An Efficient Representation of Quadtrees and Bintrees for Multiresolution Terrain Models

Yusnier Valle1, José Ortiz2

1. Center of Geoinformatics and Digital Signals, University of Informatics Sciences, Havana, CP. 19370, Cuba
2. Dean of the Faculty 1, University of Informatics Sciences, Havana CP. 19370, Cuba

© Wuhan University and Springer-Verlag Berlin Heidelberg 2011

Abstract A space-filling curve in 2, 3, or higher dimensions can be thought as a path of a continuously moving point. As its main goal is to preserve spatial proximity, this type of curves has been widely used in the design and implementation of spatial data structures and nearest neighbor-finding techniques. This paper is essentially focused on the efficient representation of Digital Elevation Models (DEM) that entirely fit into the main memory. We propose a new hierarchical quadtree-like data structure to be built over domains of unrestricted size, and a representation of a quadtree and a binary triangles tree by means of the Hilbert and the Sierpinski space-filling curves, respectively, taking into account the hierarchical nature and the clustering properties of this kind of curves. Some triangulation schemes are described for the space-filling-curves-based approaches to efficiently visualize multiresolution surfaces.

Keywords bintrees; quadtrees; space-filling curves; spatial data structures; digital terrain models

CLC number P208

Introduction

A Digital Terrain Model (DTM) is a digital representation of the properties of the topography of a surface. Specifically, Digital Elevation Models (DEM) are among the most used by the scientific community, which are usually stored as regular grids, Triangulated Irregular Networks (TIN), or contour lines. These aforementioned models are an important component in a wide range of application domains, such as scientific visualization, Geographic Information Systems (GIS), and interactive 3D games. Usually, the terrain model is not the most important component in a simulation; hence, the importance of ensuring its visualization at high frame rates. On the other hand, due to the increasing size and complexity of DEM, its real-time management imposes significant efficiency constraints on the visualization engine, which is forced to dynamically trade rendering quality with usage of limited system resources.

The steady increase of the storage capacity in recent computer systems, and the development of techniques for data acquisition of the earth surface relief, allow the creation of huge datasets of spatial information requiring high efficiency in data structures used for its management and representation. Among the most commonly used data structures are those based on Binary Triangles Trees (bintrees) hierarchies[1] and quadtrees.[2]

This paper addresses the problem of the efficient
representation of regular grid-based DEM that entirely fit into the main memory. The main contribution is focused on a compact representation of quadtrees and bintrees by means of the Hilbert and Sierpinski space-filling curves, respectively. First, unlike a classical size-restricted representation of a quadtree,[2] we propose a hierarchical data structure for representing a given regular domain of unrestricted size data. After that, a space-filling-curves-based multiarray representation for these data structures is described, which avoids the need to store information about the position of parent, children, and neighbor nodes, as these can be calculated by simple mathematical operations representing a significant saving in the overall memory consumption.

1 Previous work

The problem of mesh simplification and multiresolution surface triangulation has been widely studied over the last two decades. A number of approaches are based on the principle of Delaunay triangulation[3] to create TIN over irregularly spaced sets of points.[4] Other important contributions belong to Hoppe,[5] in which the progressive mesh representation is introduced, a new scheme for storing and transmitting arbitrary triangle meshes.

Other significant techniques based on quadtrees and bintrees are among the most used in the scientific community. Right-Triangulated Irregular Networks (RTIN),[1] like the ROAM[6] approach, which uses a priority-queue driven triangulation based on the notion of longest side bisection, is based on the same bintree subdivision scheme, but this mainly focused on the efficient representation of the hierarchy and the fast traversal of the structure for neighbor finding. The data structure described for the RTIN is widely considered as one of the most efficient representation of bintrees for multiresolution terrain modeling.

