in time linear of the size of the input data, and no dependency relation of G_{dep} is omitted.*

Proof The while loop in Algorithm 2 iterates over all levels, and for each level the inner foreach loop must only visit points on that level. For a level l and the corresponding grid resolution d_l this can be done by only visiting the points $\{P_{i,j} | (i \text{ DIV } d_l) \text{ MOD } 2 \neq 0 \vee (j \text{ DIV } d_l) \text{ MOD } 2 \neq 0\}$. Hence every vertex is visited exactly once. $\Rightarrow O(n)$ for n input points

To further guarantee consideration of all relevant dependencies of G_{dep}, the non center-vertices have to be examined first in the iteration over all vertices of one level. This is sufficient because the dependencies of non center-vertices of level l point to the center-vertices on the same level, and the center-vertices of level l point to vertices on level $l-1$. For the inner foreach loop this can be achieved by first visiting only the points $\{P_{i,j} | i|_2^{d_l} \neq 0 \text{ XOR } j|_2^{d_l} \neq 0\}$ and afterwards $\{P_{i,j} | i|_2^{d_l} \neq 0 \wedge j|_2^{d_l} \neq 0\}$ as indicated by Equation 3. ($k|_2^{d_l}$ abbreviates $(k \text{ DIV } d_l) \text{ MOD } 2$ in this context) □

In comparison to the bottom-up method in [16], we do not need to handle merges of nodes in a hierarchical quadtree data structure or examine neighboring quadtree nodes if we use an implicit quadtree defined on the height field.

2.3. Error function

As mentioned earlier, we compute an object-space error measure on the triangulation and associate an approximation error value to every point and quadtree block. Our error measure $e_T(P)$ for a Point on the grid and a triangulation T is the point's euclidean distance $d(P, t)$ to the triangle $t \in T$ with $P_{z=0} \in \Delta(t)$. $\Delta(t)$ is the domain of the triangle in the x,y-projection $\mathbb{R}^2$. If $P_{z=0} \notin \Delta(t)$ then $d(P, t) = 0$. Therefore, the error of a triangulation T is the maximal error of all points not included in T, given by $e_T = \max_{\forall P \notin T}(d(P, t)) \wedge t \in T$. However, instead of maintaining the distances to all possible triangulations for every point we impose a partial selection order on the vertices. This order is based on the restricted quadtree level hierarchy. The error calculation is derived from the two-dimensional line simplification method of [5]. The error value of a point P is the largest distance of P and all lower level vertices affected by P to the *best* triangulation not containing P. The error of a quadtree block is the maximum error of all points in its subtree.

Starting at the highest level, the approximation errors can be computed level by level, center-vertices first. To calculate the errors on level l for the center-vertices L_l^{center}, we consider the *next best* triangulation to be $T_{l-1} = \{\text{RQT of } \bigcup L_{l-1}\}$, and for the other points $L_l \backslash L_l^{center}$, it is $T_{l-1} = \{\text{RQT of } \bigcup L_{l-1} \cup L_l^{center}\}$. With the *covering* of a point P we denote the triangles $Cov(P \in L_l) = \{t \in T_{l-1} | \Delta(t) \cap P_{z=0} \neq \emptyset\}$. $Cov(P \in L_l)$ is the set of all triangles which are affected, will be subdivided, if P is selected for triangulation. The transitive closure of P is $\zeta(P) = \{Q \in V_{dep} | \exists P_1, .., P_k; P \leftarrow P_1, .., P_k \leftarrow Q\} \cap \{P\}$ using the inverse dependency relation $P \leftarrow Q \Leftrightarrow (Q, P) \in E_{dep}$. Given the dependency graph $G_{dep} = (V_{dep}, E_{dep})$ and the inverse dependency relation, the approximation error of $P \in L_l$ is the maximum error with respect to the covering *Cov(P)* of all points in the transitive closure. Hence, the approximation error for $P \in L_l$ is

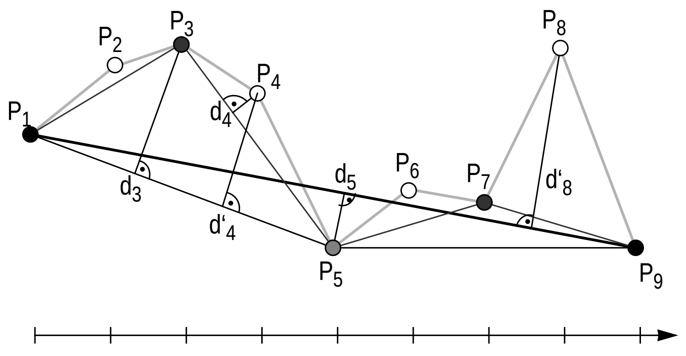
$$e_{T_{l-1}}(P) = \max_{Q \in \zeta(P)}(d(Q, t)) \wedge t \in Cov(P). \tag{EQ 4}$$

Figure 8 illustrates a situation in a two-dimensional analog with 3 levels. The levels and the dependency graph are:

$$\begin{aligned}L_0 &= \{P_1, P_9\}, L_1 = \{P_5\}, L_2 = \{P_3, P_7\}, L_3 = \{P_2, P_4, P_6, P_8\}\\ V_{dep} &= \bigcup L_3\\ E_{dep} &= \{(P_5, P_1), (P_5, P_9), (P_3, P_1), (P_3, P_5)(P_7, P_5), (P_7, P_9), (P_2, P_1), (P_2, P_3), \ldots\}\end{aligned}$$

The error measure is the distance d_k of a point P_k to a line-segment $\overline{P_i, P_j}$ between points of higher levels. As there are no more higher level points, the errors for P_1 and P_9 are equivalent to their elevation above the baseline. Point P_5 will be assigned the error d'_8 because this is the maximum error of all vertices in the transitive closure $\zeta(P_5) = \{P_2, P_3, P_4, P_5, P_6, P_7, P_8\}$ to the line-approximation $\overline{P_1, P_9}$, especially $d'_8 > d_5$. On the next level however, the error of P_3 is exactly d_3, because no other vertices in $\zeta(P_3) = \{P_2, P_3, P_4\}$ have a bigger distance to $\overline{P_1, P_5}$.

FIGURE 8. Error calculation, 2D analog to restricted quadtree



More informally, this error measure prevents unintentional omission of lower level vertices. Let us assume that the error $e_T(P)$ had just been set to the distance of P to its next best triangulation. The selection process could then miss out vertices on a higher resolution grid because their error was computed against a better triangulation than the current one of the selection process.

