ScanGIS'2001: Distributed Handling of Level of Detail Surfaces with Binary Triangle Trees

Rune Aasgaard[1] and Thomas Sevaldrud[1]

SINTEF Applied Mathematics
PO Box 124, N-0314 Oslo, Norway
{tse, raa}@math.sintef.no
WWW home page: http://www.math.sintef.no/

Abstract. A system for efficient distribution of a Level-Of-Detail (LOD) triangulated surface in a client-server environment is described. The data structures are divided into server oriented structures for storage and retrieval, and client structures for efficient rendering of a subset of the surface. The client structures are based on a binary tree of right isosceles triangles which are split recursively until a predefined error tolerance is met. The binary triangle tree is mapped to a quad tree which is used for storing and accessing the data on the server.

1 Introduction

Large terrain models are increasingly more available for public use. Earth covering data sets with 1km resolution are distributed freely on the Internet, and national scale data sets with resolution of 30-100m can be purchased for many countries. Even a 30m resolution data set covering major parts of the world is in preparation (http://www.jpl.nasa.gov/srtm/).

The data volumes of such terrain models are very large, and in applications, as for example image generation, using the whole data set at full resolution may be unnecessary because of the limited resolution of the computer display.

One is seldomly interested in the entire world at full resolution. Instead one could use a coarse approximation of the world for finding the area of interest, then adding further data to the required region until the desired level of detail is reached. Ideally one should use the highest data density close to the viewer and in areas with much variation in the surface.

In a client-server environment the client resources, server resources and the capacity of the communication channel may be limiting for the performance of the application. Care must be taken to balance the usage of resources in the system.

Most of todays 3D rendering hardware is optimized for drawing triangles. Triangles are also a flexible building block for continuous surfaces, capable of handling complex geometries. On the other hand, triangles require many pointers or references to express the topological relations in the triangulated network.

Using a triangulated surface with a strict tesselation algorithm simplifies the problem slightly as there is a simple relationship between position and connections in the surface.

Our work is based on the ROAM algorithm [1], but we make certain optimizations of the split/merge priority data structures enabling us to more efficiently determine whether a triangle should be split or merged. These optimizations are based on ideas proposed by Jonathan Blow in [4], and one contribution of this paper is a detailed description of our interpretation and implementation of these ideas. The other main contribution is to describe a client/server architecture for progressive transmission of large amounts of terrain data, a.k.a "streaming".

2 Previous Work

Not much has been presented in detail on the topic of client/server-based terrain visualization, however a few data structures have been presented which are well suited for streaming. We have selected some of the work presented on these data structures. The following articles all give good descriptions of the data structures mentioned, as well as extensive references to the origins of these structures.

Hoppe, [7] presents a TIN (Triangular Irregular Network) model based on his work on *View-Dependent Progressive meshes* (VDPM). This model consists of a hierarchy of vertex split and edge collapse operations. By using a sequence of such operations on a base mesh, one can obtain a terrain model with the desired detail for visualization. In a client/server-structure one could send vertex split records to the client on demand, thus adding detail to the mesh incrementally.

Lindstrom et. al, [5] presents a model based on regular subdivision of the mesh, thus eliminating the need for explicit storage of the (x, y)-positions of the triangulation nodes, at the cost of using some more triangles (about 30-40%, according to Hoppe) to obtain the same accuracy. This model is also hierarchical, built around a quad tree data structure, with 9 potentially enabled vertices per quad tree node (center point, corners and edge midpoints). By enabling and disabling these nodes according to a set of rules, one will obtain a triangulation with a mesh density varying according to the tolerance function of the application.

Duchaineau et. al, [1], Gerstner, [2], and Turner, [6] all use the Binary Triangle Tree structure which has similar geometrical properties to the quadtree triangulation used by Lindstrom. Whereas Turner and Gerstner traverse the binary tree top-down for each frame to construct a triangulation, Duchaineau et. al. seek to reduce this traversal time by taking advantage of the frame-to-frame coherence of the surface. To achieve this they only make modifications relative to the current triangulation by coarsening or refining the mesh.