The main contributions on Restricted Quadtree Triangulations (RQT) belongs to Lindstrom et al.,[7] who introduces the vertex dependencies graph that can be used to prevent cracks and create triangulations at variable resolution, and to Pajarola,[8] who uses the dependencies relations presented by Lindstrom et al.[7] to generate minimal triangulations over implicit region quadtrees and reduces the storage cost effectively using both arithmetic and logical operations on large height-field grids. Röttger et al.[9] also proposes a quadtree-based terrain rendering engine, which is mainly focused on a memory efficient representation of the quadtree data structure by means of a Boolean matrix. The total memory consumption of this approach is very low due to, besides the height field, texture data and error values; just one byte per grid point is needed. A complete survey on these techniques was presented by Pajarola and Gobbetti in a recent paper.[10]

Today’s GPU-friendly approaches are mainly focused on composing at runtime precomputed surface patches using the GPU memory as a cache. Schneider and Westermann[11] proposes an effective CPU/GPU communication scheme to incrementally update a data structure of vertex and index buffers in the GPU. Triangle patches are also the basic unit of refinement in the Cignoni et al.[12] approach, which efficiently approximates the surface by means of a hierarchy of precomputed and stripped TINs, taking advantage of the high geometry throughput of current GPUs.

2 The FTTree data structure

The quadtree data structure is created from a given square with initial vertices in (0, 0), (0, U), (U, 0), and (U, U), with \( U=2^j \) for a positive integer \( j \). The restriction in size over \( U \) is assumed by other authors[8,9] to create quadtrees over regular grids representing height fields. If the original size of a dataset is unrestricted, it is necessary to deal with undesired additional nodes in the tree in order to hold this restriction. An FTTree (i.e., Four-Two Tree) is defined as follows:

**Definition 1** An FTTree is a tree in which each internal node has two or four children.

1. Each node in the FTTree corresponds to a quadrant.
2. If a node \( v \) has children, then their corresponding quadrants are two or four quadrants of the quadrant of \( v \).

2.1 Construction

The process of creating an FTTree is performed in a similar way as a quadtree.[2] Three different types of quadrants can be created in the division process:
(1) With four children, each child node can be identified as SW, NW, NE, and SE.

(2) Horizontally oriented with two children, the child node holding the leftmost (rightmost) quadrant is marked both NW and SW (NE and SE).

(3) Vertically oriented with two children, the child node holding the upper (lower) quadrant is marked both NW and NE (SW and SE).

Let $R$ be a regular grid of size $m \times n$, $m,n > 1$, and let $\sigma([r_x,c_y],[r_x,c_y],[r_x,c_y],[r_x,c_y])$ be the initial quadrant with corners at $(m-1,0)$, $(0,0)$, $(0,n-1)$, and $(m-1,n-1)$. Remark that if $m = n = 2$, then an FTTree consists only of a single leaf node where the square $\sigma$ is stored. Let $\sigma_{nw}, \sigma_{ne}, \sigma_{se},$ and $\sigma_{sw}$ be the four quadrants of $\sigma$, and let $P(r_x,c_y)$ be the point at which a given quadrant will be divided. The pseudocode of the Algorithm 1 illustrates how the division of a quadrant is performed.

Algorithm 1 Division of a given quadrant $\sigma$.

```
procedure SPLIT_QUADRANT ($\sigma, \tau$)
1:   width $\leftarrow$ WIDTH($\sigma$) $>\tau$
2:   height $\leftarrow$ HEIGHT($\sigma$) $>\tau$
3:   ratio $\leftarrow$ MAX(width, height)
4:   if ratio $\geq \tau$ then
5:      if WIDTH($\sigma$) $>$ HEIGHT($\sigma$) then
6:         $\sigma_{nw}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
7:         $\sigma_{ne}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
8:      else
9:         $\sigma_{nw}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
10:        $\sigma_{ne}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
11:       end if
12:   else
13:      $\sigma_{nw}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
14:      $\sigma_{ne}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
15:      $\sigma_{se}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
16:      $\sigma_{sw}$ $\leftarrow$ $[(r_x,c_y),(r_x,c_y),(r_x,c_y),(r_x,c_y)]$
17:   end if
```

Algorithm 1 starts calculating a ratio between the width and height of the quadrants to be created. If this proportional relation is greater than a given tolerance threshold (e.g., $\tau = 1.5$), $\sigma$ must be divided horizontally or vertically into two quadrants depending on its size. Otherwise, if ratio is near 1 (i.e., quadrants to be created are almost squares), $\sigma$ must be divided into four quadrants.