A continuous, triangulated surface which exactly satisfies the error tolerance ε everywhere on the original grid is called an ε-approximation. For any given error threshold ε, the error measure $e_T(P)$ of Equation 4 together with the RQT guarantees, that we can efficiently construct an ε-approximation. This solves the exact approximation problem of restricted quadtrees effectively. [3] considered this (unsolved) problem to be a severe drawback of the RQT. Also the algorithms presented in [16] cannot provide an exact ε-approximation because their subdivision criterion does not take into account all distances of lower level vertices to the current triangulation accuracy.

Other error measures can be used as well, also on-line error functions which are evaluated for every examined point during the RQT construction phase. However, to guarantee an exact approximation, such error functions need to be computed for the transitive closure of every selected point as in Equation 4.

3. Progressive meshing

Progressive meshing denotes the construction and transmission of a triangulation in such a way that a given triangulation can be updated easily given a few (or only one) new elements such as points, edges or triangles. Preferably, such an update is locally bounded and can be computed in constant time. Locally bounded means that the update affects the topology only nearby the new elements. Progressive meshes [8] have recently attracted growing interest, and most hierarchical triangulations [3,4] naturally support progressive meshing. We will demonstrate progressive meshing for the RQT.

A *breadth-first* traversal of the restricted quadtree Q equals to a level-wise ordering of the vertices. This ordering builds up the progressive mesh sequence of vertices. However, within the same level, the vertices $L_l^{center}(Q)$ have to be transmitted first to warrant a consistent RQT at the receiving end. Using $\leq$ as the ordering relation, and starting with the top level $L_0(Q)$, the progressive mesh ordering of the vertices is:

$$L_0(Q) \leq L_1^{center}(Q) \leq L_1(Q) \backslash L_1^{center}(Q) \leq .. \leq L_{\log 2(n-1)}(Q) \backslash L_{\log 2(n-1)}^{center}(Q) . \qquad \text{(EQ 5)}$$

Lemma 3.1 *For a given restricted quadtree Q, the vertex ordering of Equation 5 provides a progressive, matching triangulation.*

Proof For the restricted quadtree Q, this ordering takes into account all dependency relations. No vertex is inserted before its related vertices are. Therefore, a sub-quadtree $Q_l = \{\bigcup_{k=0}^{l} L_k(Q)\}$ is still a restricted quadtree because all dependencies are retained. This is still true even if the lowest level l of Q_l only consists of the center-vertices $L_l^{center}(Q)$.□

The updates are always locally bounded. Each vertex splits exactly two triangles up into four, and this can be done in constant time. Prior to the actual update, a *vertex location* operation is needed to select the right insertion point. This vertex location takes up $O(\log n)$ time in a quadtree data structure. However, the consecutive insertion of many vertices in a row into a tree structure leads to a lower amortized update cost.

The level-wise processing is not the only progressive mesh approach for a RQT. Incremental refinement of the triangulation can also be achieved much more irregularly. Any given restricted quadtree ε-approximation Q_ε, can incrementally be updated to a smaller approximation error δ. This is done by adding the vertices of an *error-range* quadtree $Q_{\Delta_{\varepsilon,\delta}}$, for $\Delta_{\varepsilon,\delta} = \varepsilon - \delta$. The construction of $Q_{\Delta_{\varepsilon,\delta}}$ differs from the normal quadtree extraction only in the selection of vertices. For $Q_{\Delta_{\varepsilon,\delta}}$ we select vertices according to an error-range instead of comparing to a threshold. However, the dependencies are resolved equivalently. Q_ε can also be regarded as an error-range quadtree with $\Delta_{\infty,\varepsilon} = \infty - \varepsilon$. Beginning with an initial RQT Q_{ε_0}, a progressive mesh sequence is then:

$$Q_{\varepsilon_0}, Q_{\Delta_{\varepsilon_0,\varepsilon_1}}, Q_{\Delta_{\varepsilon_1,\varepsilon_2}}, .. \qquad \varepsilon_0 > \varepsilon_1 = \varepsilon_0 - \Delta_{\varepsilon_0,\varepsilon_1} > \varepsilon_2 = \varepsilon_1 - \Delta_{\varepsilon_1,\varepsilon_2} > \dots .$$

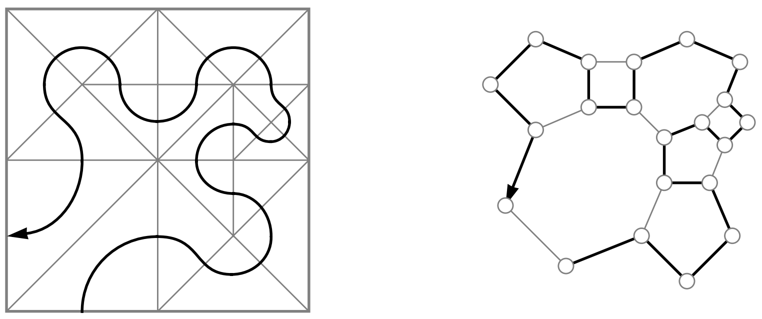
Note that transmitting each error-range quadtree's vertices ordered as in Equation 5, results in a very smooth progressive meshing. Every new vertex only requires a local replacement of two old triangles by four new ones. On the other hand, a more efficient update can be realized by merging two quadtrees. The error-range quadtree Q_{Δ_i} is inserted as a whole. Traversing both trees $Q_{\varepsilon_{i-1}} = Q_{\varepsilon_0} + \sum_{k=1}^{i-1} Q_{\Delta_k}$ and Q_{Δ_i} in *depth-first* order results in a linear time update complexity $O(|Q_{\Delta_i}|)$. The single vertex inserts lead to a $O(|Q_{\Delta_i}|\log|Q_{\Delta_i}|)$ complexity. If the error difference Δ_i is small enough and the vertices of Q_{Δ_i} are spatially scattered, a smooth mesh refinement is still achieved.

4. Performance

While the construction of a restricted quadtree from a given point-set is already very efficient as shown in Section 2.2, the basic recursive triangulation presented in Section 2.1 is not. So far we know how to construct a RQT as a set of triangles. However, *triangle strips* or *meshes* take up less space and are efficiently supported by hardware rendering engines and graphics software libraries. In contrast to [11] we present a more intuitive description of the triangle strip generation from a restricted quadtree. Furthermore, we give an algorithm outline using the quadtree data structure itself.