The Binary Tree Triangulation structure is very simple, and well suited for progressive transmission since the triangulation may be built in top-down fashion starting with the coarsest level. We have therefore chosen this data structure combined with an alternative system for taking advantage of the frame to frame coherence.

3 Data Structures

This section deals with the central data structures of our algorithm; the Binary Triangle Tree (BTT), the Grid Quad Tree (GQT) and the Error Tolerance Iso-surface Tree. The first structure represents the triangulation itself, the second a compacted version of the data needed to construct the BTT and the latter form the criteria for refinement and decimation of the current mesh.

In a distributed environment the BTT and error sphere hierarchy are stored on the client in a form that is convenient for the viewing system. Connectivity must be maintained in the active section of the triangle hierarchy to identify possible recursive splits and to support generation of connected triangle sets (strips and fans) for efficient rendering.

On the server side this connectivity is not needed, instead the structure must be optimized for storing and searching large data volumes. Here a variation of the quad-tree can be used, where the elevations and approximation errors are associated with quad-tree nodes.

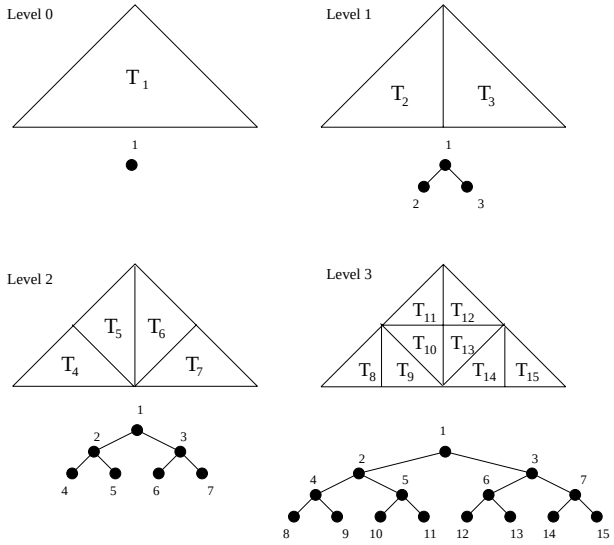

Fig. 1. The four first levels of a BTT

3.1 Binary Triangle Trees

A somewhat less-known relative of the Quadtree, the Binary Triangle Tree (BTT) has been used in several recent works on terrain visualization such as

[1], [2] and [3]. This structure is well-suited for top-down construction of a continuous triangulation surface with varying level of detail. Both [6] and [3] give good introductions to the BTT structure, so we will only briefly describe it here.

A binary triangle tree is a structure in which a parent triangle is split into two child triangles recursively. Figure 1 shows the first four levels of a binary triangle tree, and the corresponding triangulations.

Please note the indexing of the triangles. Due to the binary tree structure, we can index triangles such that triangle with index i will have two children with indices $2i$ and $2i+1$. This can be used for example for linear storage of the triangles in an array.

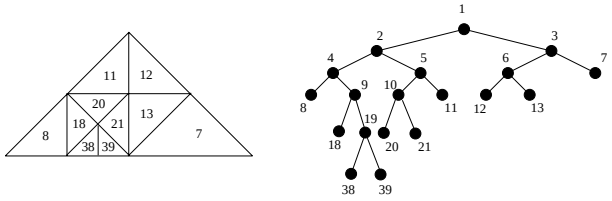

Fig. 2. An adaptive tesselation and its triangle tree.

We see that each new level is generated by splitting all the parent level triangles. These splits are done by inserting a new vertex at the midpoint of the hypotenuse of each triangle. By traversing this tree until certain refinement criteria are met we end up with an adaptive tesselation as shown in figure 2.

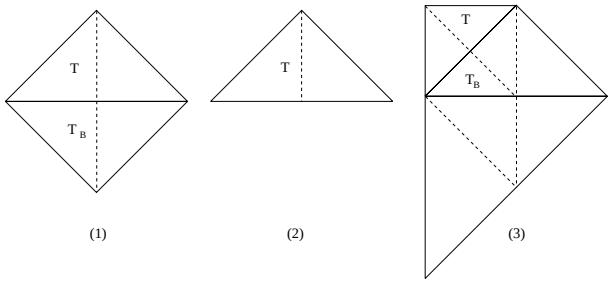

Fig. 3. Three cases of triangle splitting.