The splitting criterion used in the construction of the FTTree guarantees that if $m = n = 2^k + 1$, $k > 0$, the resulting structure is a quadtree. Similarly to a quadtree, when we descend from a node to one of its children, at least one of the edges of the corresponding quadrant is divided by two. That is why the depth of an FTTree for $R$ is $2 * \lceil \log_2 (m,n) \rceil + 1$.

2.2 Neighbor finding

The neighbor finding technique in an FTTree, Algorithm 2, works like that in a quadtree, just a new condition is added in line 2 such that, for north or south neighbors (east or west), lines from 3 to 6 are not executed for horizontally oriented quadrants (vertically oriented quadrants, in such a case, instead of VERT(PRT(v)), it will be HOR(PRT(v))).

Algorithm 2 Outputs the deepest node $u$ in $T$ whose depth is at most the depth of $v$ such that $\sigma_u$ is a north-neighbour of $\sigma_v$ and null if there is no such node.

```
procedure NORTH_NEIGHBOR ($v, T$)
1:   if $v = \text{ROOT}(T)$ then return null
2:   if CHILDREN(PRT(v)) = 4 or VERT(PRT(v)) then
3:      if $v$ is SW_CHILD(PRT(v)) then
4:         return NE_CHILD(PRT(v))
5:      if $v$ is SE_CHILD(PRT(v)) then
6:         return NE_CHILD(PRT(v))
7:   end if
8:   $u \leftarrow$ NORTH_NEIGHBOR(PRT(v), $T$)
9:   ...just like in the quadtree
```

end procedure

If $v$ happens to be the SE- or SW-Child of its parent, then for nodes of types I and III, its north-neighbour is the NE- or NW-Child of its parent, respectively. Otherwise, if $v$ is of type II, or $v$ is of type I and is the NE- or NW-Child of its parent, the algorithm behaves exactly as in a quadtree. Since at every call the algorithm spends $O(1)$ time and the depth of $v$ decreases by one, the running time is linear in the depth of $T$.

3 The HQuadtree data structure

According to Asano et al., given a regular grid $R$
of size $N \times N$, $N = 2^n$, $n \geq 0$, a space-filling curve (SFC) is defined as a numbering of the grid cells with values from $c + 1$ to $c + n^2$, for some $c \geq 0$. Authors also states that an SFC with a numbering $P$ from $c + 1, \ldots, c + N^2$ for some $c \geq 0$, is considered recursive (RSFC) if $N = 1$, or $P$ can be divided into four square RSFCs $P_n, \ldots, P_1$, following a given condition (the Hilbert space-filling curve and the Z-Order are well-known examples of RSFCs\[13\]). In this paper, the quadtree is implemented based on the Hilbert space-filling curve behavior by means of index operations and recursion. In the proposed HQuadtree (i.e., Hilbert-Quadtree; Fig.1), there is no need to store information about parent, children, and neighbor squares due to the hierarchy is given implicitly.

Fig. 1  The first three levels in a multiarray representation of the HQuadtree data structure

The starting point is an $N \times N$ regular grid, where $N = 2^k + 1, k \geq 1$, representing the height field with corners at $(0,0),(0,2^k),(2^k,0)$, and $(2^k,2^k)$, which is recursively subdivided at each step of the algorithm. Each quadtree node is represented as a segment of the Hilbert space-filling curve (“Hilbert pattern” from now on).

3.1 Construction

As shown in Fig.1, the quadtree is stored in a one-dimensional array that at, each position, stores a bidimensional array of Hilbert patterns. Each index in the one-dimensional array represents a level (from 1 to number of levels, position 0 remains empty) in the quadtree, and each matrix stores all the Hilbert patterns of its corresponding level. Algorithm 3 shows how the recursive subdivision is achieved.

Algorithm 3 Creation of an HQuadtree.

procedure CREATE_QUADTREE()
1:  levels ← $\log_2 N$
2:  $qt \leftarrow \text{array}[\text{levels} + 1]$
3:  for $i \leftarrow 1$ to levels do
4:      $qt[i] \leftarrow \text{array}[2^{i-1}, 2^{i-1}]$
5:  end for
6:  $qt[1][0,1] \leftarrow \text{Data (DOWN,EAST)}$
7:  $\text{DOWN_SPLIT}(N \gg 1, N \gg 1, N \gg 1,1)$
end procedure

The CREATE_QUADTREE procedure starts calculating the number of levels to create the one-dimensional array. Lines 3 to 5 set the corresponding bidimensional array to each level. After that the starting Hilbert pattern, Fig. 1, is created and stored in the $(1 \times 1)$ matrix at level 1, besides the direction to follow by the space-filling curve for further triangle-strip creation (i.e., the direction of the north-west Hilbert pattern in Fig.1 at level 2 is SOUTH). Both attributes, the Hilbert pattern orientation and the direction, can be represented with values from 0 to 3, needing only 2 bits per attribute. Finally, at line 7, the recursive subdivision is started by calling Algorithm 4.