In a triangle strip, one has to assure that successive triangles always share an edge. Moreover, the orientations of subsequent triangles must alternate. This alternation can be enforced by *swap* operations, or multiple inserts of vertices resulting in invisible *line triangles*. Within a quadtree block triangles are arranged in a circle around the center-vertex. Hence triangles of one block share edges as required for triangle strips. In addition, because the RQT is a matching triangulation, neighboring blocks also share edges and can be queued up next to each other. Figure 9 on the left shows the basic idea of generating a triangle strip from a restricted quadtree. We recursively circle counter-clock wise around the block centers and output every triangle once with alternating orientation. Such a triangle strip represents a *Hamiltonian* path in the dual graph. In this graph each triangle is a node, and shared edges between triangles are edges between two nodes in the graph [2]. The dual graph example and the Hamiltonian path is shown in Figure 9 on the right hand.

FIGURE 9. RQT triangle strip and dual Hamilton path

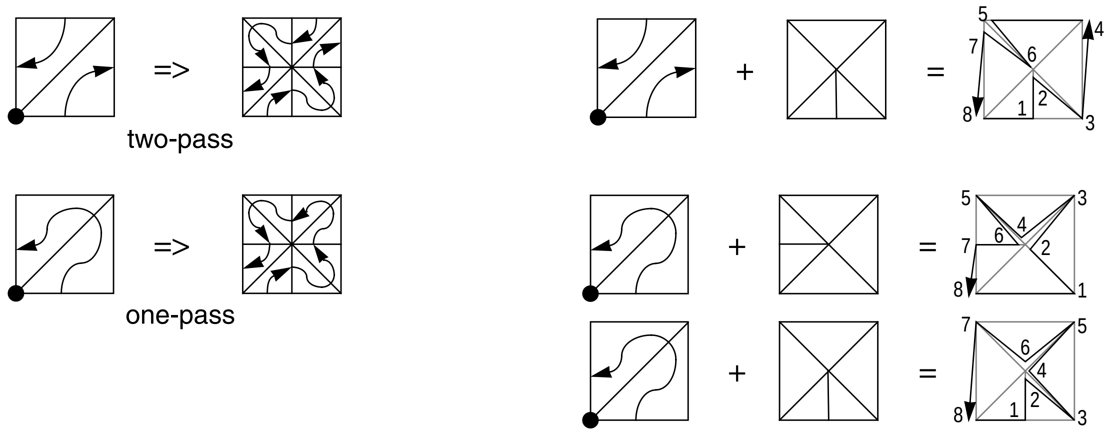


For each quadtree block, we can specify the corner-vertex where the triangle strip path enters to the right (inlet), and leaves the block on the left of this vertex (outlet). Therewith, all blocks can be characterized using only two different rotation invariant path types:

1. **two-pass**
   The block is triangulated in two passes. First the right side of the diagonal is triangulated, and in a second pass the left side. The block has two inlet-outlet pairs, one at the indicated corner and oneat the opposite corner of the diagonal.
2. **one-pass**
   The block is triangulated in one pass, starting at the specified corner on the right side of the diagonal. The block has only one inlet-outlet pair at the indicated corner.

When triangulating a quadtree block in this manner, we have to ensure that the last edge at the outlet is on the border of the block. This is required to satisfy the sharing edge criterion of consecutive triangles in a strip. Therefore, also the first edge at the inlet must be on the border of the block. Figure 10 shows the hierarchical decomposition and some of the triangulation rules for the different block types. On the right side, vertices are numbered in the order of their occurrence in the triangle strip. Note that the two vertex sequences of a two-pass block are not contiguous. With this triangulation scheme it is possible to traverse the quadtree, and issue the vertices in the correct order forming a triangle mesh. Besides the efficiency of rendering a triangle mesh, a mesh also offers an economic use of space. In the pathological case of circular arranged triangles still less than 2/3 of the number of vertices used for independent triangles are needed to describe the mesh.

FIGURE 10. Triangle strip decomposition



Theorem 4.1 *A triangle strip for the restricted quadtree Q can be constructed in time linear in the size of Q.*

Proof The traversal of the quadtree is limited by the one- and two-pass recursion schemes. Only the two-pass nodes are visited exactly twice, once for each inlet-outlet pair. Therefore, the number of nodes processed is at most twice the number of nodes in the quadtree. $\Rightarrow O(|Q|)$ nodes visited

Now, for each inlet-outlet pair we have to show that the correspoding sequence of vertices can be created in constant time. Note that in the triangle strip recursion, a quadtree block with only two triangles is triangulated in its father block, see also Figures 10, 3 and 4. Therefore, every visited block has a center-vertex called *c*. The other vertices are numbered counter-clock wise from the inlet = 0 to the outlet = *k*. Each triangle strip sequence of vertices for an inlet-outlet pair consists of:

$$\{0, 1, c, [\forall i, 1 < i < k-1 : i, c], k-1, k\}$$

The size of such a sequence is limited by15 if all vertices, including the corner-vertices, of a block are needed for triangulation. Therefore, for each inlet-outlet pair the corresponding triangle strip sequence can be generated in constant time. □

While already quite efficient, even the construction of the restricted quadtree can still be optimized for certain error measures. If the approximation error can be precomputed and stored with each vertex, the dependency resolution can be resolved beforehand of the point selection. Thereafter, the dependency resolution can be omitted in the construction phase. For that, we first calculate the correct approximation errors for each vertex as described in Section 2.3. Afterwards, the error is maximized on behalf of all vertices in the transitive closure of the inverse dependency relation. That is,

$$e(P) = \max_{Q \in \zeta(P)}(e(Q)) . \qquad \textbf{(EQ 6)}$$

Hence, we have a monotone error decrease from the lower levels in the quadtree to the higher level vertices. Consequently, dependencies are already encoded in the error values. A dependency $(P, R) \in E_{dep}$ has not to be resolved because *R* is automatically selected if *P* is, since $e(R) \geq e(P)$ from Equation 6. Using this modification of the approximation error, the construction of a restricted quadtree is simplified to the selection of all vertices satisfying the requested error threshold ε. The restricted quadtree selection and triangulation can be done on the fly while recursively traversing the input quadtree.