To avoid illegal triangulations with T-junctions we have to ensure that each triangle has neighbors on the same level as, or at most one level finer or coarser than itself. We define the *base neighbor*, T_B to be the neighbor triangle across

the hypotenuse of triangle T. If T and T_B are on the same level, we say that they form a *diamond*. We now have three cases when attempting a triangle split:

1. T and T_B on same level (diamond): Split both triangles.
2. T has no base neighbor (on a border): Trivial, split T.
3. T is one level finer than T_B: Force split T_B before splitting T.

These three cases are illustrated in figure 3.

Note that in figure 3, case (3), the base neighbor of the base neighbor is also illegal, so that we have to do two forced splits before we can split T, thus a forced split should be done recursively.

Conversely, the child triangles from the splits in the configurations (1) and (2) in figure 3 can be merged into coarser triangles if that should be required. Non-diamond triangles have to wait until the configuration allows merge.

Error Function To perform the adaptive tesselation we need an error indicator function, $\Delta(T)$. Assuming that the terrain surface is a functional surface, $z = f(x, y), \{x, y, z\} \in \mathbb{R}$. Let (x_l, y_l, z_l) and (x_r, y_r, z_r) be the two vertices of the hypotenuse of triangle T. The distance between the midpoint of the hypotenuse and the actual surface can now be used as an error measure. Let $\{T_C\}$ be the set of triangles consisting of T and all its children, i.e all triangles in the subtree of which T is root. One simple error function will then be

$$\Delta(T) = \max_{t \in T_C} \{\delta(t)\}, \tag{1}$$

where $\delta(T)$ is the distance between the hypotenuse midpoint and the surface,

$$\delta(T) = \left\| \frac{z_l + z_r}{2} - f\left(\frac{x_l + x_r}{2}, \frac{y_l + y_r}{2} \right) \right\|. \tag{2}$$

By using this error function we are guaranteed that the triangle error is monotonously decreasing as we descend into the tree. This is useful since we now know that if a triangle is within our given tolerance, all of its children must also be within this tolerance.

3.2 Error Tolerance Isosurfaces

The original ROAM algorithm [1] used two priority queues to decide when a triangle should be split or merged. These queues, however need to be sorted each time the triangulation is modified. This can lead to an increasing delay if we have a fast moving camera. Each time the queues are sorted we get a delay causing the camera to travel further in the meantime, thus forcing us to make more changes to the mesh the next frame. This again leads to a new sorting of the queues, perhaps with even more changes than last frame, effectively grinding the visualization to a halt.

We have chosen another approach, which was first loosely described in [4]. If we simplify things a bit, the priority of a triangle is a function of the distance between the observer and the triangle, and an error measurement for the triangle.

Now, instead of compressing all these parameters into one priority value, we create an isosurface which is such that all points within the surface are close enough to the triangle that the triangle error is larger than the current tolerance. I.e if the observer is within the surface, the triangle needs to be split, and if the observer is outside the surface, the triangle should be merged. For simplicity we choose a sphere for our isosurface and we get the following equation

$$x^2 + y^2 + z^2 = r^2 \tag{3}$$

where

$$r = \frac{\Delta(T)Y}{\lambda_y}. \tag{4}$$

Here, λ_y is the vertical field of view angle of the camera (in radians), and Y is the pixel resolution in y-direction of the screen.