Algorithm 4 Recursive division in the process of creating an HQuadtree.

procedure DOWN_SPLIT(size, r, c, level)
1:  if $(\text{size} >> 1) \neq 0$ then
2:      $r_m \leftarrow r - (\text{size} >> 1)$
3:      $c_m \leftarrow c - (\text{size} >> 1)$
4:      $r_n \leftarrow \lceil r_m / \text{size} \rceil$
5:      $c_n \leftarrow \lceil c_m / \text{size} \rceil$
6:      $qt[\text{level} + 1][r_n, c_n] \leftarrow \text{Data (RIGHT, SOUTH)}$
7:      $\text{RIGHT_SPLIT}(\text{size} >> 1, r_m, c_m, \text{level} + 1)$
8:      $qt[\text{level} + 1][r_m, c_n] \leftarrow \text{Data (DOWN, EAST)}$
9:      $\text{DOWN_SPLIT}(\text{size} >> 1, r_m, c_m, \text{level} + 1)$
10:     $qt[\text{level} + 1][r_n, c_n] \leftarrow \text{Data (DOWN, NORTH)}$
11:     $\text{DOWN_SPLIT}(\text{size} >> 1, r_m, c_m, \text{level} + 1)$
12:     $qt[\text{level} + 1][r_m, c_n] \leftarrow \text{Data (LEFT, PRT.Dir)}$
13:     $\text{LEFT_SPLIT}(\text{size} >> 1, r_m, c_m, \text{level} + 1)$
14:  end if
end procedure

The DOWN_SPLIT procedure takes four parameters: the size of the squares that will be created in the next level, and the grid row, grid column, and level of the parent square. In this procedure, a DOWN Hilbert pattern is subdivided in four patterns that are stored on the next level, Fig. 1. Before executing the procedures in lines 9, 11, and 13, the values of the row and column in the grid and in the corresponding matrix
for the south-west, south-east, and north-west children have to be calculated. The calculation is done similar to lines 2 to 5:

\[
\begin{align*}
\text{MR} & \leftarrow r + \text{(size >> 1)}, c \leftarrow c + \text{(size >> 1)}, \\
r_m & \leftarrow r_m, c_m \leftarrow c,
\end{align*}
\]

The \( r_m \) and \( c_m \) values are updated for each child. Procedures \text{RIGHT\_SPLIT}, \text{LEFT\_SPLIT}, and \text{UP\_SPLIT} are very similar to \text{DOWN\_SPLIT}. Given a valid position \( h = [l, r, c] \) of a Hilbert pattern in the HQuadtree \( Q \), the corresponding information in the height field (\( x, y \) coordinates and the height value) can be retrieved by the Eq. (1):

\[
P(x, y)_h = \left( r \ast 2^{l-1} + 2^{l-2}c, c \ast 2^{l-1} + 2^{l-2} \right) \quad (1)
\]

where \( l \) is the number of levels in the structure. In a similar way, the corners of the four quadrants represented by each Hilbert pattern can be found. Positions of the parent, children (e.g., the SW-child), and neighbors (e.g., the North-neighbor) can be calculated by the equations from Eqs. (2) to (4).

\[
\begin{align*}
\text{Parent}_h &= Q[l - 1][r, c] \quad (2) \\
\text{SWChild}_h &= Q[l + 1][r \ast 2 + 1, c \ast 2] \quad (3) \\
\text{NNeighbour}_h &= Q[l][r - 1, c] \quad (4)
\end{align*}
\]