Theorem 4.1 *The set* $Q = \{P | e(P) \geq \varepsilon\}$ *is a restricted quadtree, given the error* $e(P)$ *is calculated as in Equation 4 and maximized according to Equation 6.*

Proof Assume that Q is not a restricted quadtree, then $\Rightarrow \exists P \in Q:(P, R) \in E_{dep} \wedge R \notin Q$ and $R \notin Q \Rightarrow e(R) < \varepsilon$. However, because of $P \in \zeta(R)$ and Equation 6 we have $e(R) \geq e(P)$. Because of $P \in Q \Rightarrow e(P) \geq \varepsilon$ we conclude that $e(R) \geq \varepsilon$ in contradiction to the assumption. □

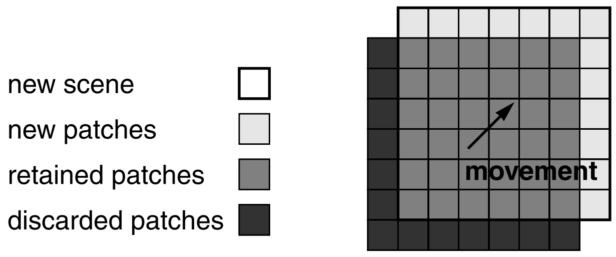
The preprocessing computation of Equation 6 can algorithmically be solved either by calling a recursive procedure similar to *ResolveDependencies()* of Algorithm 1, or by a bottom-up method similar to Algorithm 2. In both cases the approximation error of a vertex is maximized with the error of the dependent ones.

5. Dynamic scene management

As mentioned in the introduction Section 1 our aim is to visualize very large terrain databases. Therefore, we have to cope with the emerging data management problems. In contrast to [9] we advocate a *windowing* concept to dynamically move over the surface. This is similar to *panning* and *roaming* used in image viewing but with a perspective view of the terrain. Such a windowing approach is essential because we do not assume as [9] that the whole terrain database is maintainable in main memory. Conversely, the data can be managed on external storage. We also do not want to restrict access to the data based on a per file basis as in [6,17,18]. An interactive real-time environment restricts the visible scene to a fraction of such a large database. This calls for a dynamic scene management which frequently updates the partial scene to reflect changes of the view coordinates. Our visible scene always represents a window onto the world. Any change of the user's view coordinates likely incorporates an update of the visible scene. Such a dynamic scene management involving secondary storage is also often called a *paging* mechanism. This paging requires the database to provide spatial range-query functionality.

Whenever the coordinates of the visible scene window *W* change to represent a new view *W'* onto the world, the outdated data of region *W\W'* has to be discarded and the newly visible region *W'\W* must be loaded from the database. However, it is not very efficient to perform such an update for every small window variation $\varepsilon = W - W'$. Therefore, moving the visible window in discrete steps allows more economical and fewer updates. A partitioning of the visible scene into *rectangular patches* as shown in Figure 11, also called a tiling, efficiently supports discrete scene updates. The database reloads can be composed from fixed sized range-queries, one range-query for every newly visible patch. Tiling is an often used paradigm in visualization, GIS and spatial databases, and proofed its usefulness in many applications.

FIGURE 11. Dynamic scene update



The two-dimensional matrix of terrain patches of the visible scene is also called the *scene map*. Such a dynamic scene map enhances automatic culling of invisible terrain patches prior to rendering. Of the current scene map, only patches which have a non-empty intersection with the view frustum are kept in the display list. Hence, the per frame rendering operations need to process fewer data because rough culling is already performed, which improves display performance. Additionally, the scene manager keeps track of the levels of detail (LODs) at which the different patches have to be displayed. The resolution or LOD of each patch is determined by an application dependent parametrized function. We propose a function of the patch position relative to the observer's view coordinates. Section 6 goes into more detail about the LOD handling.

The display settings parameters are the visible scene window position, view frustum coordinates and LOD distribution. Although these parameters are computed for every frame, the visible scene map and the display list are not updated for every small change in the parameters. However, an update occurs only when one parameter exceeds a specific threshold. The window position update threshold is given by the size of a terrain patch. Whenever the observer crosses a patch boundary the scene matrix has to be adjusted as shown in Figure 11. Changes of the view frustum coordinates, mainly caused by changes of the view direction, affect the culling of invisible terrain patches. One one hand, an update is necessary when the view frustum's parameters have changed. On the other hand, an update is also required when the difference of the view direction exceeds the given rotation angle threshold α. Hence when actually culling the visible scene against the view frustum this threshold has to be taken into account. The opening of the view frustum has to be temporarily enlarged by α to perform the culling. Furthermore, updates in the resolution of terrain patches are only performed when the currently required LOD differs by more than the given threshold from the actually displayed one. Altogether, this approach of *deferred cumulative updates* helps to reduce the data management costs significantly without severe loss of display quality. It additionally provides further means to tailor the visualization system to available resources.

The dynamic scene manager maintains the points of each terrain patch in a quadtree data structure. Therein, the selected vertices that are currently marked to be displayed form a restricted quadtree at any time. Whenever an update is scheduled, a RQT reflecting the new situation is extracted and provided for rendering the patch at the requested resolution. The restricted quadtree Q_ε, or an error-range quadtree Q_Δ for incremental mesh refinement as described in Section 3, is also the basic transfer and query unit. This unit is used for reloading or updating a terrain patch from the spatial database. A quadtree can efficiently be transmitted in a depth-first order traversal synchronously for sending and receiving. That eases the communication between scene manager and terrain database.

Moreover, not only terrain surface data can be attached to every patch. Other data types include texture information, land use coverage and other geographically referenced objects, and points or polygons with additional non-geometrical attributes. For every patch, the scene manager maintains the various data types in different data structures. Each structure is chosen to be appropriate for the different query and update mechanisms.

6. Continuous LOD rendering

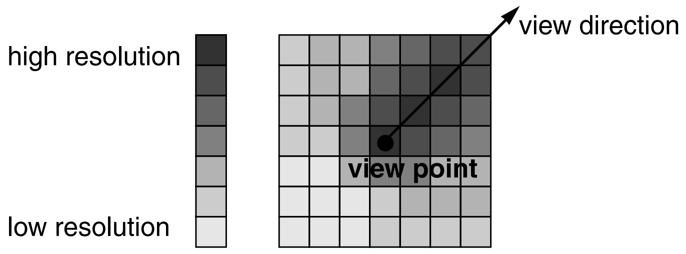
Continuous LOD stands for three different aspects of multiresolution surface visualization, which are all considered in this section:

1. representation of a scene in an (almost) unlimited number of different LODs,
2. display different parts of a scene at different resolutions without discontinuities in between,
3. smooth changes between different LODs of the same scene.

The first aspect is handled by a multiresolution triangulation algorithm and data structure as described in Section 2 that can extract an ε-approximation for any given error tolerance ε. The number of different LODs is usually not unlimited but finite. That is because not more LODs can be constructed using vertices from a given input model than the number of points of this model. Therefore, also the difference between two resolutions is discrete, at least one vertex, and is not infinitesimally small. However, more important is that the different LODs do not have to be precomputed in advance and stored redundantly along with the original full-resolution model. The different LODs must be efficiently extractable at any desired resolution from a compact multiresolution data structure. This is not the case in [12,17,18,19] but supported by the restricted quadtree data structure.