By centering these spheres at the potential split vertices of the triangles, we may now build a hierarchy of priority spheres. This hierarchy is a tree of spheres where each sphere has as its children all spheres which are completely enclosed by the parent sphere as shown in figure 4. As you will see from this figure, we

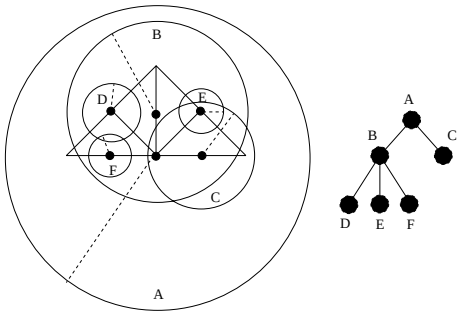

Fig. 4. Sphere tree and the corresponding triangulation.

actually only use one sphere for each pair of triangles, since the two spheres will be located in the same place, only with different radii. When one of these triangles is split, the other will also have to be split to avoid cracks in the surface. Thus we only use the sphere with the largest radius of the two.

Looking at this figure we can tell immediately that if the camera is located outside of sphere A, we don't have to check any of the other spheres, the triangles corresponding to this sphere should be merged anyway. On the other hand, if the camera should be inside sphere F, we also have to split A and B.

3.3 Grid Quad Tree

In the Grid Quad tree the grid cells of the original DEM are represented by nodes in a quad tree. Each level in the tree represents a successive subdivision of the terrain surface, where nodes at a deeper level gives more detailed data.

A hierarchical triangulation using binary triangle trees is shown in figure 5.

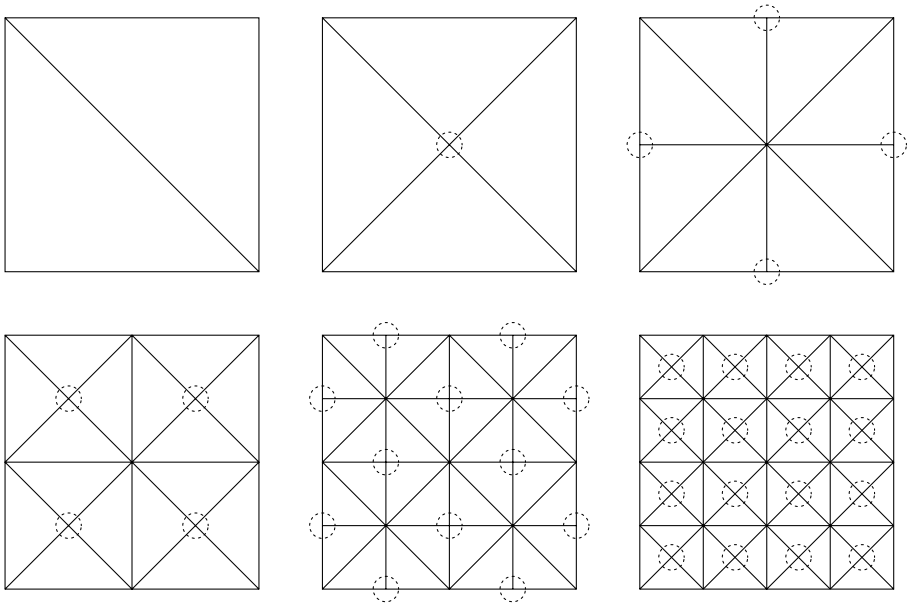

Fig. 5. Levels of triangle splits. Vertices added in each split level are marked.

The vertices in the triangulation tree are mapped to the quad tree nodes by clustering three and three vertices in each quad tree node as shown in figure 6. The planar position of a vertex in the triangulation is used as an index into the quad tree. In this way the server quad tree is handled as a large virtual grid indexed on the vertex positions.

The storage structures ensure that data items which are closely related with regards to position and resolution, will also be stored near each other in the data structures.

The quad tree is filled with elevation values in a recursive procedure that descends into finer and finer detail until the limiting resolution of the data source is reached. The same triangular subdivision pattern as in the BTT is is used, starting with a pair of triangles covering the whole data domain.