### 3.2 Linear representation

Efficient management in the main memory of this multiarray representation may be a little cumbersome. Alternately, following this way of representing a valid position of a Hilbert pattern in the structure, a given HQuadtree with \( l \) levels can be stored in only a one-dimensional array \( A \) of size \( \sum_{i=0}^{l-1} 4^i \). For \( h \), the corresponding position \( h_a \) in \( A \) can be calculated by Eq. (5):

\[
h_a = \sum_{i=0}^{l-1} 4^i + r \ast 2^{h-1} + c \quad (5)
\]

### 4 The SBintree data structure

The process of creating an SBintree (i.e., Sierpinski Bintree) is based in the behavior of the hierarchical Sierpinski space-filling curve and is performed from an arbitrary set of points in the plane, which form a regular grid represented by a two-dimensional array \( R \) of size \( (2^n + 1) \times (2^n + 1), n \geq 2 \).

The process starts by dividing the main quadrant of the mesh with a diagonal, obtaining two triangles as a result. The division continues recursively by adding the midpoint (in the mesh) of the hypotenuse of each triangle until the desired resolution is reached. Instead of representing the structure as an RTIN, \(^1\) the SBintree is represented as a one-dimensional array that, at each position, holds a two-dimensional array of triangles, Fig. 2.

![Fig. 2 First four levels in a SBintree. Black (white) triangles corresponds to left (right) children in the hierarchy](image)

Each index in the one-dimensional array corresponds to a level in the structure, and each two-dimensional array holds the information corresponding to the nodes in the corresponding level. The first four levels are fixed patterns of triangles, and from level \( l, l \geq 3 \), the information from level \( l-2 \) is replicated four times in the level \( l \). Given \( R \), Algorithm 5 illustrates the way the process is performed.

**Algorithm 5** Creation of the SBintree.

```
procedure SIERPINSKI_TREE()
1: levels ← (n >> 1) + 1
2: stree ← array[levels].
...create patterns from levels 0 to 3...
3: for i ← 4 to levels − 1 do
4: REPLICATE_PATTERN(i − 2, i)
5: end for
end procedure
```

In lines 1 and 2, the structure is initialized, and subsequently, the fixed patterns are constructed for levels 0 through 3. The loop from line 3 completes the process of creation by multiple callings to Algorithm 6, which receives as parameters the source level to replicate to a given destiny level. In line 3, the two-dimensional array is created for the destination level. The section from lines 4 to 11 performs a replication of the triangle’s children.

**Algorithm 6** Replication of the information stored in a source level to a destiny level in the SBintree.
data structure.

procedure REPLICATE_PATTERN (src, dest)
1:  sRws = ROWS(stree[src])
2:  sCls = COLUMNS(stree[src])
3:  stree[dest] ← array[sRws << 1, sCls << 1]
4:  for i = 0 to sRws − 1 do
5:      for j = 0 to sCols − 1 do
6:         stree[dest][i,j] = stree[src][i,j]
7:      end for
8:  end for
end procedure

Each triangle requires at least 3 bits to store its type (there are eight different types according to its orientation). For each triangle in the fixed arrays from levels 0 to 3, one additional byte is needed to store the relative position of its left (the first four bits, from left to right) and right child (the last four bits). Based on this information, Algorithm 7 shows the way the calculations are performed for the left child of the given triangle at the valid position [level][rc].

Algorithm 7 Position of the left child of a given triangle.

procedure LEFT_CHILD(level, r, c)
1:  lr, lc : integer
2:  if level mod 2 = 0 then
3:      lr ← (r << 1)
4:      lc ← c
5:  else
6:      lr ← ((r >> 1) << 1)
7:      lc ← ((c >> 1) << 2)
8:  end if
9:  r ← lr + ((child >> 6) & 3)
10:  c ← lc + ((child >> 4) & 3)
11:  return stree[level+1][r,c]
end procedure

Lines for 2 to 8 states the starting point of a basic block (i.e., blocks from levels 1 and 2) in the given level. On each pair of bits from an overall of four dedicated to the left child, a relative position is stored, indicating the value to be added to each coordinate in lines 9 and 10 of the algorithm (e.g., for the triangle in [1][0,0], the values to be codified are 1 and 0, indicating that the left child is in the next row and its same column). Both processes, coding and decoding, are performed by simple logical and shift operations. Algorithms for finding the right child and parent information are very similar to Algorithm 7.