As announced in the previous section, presentation of different regions at different LODs is managed by the dynamic scene manager. An independent resolution is assigned to each terrain patch by the scene manager. This LOD is driven by an application dependent visibility priority calculation. We use the proposed object-space error measure of Section 2.3 to define the approximation accuracy of a patch. The LOD defines the approximation error threshold. To every patch a LOD is assigned based on relative distance to viewpoint and view direction, as schematically shown in Figure 12. A LOD update reflects variations of the current scene map settings and view coordinates. The update strategy is to load a terrain patch that is visible for the first time exactly at the desired LOD, and update it incrementally as required by changes of the view coordinates up to the full resolution if necessary. Note that unused vertices are never deleted until the complete patch is thrown out of the scene map. A marking scheme is used to identify the vertices which have to be rendered for the currently specified LOD.

FIGURE 12. LOD distribution

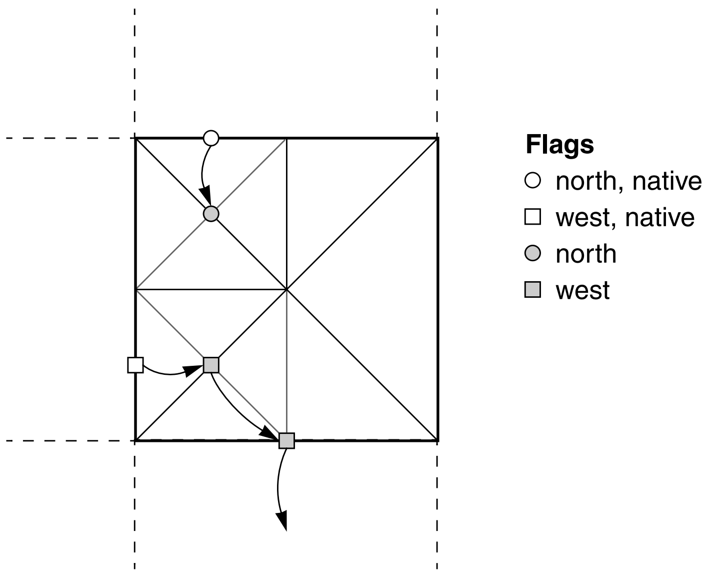


Avoiding discontinuity, or cracks, between regions of different terrain complexity or LOD, is another requirement of aspect two of continuous LOD rendering. This problem is difficult to solve for multiresolution triangulations, see also [1,4]. However, it is efficiently solved for grid-based models by the restricted quadtree triangulation (RQT) presented in Section 2. Nevertheless, neatly stitching together independent terrain patches of different resolutions is yet another problem. In [9] discontinuities were allowed between quadtree blocks. Whereas [17,18] solved the problem by consistently triangulating the borders of all patches at the same resolution for all LODs. In general, a matching triangulation between two patches with different borders can be achieved by mutually exchanging and inserting the missing border vertices on both sides. However, such point inserts can be quite expensive in sophisticated triangulation structures such as [4]. The given triangulation topology for a patch has to be altered, or a new, constrained triangulation has to be extracted for that patch such that it contains the required vertices on its border.

Even when using a RQT, modification of the triangulation is clearly not avoidable to match adjacent, triangulated regions but it is very simple and efficient. The RQT T of a patch can be modified by insertion of the missing border vertices V_b. The new set of vertices is then $V = V_T + V_b + V_{b_{dep}}$, consisting of the old vertices V_T of T, the missing border vertices V_b and the related vertices $V_{b_{dep}} = \{P | \exists R \in V_b : (R, P) \in E_{dep} \wedge P \notin V_T \wedge P \notin V_b\}$. Note that the vertices $V_{b_{dep}}$ can be interpolated without negatively affecting the approximation precision of T, and the vertices V_b only improve the accuracy. The modification is done by inserting the vertices V_b into the quadtree Q of the patch, resolving the dependencies, and extracting a new RQT. The related vertices $V_{b_{dep}}$ can be constructed or selected while traversing the quadtree Q top-down for the insertion of the vertices of V_b. The vertices V_b and $V_{b_{dep}}$ which guarantee a matching triangulation between two patches are marked to be included in the triangulation. Otherwise, they could be left out in the construction of the modified RQT.

A terrain patch not only has to maintain the vertices in a restricted quadtree but must also keep track of the status of each point indicating where it comes from. This source status consists of a combination of the flags *native*, *north*, *south*, *east* and *west*. The *native* flag signifies that all information of that point is known from the database. The other four flags denote an origin from a neighboring patch in the scene map. A non-*native* vertex has always an interpolated height and an inherited error value. These values will be overwritten as soon as the correct values are known. Vertices with flags *north*, *south*, *east* or *west* set, denote points which are not selected to satisfy the approximation tolerance of the corresponding terrain patch but required to build a matching triangulation between adjacent patches. Therefore, only updates of a terrain patch's triangulation which affect vertices on the borders require again an exchange of border vertices between the patches concerned. Figure 13 shows a patch with induced vertices from the northern and western map neighbors, and it demonstrates the transitive aspect of border exchanges.

FIGURE 13. Border exchange



In Figure 13, the induced vertices from the west lead to a modified patch border on the south. This invalidates a previously matching triangulation with the southern patch neighbor. Thus forcing anew a border exchange step. Consistency signifies matching triangulations between all patches. Generally, the scene map has to be checked for consistency after every border exchange as long as invalid borders occur. However, clever organization and ordering of patch updates, border exchanges and map checks reduce the required computation time to an insignificant share in the overall time consumption. In a traversal of the map we exchange borders where necessary between patches. Traversing the map twice in opposite directions, and checking for map consistency afterwards, eliminates the need for a second exchange-check phase in most cases. Performance can further be improved by choosing a path through the map from high resolution to low resolution patches, because induced vertices tend to originate from the higher LOD patch.

Aspect number three of continuous LOD rendering calls for smooth continuous changes between LOD updates within the visible scene. In [8] the term *geomorph* was coined to denote morphing between different LODs of the same terrain segment. For geometries, and especially triangulations, LOD morphing can be described in two steps:

1. Refinement without change of the shape
   Geometric primitives are subdivided in such a way that the new, interpolated vertices, edges and faces exactly remodel the object without change of approximation accuracy. Besides the interpolated coordinates v', each vertex also stores its original coordinates v where it finally should reside.
2. Morphing vertices
   The (new) vertices are moved continually from their interpolated position v' to their true location v, using a blending function $f(v', v, t)$ over time or distance t.
   (i.e. $t = 0.0..1.0$ and $f(v', v, t) = (1.0 - t)v' + tv$ with t dependent from the distance to the current view coordinates)

The RQT is very well adapted to morphing between LODs. Incremental refinement has been described in a previous section. A vertex v always lies in the middle of an edge of two vertices v_1 and v_2 further up in the dependency graph. These two vertices are in diametrical position to the immediate two dependencies of v. The interpolation can be calculated as $v' = 0.5(v_1 + v_2)$. The morphing function $f(v', v, t)$ can be applied directly to the geometries used for rendering without retriangulation of the quadtree itself. A pointer from the vertices in the quadtree to the coordinates of the rendered geometry primitives and vice versa will provide the necessary link.