The triangle approximation errors are computed as shown in section 3.1 and stored in the quad tree as the recursive procedure backtracks. Data is only stored

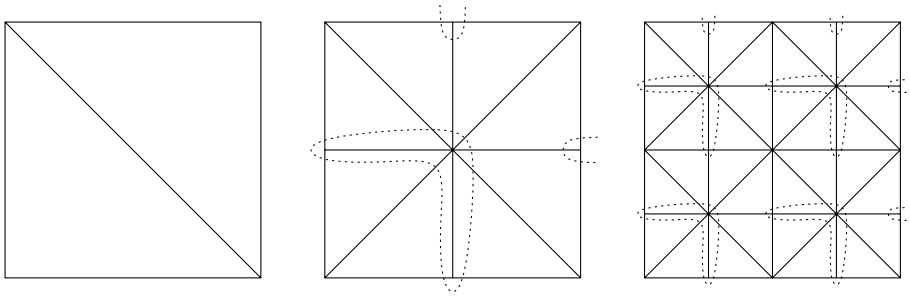

Fig. 6. Triangulation related to the quad tree. Vertices added in each quad tree level are marked.

if at least one of the triangle errors for a split are larger than a given tolerance, to conserve storage space in areas with low variation.

When a triangle pair is split, four new triangles are created (as shown in figure 3), as well as a new split vertex (naturally, with the exception of border triangles). Thus, for each split, four triangle errors and a elevation value is stored.

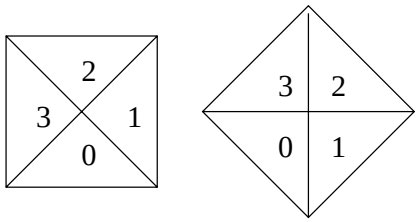

Fig. 7. Direction enumeration of child triangles around split node.

The quad tree has no implicit knowledge of the split pattern of the BTT. To associate each of the four error values with the correct triangles, the relative positions of the triangles are used to create an enumeration as shown in figure 7.

4 Architecture and Implementation issues

As was described in section 3 this work is aimed at creating a client-server environment for hierarchical terrain models. The client program contains and maintains a subset of the whole terrain model as a Binary Triangle Tree. The validity of the local terrain model is continously evaluated by traversing the Error Tolerance Isosurface Tree. Refined data is fetched from the server where the data is stored in a Grid Quad Tree.

The client program is programmed in Java and the server in C++. The client uses the Java3D extension for generating the graphics. The communication channel uses binary data over simple TCP/IP stream sockets. All elevation data queries and replies are transfered as 2 and 4 byte signed integers, in "network byte order".

During the work we encountered several practical problems and found some solutions that we would like to share in the following.

4.1 Updating the Triangle Tree

When a pair of triangles is split, one new node and four new triangles are created. The planar coordinates for the split node are readily computed from the coordinates of the two end nodes of the hypotenuse, as already shown. However, the new node also requires an elevation value, and the new triangles will require triangle errors for the computation of error spheres.

A query is sent to the server, using the planar position of the node as an index into the server quad tree. The elevation value and four triangle approximation errors are returned and used for updating the triangulation. As there may be a considerable time delay in the query-reply communication loop the client program should handle the replies asynchronously. To avoid sending many very small packages the queries for one cycle of updating are assembled into one query set. Seveal small query sets from subsequent cycles can also be aggregated before sending.

When a triangle leaves the active triangulation by merging of its parents, it may be retained in the triangle tree in case it should be required for a later refinement. Children of merged triangles get a time stamp, and are delteted in a least-resently-used scheme.

4.2 Handling the graphic data

Java3D uses a retained mode scene graph structure. All graphical data has to be represented by objects in the scene graph. The scene graph is traversed asynchronously by a built-in render thread. Updating the scene graph requires synchronizing with the render thread and may be very time consuming. Therefore all scene graph updates are assembled and performed in one operation when the render thread is stopped between displaying frames.

A quad tree of "quadratic" tiles are used as containers for the graphic data. Each tile covers a pair of triangles in the BTT. Usualy, this triangle pair is internal to the BTT and has sub trees of child triangles. The child triangles of a visible tile are traversed to generate graphic arrays for the tile.

A tile can also contain a georeferenced image for rendering of textured terrain. The resolution of the texture images and the required resolution at the tile position determines on which quad tree level the data is displayed.