The way we represent the SBintree is the basis for the neighbor finding technique for a given triangle T, since each neighbor is in the same two-dimensional array as T in the structure. Just as in the Evans et al.\cite{2} approach, if we number the vertices of T from 1 to 3 in counter-clockwise, the i-neighbor $N_i$ of T is defined as the neighbor that does not share the vertex i of T. The relative position to T of each $N_i$ can be calculated by adding or subtracting 1 to the T coordinates, for which a code consisting in 2 bits for each $N_i$ is stored in T: one is to indicate the coordinate of T that is affected by the operation, and the other is to indicate the type of an operation (addition or subtraction). Algorithm 8 returns the same-size i-neighbor of a triangle at a valid position [level][rc].

Algorithm 8 Find the same-size i-neighbor of a triangle at a valid position [level][rc].

procedure I-NEIGHBOR(i, level, r, c)
...find op and n depending on i...
1:  \(r_r \leftarrow r + (op^{i}) \times (-1)^{n}\)
2:  \(c_r \leftarrow c + (op \& 1) \times (-1)^{n}\)
3:  return stree[level][r_r, c_r]
end procedure

Before line 1, the values of op and n (0 or 1) are calculated by using logical operations over the code of the triangle at [level][rc] depending on the parameter i. The operations \((op^{i})\) and \((op \& 1)\) determine the parameter \((r \text{ or } c)\) affected by \((-1)^{n}\).

5 Results

As stated for the quadtree,\cite{2} if the original size of a dataset is unrestricted, the resulting quadtree can be quite unbalanced. The corresponding balanced quadtree can be obtained by means of splitting the leaf nodes whose squares are neighbors that defers more than a factor of two in size. As a result, it is necessary to deal with undesired additional nodes, and therefore, additional memory is needed. The FTTree as presented in Section 3 avoid this problem, thus representing a
significant saving in memory requirements (Fig. 3).

Like the Röttger et al.\cite{9} approach, the main contribution of the HQuadtree is focused on the efficient representation in memory of the model. Given $R$ as in Section 3, a classical quadtree (represented using nodes and pointers) built over $R$ have $k + 1$ levels and
\[
\sum_{i=0}^{k} 4^i \text{nodes.}
\]
Besides, at least two bytes per height value in the regular grid, for each node in the quadtree information about error, descendants, parent, and neighbors have to be stored. That is why the size of each node $q$ is at least
\[
S_q = s_p (d_q + v_q + f_q + e_q)
\]
with $s_p$ the size of a pointer to a node, $d_q$, $v_q$, $f_q$, and $e_q$ the number of descendants, the number of neighbors, the parent, and the associated error, respectively.

Besides, at least two bytes per height value, for each Hilbert pattern, only four bytes per error and four additional bits, two for the kind of Hilbert pattern and two for the direction to follow in the triangle-strip generation, are needed. Calculations done for some regular grids show that our approach takes about 6% of the total memory needed by a classical representation using nodes and pointers.

Due to the fact that in our quadtree representation, each Hilbert pattern corresponds to nine points in the regular grid and to four quadrants in the hierarchical model, we only need $k$ levels. Therefore, the number of bytes needed for the Hilbert patterns stored is
\[
\sum_{i=0}^{k-1} a^r \cdot \frac{1 - r^{k-1}}{1 - r} ; a = 1, r = 4
\]
that is lower (approximately 25%, Fig. 4) than the $n^2$ bytes needed by the RQT of Röttger et al.\cite{9}, a well-known algorithm because of its extremely low memory consumption.

Both the RTIN\cite{1} as the SBintree data structure addresses the problem of the efficient representation in memory of the surface data. The running time complexity of the main algorithms is the same in both structures, in addition to the similarity in the size of each structure in main memory. For each data point in the regular grid, in addition to the corresponding coordinates, 4 bytes to store a single error measure are needed. For each triangle in the hierarchy, 4 bytes are enough to store the information for parents, children, the type of each triangle (3 bits), and the information required for neighbor finding (6 bits, two for each direction).