7. Storage and retrieval

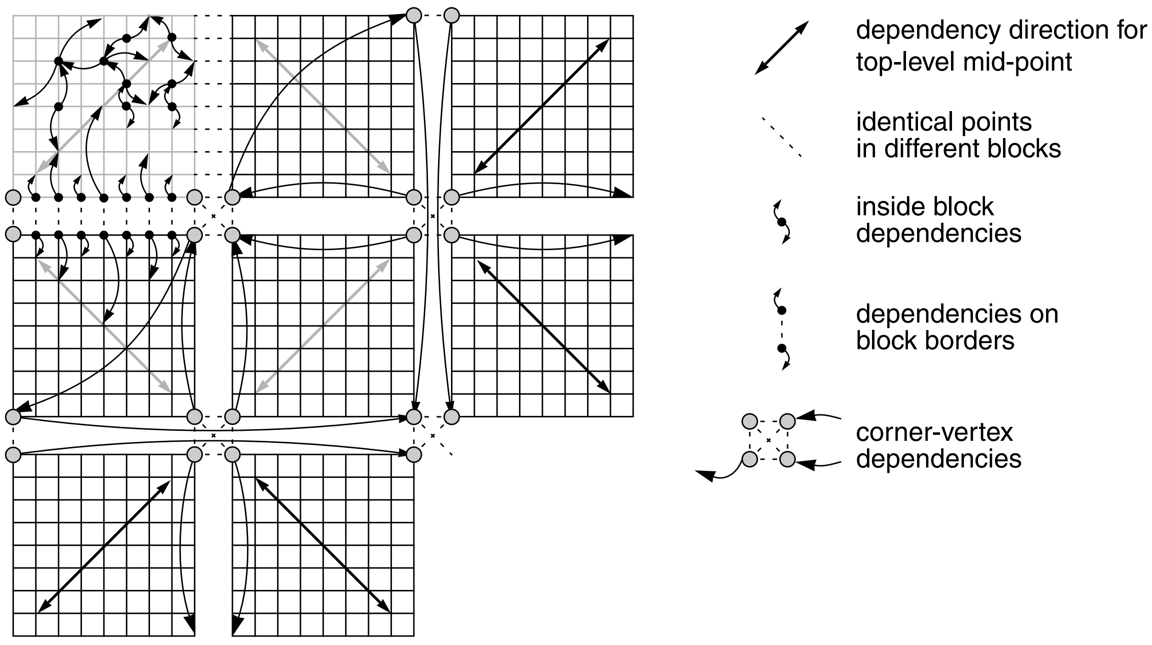
An efficient terrain database providing spatial access to the elevation data is the indispensable back-end for efficient large scale terrain visualization. The dynamic scene management has to rely on high-performance spatial retrieval of elevation data. Moreover, the multiresolution approach calls for spatial access enhanced by a LOD specification. The vast amount of elevation data has to be maintained on external memory where spatial data structures provide efficient access through spatial indexing and clustering. The proposed dynamic scene management of Section 5 requires that the spatial access structure supports rectangular range query operations with restriction to a given LOD range. The return value must be a restricted quadtree data set of elevation points. Therefore, the database must conform to the same *global* restricted quadtree Q as it is used for the triangulation. Hence the database has to use the same assignment of points to levels L_l, dependency graph G_{dep}, and relation of points to quadtree nodes as described in Section 2.1. However, no large data structure is needed for that. Specifying the relative origin for the quadtree, the smallest distance between elevation points on the grid, the dimensions of the data domain and the dependency direction for the top-level vertex in L_1^{center} of the largest quadtree region is sufficient for this purpose.

The regular nature of the height field as input data suggests the use of an equally regular data structure such as the gridfile, quadtrees or multidimensional hashing. They all partition the space into equally structured rectangular regions and support efficient spatial access. The LOD constraint can be satisfied by selecting points with an approximation error within the specified error tolerance given by the query's LOD range. Building a set of vertices forming a restricted quadtree follows the same principles as described in Section 2.2. Therefore, we can concentrate on a data structure for maintaining a very large height field, or grid of elevation points. Furthermore, the restricted quadtree dependencies must also be stored or computed efficiently. Among all possible data structures we will describe one here that fits best with the second restricted quadtree construction method, Algorithm 2 of Section 2.2, and that provides fast access to the elevation data.

The input data space is partitioned into blocks of $(2^k+1) \times (2^k+1)$ grid points. These blocks equal to the kth-last level nodes of the global quadtree Q and contain the vertices of the last k levels plus the four missing corner-vertices. This partitioning provides efficient spatial selectivity and supports physical clustering on external storage. Each block is organized as an array of $(2^k+1)^2$ vertices. A block can be located through a table look-up in a two-dimensional index structure, a matrix with header information for each block and references to the data arrays. An implicit restricted quadtree structure can be superimposed on every block given the dependency direction for the top-level mid-point of that block, as indicated in Figure 14. This top-level direction is part of the block's header information in the index structure. Note that all vertices *inside* of such a block have dependencies directed to vertices within the same block as well, see also Section 2. Contrarily, dependencies of border vertices point to one vertex in the same block and to one symmetrically in the adjacent block (*overlapping* dependencies).

Finally, the corner-vertices' dependencies point to other corner-vertices of blocks as depicted in Figure 14. It is also shown that the vertices on block borders are stored twice in adjacent blocks and the corner-vertices even four times. This results in a space overhead inversely proportional to the block width but only growing with the square root of the entire height field size.