4.3 Coordinate system

The graticule (meridian/parallel net) is often used for creating a regular tesselation of the globe. This is a very simple solution that may be sufficient for many purposes. Several global elevation data sets (notably DTED and derivatives) are structured this way. However, this scheme leads to very un-square cells as we approach the poles.

In this project another projection has been tested to relate the cells of the quad tree with the graticule. The cells are scaled to retain an aspect ratio of near 1 over most of the globe, while still avoiding the asymptotic behaviour of conformal map projections when nearing the limits of the defined area. This method will be discussed in a later article.

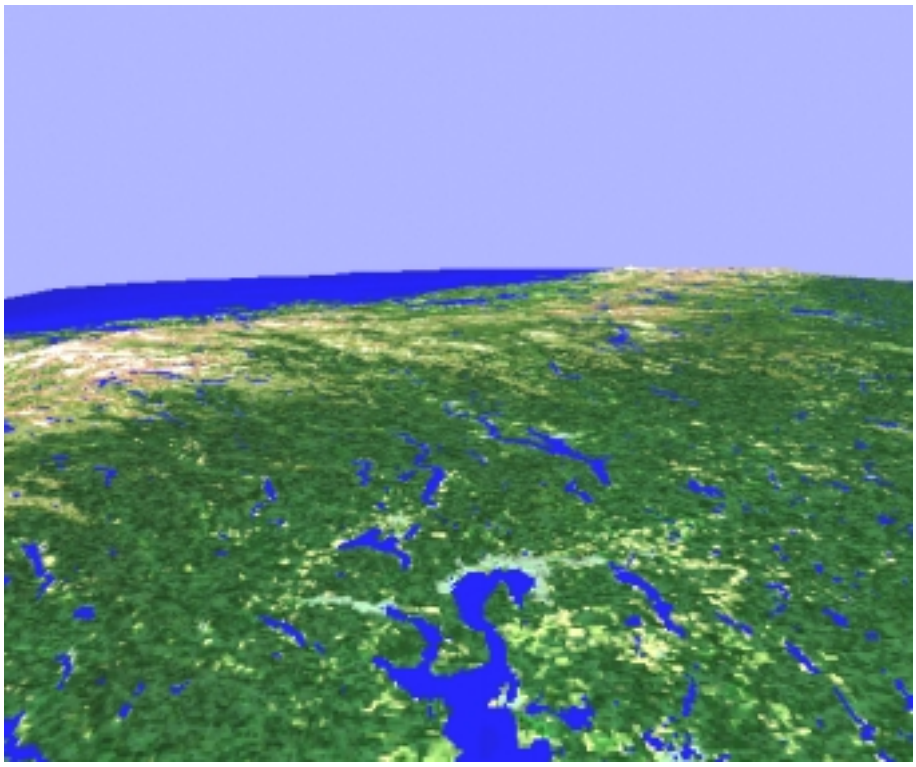

Fig. 8. The textured TerrainBase dataset showing the Oslo area with major parts of southern Norway.

5 Tests and Results

Two free global datasets where loaded into the system to test the metods:

1. **GLOBE:** a 30 second grid of land elevations (`http://www.ngdc.noaa.gov/seg/topo/globe.shtml`).
2. **TerrainBase:** a 5 minute grid covering both land elevations and ocean depths (`http://www.ngdc.noaa.gov/mgg/global/seltopo.html`).

To stay within the system limit of 2GB files on our current linux file system, only Europe, half of Asia and Africa north of equator were covered by the *GLOBE* dataset. A complete global coverage will be prepared when we have upgraded to 64 bit file systems. The *TerrainBase* dataset needed only a 100MB quad tree file. As the quad tree database loads data on demand and unloads on a least-recently-used sceme the server performance is independent of the quad tree file size.