Triangle-strip generation based on space-filling curves has been widely used to efficiently visualize multiresolution triangulations extracted from hierarchical data structures.\cite{8} In Algorithm 4, once each Hilbert pattern is created, an object space error like in the Röttger et al.\cite{9} approach is calculated (e.g., after line 9) and stored for further extraction of the multiresolution triangulation. Due to the fact that our recursive subdivision starts with a down-oriented Hilbert pattern, the starting (ending) point of the space-filling curve is in the north-west (north-east) region of the surface (Fig.6).

The starting point of the space-filling curve is found by recursively descending the north-west region of the quadtree until a marked (indivisible) quadrant is reached. Then, following the directions stored in the structure, each Hilbert pattern is triangulated in a continuous way. A similar triangulation scheme can be applied to the SBintree.

For visualization purposes, some screenshots were taken from an application running in a system with a 2.53-GHz Core 2 Duo E7200 processor and a 512-MB Nvidia GeForce 9400 GT graphics card. To show the effectiveness of our structures (Figs.5(a),
Yusnier Valle, et al./ An Efficient Representation of Quadtrees and Bintrees ...

Fig. 5  A textured view of the Grand Canyon (a) and Puget Sound (b) height fields with the underlying triangulation and curve. A multiresolution triangulation of the 4K×4K Puget Sound (c), Washington, USA, dataset

5(b)), we use classical sample data from the Grand Canyon and Puget Sound areas, USA, with the elevation data artificially scaled in order to exaggerate the elevation changes. Because of the low memory requirements of our approaches, Fig.5(c) shows a view of a dataset with millions of points completely represented in main memory.

6  Conclusion

In this paper, we have presented a novel approach for the representation of quadtree (bintree)-based hierarchies of quadrants (triangles). The performance of the FTTree is very similar to the quadtree\(^2\) data structure, except that it can be built over size-unrestricted domains without the addition of fictitious point. Even when the running time bounds of the main algorithms are identical in both data structures, some tests accomplished for different tolerance thresholds have shown that the overall number of nodes in the FTTree is lower.

The multiarray representation of the HQuadtree data structure guarantees extremely low memory requirements to store a hierarchy of quadrants given a regular grid. Because of the distribution of the Hilbert patterns, the hierarchy is implicitly expressed in the structure. As a result, the HQuadtree significantly overcomes the RQT of Röttger et al.\(^9\) by using approximately 25% of the overall memory of this very well-known approach. For the SBintree, further research is needed to improve the indexing scheme to implicitly represent the hierarchy of triangles.

Future work involves the development of a real-time terrain visualization engine based on the proposed data structures, taking advantage of their representation schemes and underlying space-filling curves to efficiently generate triangle-strips for fast rendering of multiresolution surfaces. We are also working on a compact encoding of Hilbert patterns as triangle fans for fast decoding by geometry shaders on the GPU.

References

[1] Evans W S, Kirkpatrick D G, Townsend G (2001) Right-triangulated irregular networks [J]. Algorithmica, 30(2): 264-286
[2] de Berg M, van Kreveld M, Overmars M, et al. (2000) Computational geometry: algorithms and applications(2nd Edition) [M]. Beilin: Springer
[3] Kreveld M J V (1997) Algorithms for triangulated terrains [C]. SOFSEM ’97: Proceedings of the 24th Seminar on Current Trends in Theory and Practice of Informatics, London
[4] Floriani L D, Marzano P, Puppo E (1996) Multiresolution models for topographic surface description [J]. The Visual Computer, 12 (7): 317-345
Notes to Contributors

Contributions are welcomed on one of the following subjects or in related areas:

- GIS
- Geodynamic
- Physical geo-surveying
- GPS
- Geo-surveying
- Engineering surveying
- RS
- Photogrammetry
- Mapping apparatus
- Cartology
- Graphics

The main text should be preceded by the abstract, followed by key words. Full references should be listed in the order of the citations in the text under the heading "References", guided by standard publication format. The name of the fund and project series number for articles of funded projects should also be given.

Contributions submitted for publication will be peer reviewed, i.e., the entire articles will be examined by qualified experts independent of the authors before publication.

The authors assume sole responsibility for their dissertations. Canned theses are not allowed. Once the paper is submitted, the editors are authorized to make necessary literate processing. Authors who receive no notice in one month after submission may contact us for inquiry.

Feel free to contact us.

E-mail: gsis@whu.edu.cn
   sjjs.2008@gmail.com
   Tel:+86-27-68778045