FIGURE 14. Height field partitioning

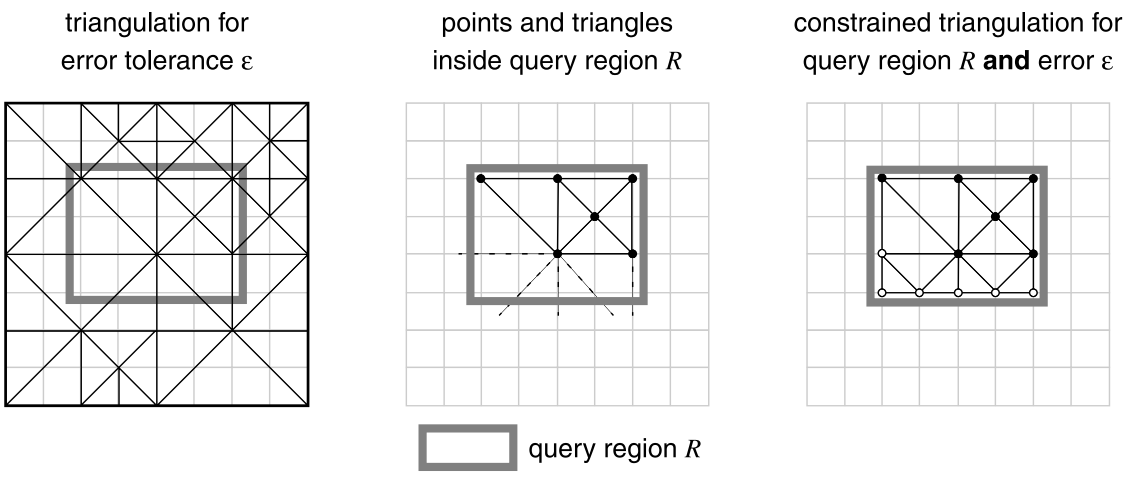

dependency direction for top-level mid-point
identical points in different blocks
inside block dependencies
dependencies on block borders
corner-vertex dependencies

Spatial access can be performed as usual in spatial data structures. The relevant data blocks intersecting the rectangular query region R are retrieved using the spatial index, and the exact query result is obtained by inspection of all elements in the selected blocks. This inspection step can readily be combined with the LOD selection criterion and the construction of a restricted quadtree. The bottom-up Algorithm 2 of Section 2.2 is performed level-wise on all blocks intersecting R simultaneously. Inspection of every point involves checking it against intersection or inclusion in R, testing the approximation error tolerance and resolving dependencies. Note that dependencies going outside of the smallest enclosing quadtree region of R do not have to be resolved as they no more contribute to vertices in R. Notice also that the blocks' corner-vertices and their dependencies lie on a lower resolution grid and make up the top $l - k$ levels of the global quadtree Q with width (2^l+1) and block width (2^k+1). The corner-vertices and dependencies can also be stored with the block header information in the index structure. Thus eliminating the need to access data blocks unaffected by the query region R just for resolving higher level dependencies. Below the necessary schematic steps for a query with region R and error tolerance ε, and the index structure S. Selected vertices are inserted into a quadtree dedicated to hold the query result, and used for transfer purposes (i.e. over a network).

1. find all blocks b in S intersecting the query region:
   $B = \{b \in S | b \cap R \neq \varnothing\}$
2. apply bottom-up restricted quadtree construction algorithm $\forall b \in B$, check each point against error threshold ε and intersection with R, keep track of *overlapping* dependencies
3. apply bottom-up or top-down restricted quadtree construction to the remaining corner-vertices depending on the organization of the spatial index structure

Special attention has to be paid to the border ∂R of the query region R. The geometrical test whether a vertex belongs to R or not, must be enhanced by constraints of the restricted quadtree triangulation as shown in Figure 15. The selection of vertices according to the approximation threshold and the query region does not lead to a correct RQT. The border ∂R actually constrains the selection to include all vertices needed to map ∂R on edges of maximal length which conform to the RQT. This is shown on the right of Figure 15. Due to the regular nature of the RQT, the constraints for each of the four edges of ∂R_{north}, ∂R_{south}, ∂R_{east}, and ∂R_{west} can be formalized in terms of levels in the restricted quadtree. If ∂R_x lies on a grid line of level l then all vertices of level l on ∂R_x have to be included. The four corner points of R must always be selected too.

FIGURE 15. Range query constraints



Also incremental refinement of the identical query region R is supported with this storage and retrieval structure. In contrast to the basic query with an approximation tolerance ε_0, equivalent to the error-range $\Delta_{\infty, \varepsilon_0} = \infty - \varepsilon_0$, an incremental query with $\Delta_{\varepsilon_{i-1}, \varepsilon_i} = \varepsilon_{i-1} - \varepsilon_i$ does not have to deal with the border problem ∂R of the query region. That is because the first query for R with ε_0 has already made sure that the border ∂R is consistently triangulated with respect to the global quadtree restriction rule.

Furthermore, the transfer of data from the terrain database to the visualization is very efficient. The triangulation topology does not have to be encoded and transmitted at all as it is given implicitly by the global restricted quadtree rules. The same is true for the dependency graph. Moreover, not even the spatial location of the vertices has to be conveyed because it is given by the vertex' position in the quadtree and the quadtree's origin in space. Therefore, only the height and error values for each vertex need to be transferred. Moreover, while constructing the restricted quadtree for a specific query, the dependencies can already be encoded in the error values as pointed out in Equation 6 on page 16. And this coding of dependencies facilitates the triangulation in the visualization. The transmission is reduced to a tree traversal of the restricted quadtree where for every point only its height and error value must be sent.

To benefit from transaction mechanisms such as concurrency control and failure recovery we advocate to integrate the terrain database into a commercial database management system. In [12] a terrain database using a hierarchical spatial data structure is integrated into ObjectStore, and [7] investigate on the integration of spatially clustering index structures into Oracle.

8. Conclusion

We have proposed an efficient model for large scale terrain visualization. It turned out that we cannot simply glue together known solutions for the various problem aspects. Knowledge from different fields, such as computer graphics, computational geometry, database systems and spatial data structures, needs to be incorporated to arrive at an efficient system. Furthermore, strong attention has to be paid to the multiresolution triangulation model and its integration into the system.

The restricted quadtree triangulation has several advantages compared to other multiresolution triangulations. Regular grid triangulations have the severe drawback that they are not adaptive to the terrain structure. That is caused by the selection of points which depends purely on the position and not on the terrain model itself. The consequences of sub-sampling and interpolation are smoothing and omission of relevant details. Another approach is to maintain a set of different triangulations for the different LODs. However, no continuous LOD rendering is supported in this way as the number of LODs is very limited. The transitions between different LODs are generally not matching as well. Furthermore, the storage costs are very high due to redundancy in the data and the representation of the topology for each LOD. An overview over grid-to-TIN[1] algorithms can be found in [10]. Therefore, these models do not meet several requirements mentioned in the introduction Section 1.