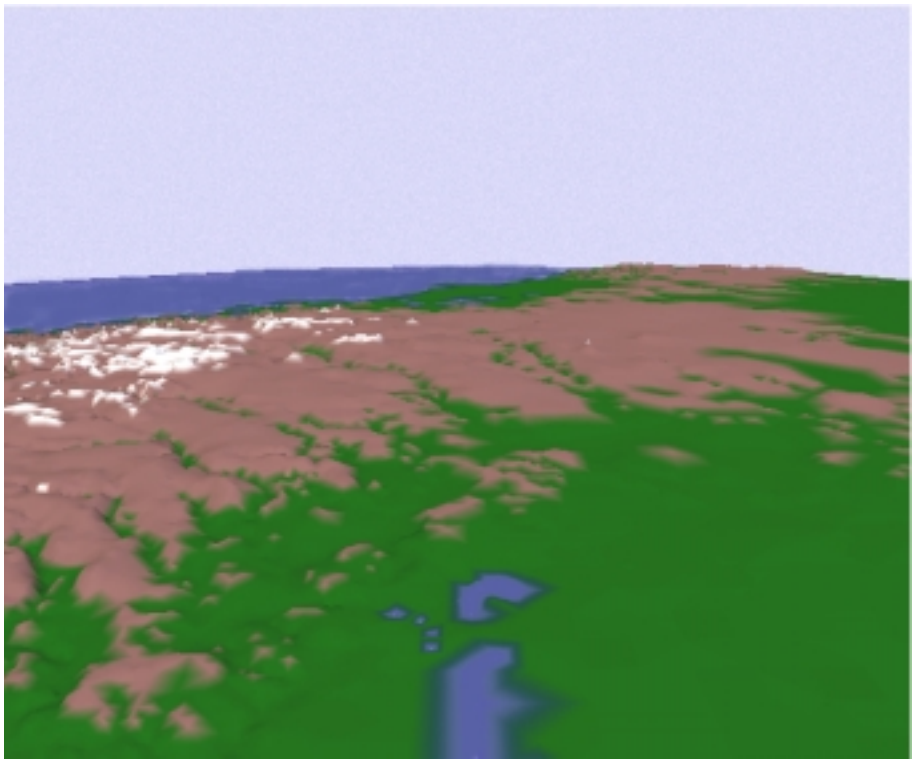

Fig. 9. The globe dataset.

The *GLOBE* dataset was stored with the new coordinate system only, while the *TerrainBase* dataset was stored with both the new coordinate system and in a simple latitude/longitude grid. As we already have a image texture coverage in a lat/lon grid this dataset was used for testing rendering of textured terrain.

The server program was instrumented to log the data communication. The client was instructed to load data to generate perspective images at given positions and the data volume needed for the client to build an adequate terrain model was counted.

For each vertex query two 32bit coordinate values are sent from the client, and one 16bit elevation and four 16bit deviations are received. Totally 8 bytes from the client and 10 bytes returned.