For other multiresolution models the selection of points for the different LODs is closely coupled with the triangulation. In general, a multiresolution triangulation model replaces a number of triangles by more and smaller triangles to refine the approximation, see also [1,14]. The construction of such a triangulation model is quite complex but can be done beforehand. However, the extraction and especially the incremental refinement of a specific triangulation is very slow for on-demand requirements. The topology representation is very complicated in comparison to the RQT. The same arguments hold for hierarchical triangulation models as described in [4], and no graphics optimizations are known for these models. Therefore, the requirements 4, 5 and 8 are not met. Furthermore, efficient large scale database management is not supported. The precomputed triangulation topology and hierarchy is complicated to maintain, and uses much more space than the restricted quadtree. Spatial access is not as efficient as well because topology and vertices have to be extracted from the database. Hence also the demands 1 and 2 cannot be fulfilled effectively. An overview over multiresolution models is presented in [3].

We have already compared our proposed model to the other terrain visualization systems in the text. Let us only summarize here. The systems [9,11] do not consider the problem of

1. TIN - triangulated irregular network

very large scale terrains. Therefore, the problems 3, 6 and 9 of dynamic scene management and the database aspects 1 and 2 are ignored. The proposal of [17] does not focus on the dynamic scene and database aspects too. Furthermore, it is lacking effective solutions to the triangulation requirements 4, 5 and 8. In [6] no multiresolution approach is followed at all. The first approach of [12,19] is missing an efficient multiresolution triangulation and continuous LOD rendering.

In contrast to other triangulation and visualization models our proposal fully satisfies al requirements for large scale terrain visualization listed in Section 1. In particular it supports many exceptional features also suggested in other publications:

- progressive transmission [8]
- smooth, continuous LOD transitions [11]
- fast access and manipulation of triangulation [3,4]
- real-time triangulation [11,14]
- LOD constrained, spatial access [14]
- efficient storage costs, mesh compression [3, 8]

Most of the presented concepts and solutions have found their way into a mature version of the ViRGIS[1] project [13]. ViRGIS mainly consists of a prototypical client-server system which allows interactive, three-dimensional visualization and exploration of large scale terrains, including satellite image texturing. The achieved high graphics performance allows to run the visualization even on small graphics workstations (SGI O2).

Acknowledgments

I would like to thank Peter Widmayer for his helpful comments on the paper. I am also very grateful to the students Christian Grob and Lukas Vonblon which implemented large parts of the newest version of the ViRGIS prototype system as a part of their master thesis.

References

[1] M. de Berg and K. Dobrindt. “On levels of detail in terrains”. In *11th ACM Symposium on Computational Geometry*, pages C26–C27. ACM, 1995.

[2] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry.* Berlin, Springer Verlag, 1997.

[3] L. De Floriani, P. Marzano, and E. Puppo. “Multiresolution models for topographic surface description”. *The Visual Computer*, 12(7), August 1996.

[4] L. De Floriani and E. Puppo. “Hierarchical triangulation for multiresolution surface description”. *ACM Transactions on Graphics*, 14(4):363–411, 1995.

[5] D. H. Douglas and T. K. Peucker. “Algorithms for the reduction of the number of points required to represent a digitized line or its caricature”. *The Canadian Cartographer*, 10(2):112–122, December 1973.

1. ViRGIS - Virtual Reality and Geoinformation Systems, http://www.inf.ethz.ch/personal/pajarola/virgis.html

[6] K. Graf, M. Sutter, J. Hagger, and D. Nüesch. “Computer graphics and remote sensing – a synthesis for environmental planning and civil engineering”. In *Proceedings EUROGRAPHICS 94*, pages C13–C22. IEEE, 1994. also in Computer Graphics Forum Vol. 13, Nr. 3.

[7] A. Henrich, A. Hilbert, H.-W. Six, and P. Widmayer. “Anbindung einer räumlich clusternden Zugriffsstruktur für geometrische Attribute an ein Standard-Datenbanksystem am Beispiel von Oracle”. In *GI-Fachtagung Datenbanksysteme für Büro, Technik und Wissenschaft*, volume 270 of *Informatik-Fachberichte*, pages 161–177. Springer-Verlag, 1991.

[8] H. Hoppe. “Progressive meshes”. In *Proceedings SIGGRAPH 96*, pages 99–108. ACM SIGGRAPH, 1996.

[9] D. Koller, P. Lindstrom, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. “Virtual GIS: A real-time 3d geographic information system”. In *Proceedings Visualization 95*, pages 94–100. IEEE Computer Society Press, Los Alamitos, California, 1995.

[10] J. Lee. “Comparison of existing methods for building triangular irregular network models of terrain from grid digital elevation models”. *International Journal of Geographic Information Systems*, 5(3):267–285, 1991.

[11] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. “Real-time, continuous level of detail rendering of height fields”. In *Proceedings SIGGRAPH 96*, pages 109–118. ACM SIGGRAPH, 1996.

[12] R. Pajarola, T. Ohler, P. Stucki, K. Szabo, and P. Widmayer. “The alps at your fingertips: Virtual reality and geoinformation systems”. In *Proceedings 14th International Conference on Data Engineering, ICDE ’98*, pages –. IEEE, 1998.

[13] R. Pajarola and P. Widmayer. “Virtual geoexploration: Concepts and design choices”. submitted for publication, 1998.

[14] E. Puppo. “Variable resolution terrain surfaces”. In *Proceedings of the 8th Canadian Conference on Computational Geometry*, pages 202–210, 1996.

[15] H. Samet. “The quadtree and related hierarchical data structures”. *Computing Surveys*, 16(2):187–260, June 1984.

[16] R. Sivan and H. Samet. “Algorithms for constructing quadtree surface maps”. In *Proc. 5th Int. Symposium on Spatial Data Handling*, pages 361–370, August 1992.

[17] M. Suter. *Aspekte der interaktiven real-time 3D-Landschaftsvisualisierung*. PhD thesis, Remote Sensing Laboratories, University of Zurich, 1997.

[18] M. Suter and D. Nüesch. “Automated generation of visual simulation databases using remote sensing and GIS”. In *Proceedings Visualization 95*, pages 86–93. IEEE Computer Society Press, Los Alamitos, California, 1995.

[19] K. Szabo, P. Stucki, P. Aschwanden, T. Ohler, R. Pajarola, and P. Widmayer. “A virtual reality based system environment for intuitive walk-throughs and exploration of large-scale tourist information”. In *Proceedings Enter95*, 1995.

[20] B. Von Herzen and A. H. Barr. “Accurate triangulations of deformed, intersecting surfaces”. In *Proceedings SIGGRAPH 87*, number 4 in ACM Journal Computer Graphics, pages 103–110. ACM SIGGRAPH, 1987.