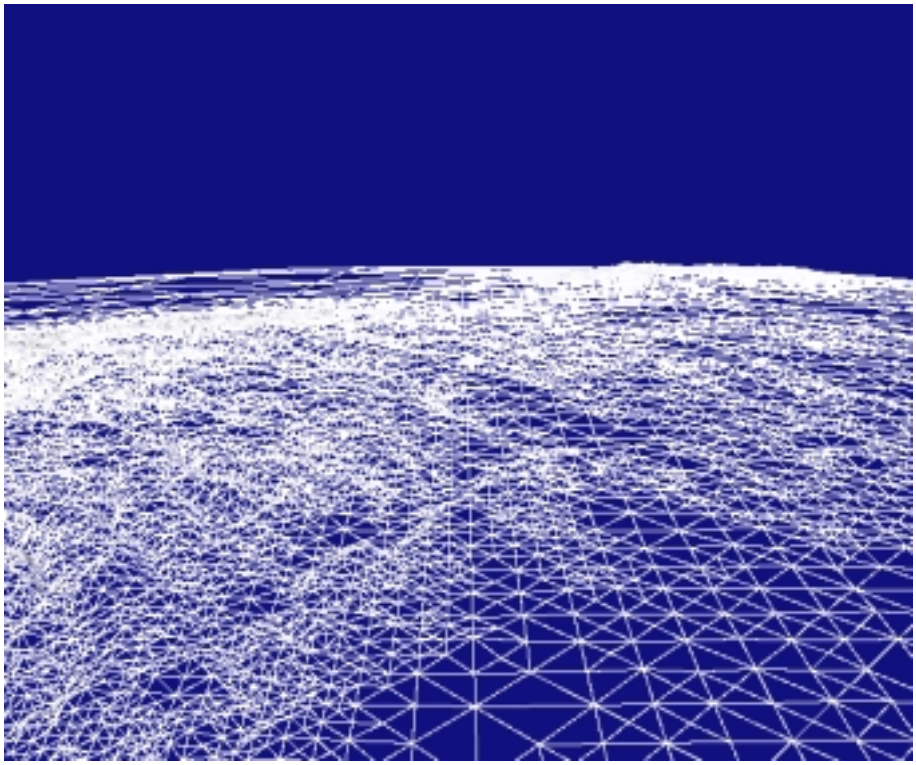

Fig. 10. Triangles from the globe dataset.

Test 1: Viewpoint at position 58.0° N 10.5° E elevation 50000m, looking north, having an elevation exaggeration of 5 and a terrain deviation tolerance of 5 pixels projected into the image plane.

GLOBE: The total number of queries was 9475, which corresponds to 75800 bytes sent from the client and 94750 bytes returned (pluss a minor amount of header data).
TerrainBase: Total number of queries was 1977, which corresponds to 15976 bytes from the client and 19770 bytes received.
TerrainBase with texture: Total number of terrain queries was 1850, which is similar to the previous test. However, 1363968 bytes of texture image data was transfered!

Test 2: Viewpoint elevation increased to 100000m, keeping the other parameters as in the previous test.

GLOBE: 9110 queries.
TerrainBase: 882 queries.
TerrainBase with texture: 2322 queries, 1245184 bytes texture images.

The large increase in the number of queries when introducing textures probably arises from the fact that the textured database is based on a more primitive coordinate system. The advantages of the better coordinate system starts to show as we can see further north, and are more influenced by the thinner and narrower cells.

6 Conclusions and Further Work

As can be seen from the tests the amount of transferred terrain elevation data is totally insignificant compared to the texture data. However, on the client side the terrain data expands to a large and complex data structure that has a large impact on the client system performance.

For streaming textured terrain data models, it may be worthwhile to take a look at more efficient streaming of the texture image data as well. Several methods for doing this has been presented in the literature, many of them based on wavelet methods, which we find interesting.

Currently it seems like the main bottleneck is in inefficiencies in the client program. As it is using Java as a programming language the opportunities for optimalization are more limited, especially with regards to memory handling. We may therefore want to create a new client in C++ using OpenGL which gives us more control over the 3D rendering process and data structures. On the other hand, for a client-server system one must remember that the client may be run on a multitude of platforms and it may be much more difficult to maintain and distribute a multi-platform C++ client program.

References

1. Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C. and Mineev-Weinstein, M.B.: ROAMing Terrain: Real-Time Optimally Adapting Meshes, in *Proc. Visualization '97*, 1997.

2. Gerstner, T.: Multiresolution Visualization and Compression of Global Topographic Data. in Proc. *Spatial Data Handling 2000*, P. Forer, A.G.O. Yeh, J. He (eds.), pp. 14-27, IGU/GISc, 2000.
3. McNally, S. and Blow. J.: Two Advanced Terrain Rendering Systems. in Proc. *Game Developers Conference 2000*, 2000.
4. Blow, J.: Terrain Rendering at High Levels of Detail. in Proc. *Game Developers Conference 2000*, 2000.
5. Lindstrom, P., Koller, D., Hodges, L.F., Ribarsky, W., Faust, N. and Turner, G.: Real-Time, Continous Level of Detail Rendering of Height Fields, *Computer Graphics (Proc. SIGGRAPH '96)*, 1996.
6. Turner, B.: Real-Time Dynamic Level of Detail Terrain Rendering with ROAM. *Game Developer Magazine*, April 2000.
7. Hoppe, H.: Smooth view-dependent level-of-detail control and its application to terrain rendering. *IEEE Visualization 1998*, October 1998, pp 35-42.