UNIVERSITE D'ORLEANS

Faculté des Sciences

LIFO

Laboratoire d'Informatique Fondamentale d'Orléans
4, rue Léonard de Vinci, BP 6759
F-45067 Orléans Cedex 2
FRANCE

Rapport de Recherche

www : http://www.univ-orleans.fr/SCIENCES/LIFO/

Large Multi-modal Terrains Rendering in Multi-display Environments Driven by a PC Cluster

Rapport N^o **RR-2005-01**

Valérie Gouranton[1*] Sébastien Limet[1†] Souley Madougou[1,2‡]
Emmanuel Melin[1§] Cyril Nortet [1¶]

[1]LIFO
Laboratoire d'Informatique Fondamentale d'Orléans
Orléans, FRANCE

[2]BRGM
Bureau de Recherches Minières et Géologiques
Orléans, FRANCE

April 28, 2005

Abstract

Large terrains rendering at interactive rates is of increasing interest in many areas. In Geographic Information Systems (GIS), for example, the elevation data are often associated to various other information such as rivers, drill-holes, roads, mineral densities and so on. The issue is then to find the right levels of detail algorithm that both scales well with raw data and preserves the other important features. As a contribution to this research, we present a generic mipmap method that intrinsically implies implementing a morphing support and which is applicable to any kind of per-vertex data. The method is integrated in a parallel framework for geometry-driven levels of detail rendering of large height fields. Relying on recent advances in PC cluster computing, we distribute in a consistent manner both geometry and extra data among cluster nodes. The previous levels of detail algorithm is combined with our mipmap method to process them in parallel. To optimize the use of the interconnecting network bandwidth a compact data structure is defined for data transfer. As a consequence, the framework sustains smooth interactive appearance-preserving rendering of gigabyte-sized terrain models.

*e-mail: Valerie.Gouranton@lifo.univ-orleans.fr
†e-mail: Sebastien.Limet@lifo.univ-orleans.fr
‡e-mail: Souley.Madougou@lifo.univ-orleans.fr
§e-mail: Emmanuel.Melin@lifo.univ-orleans.fr
¶e-mail: Cyril.Nortet@lifo.univ-orleans.fr

1 Introduction

Nowadays, interactive 3D graphics rendering is used in a broad range of human activities: scientific and information visualization, CAD (Computer Aided Design), GIS, Virtual Reality. Indeed, visualization allows scientists to extract knowledge from real or simulated data leading to a deep understanding of many phenomena. On one side, huge raw data sets are frequent in many areas: high resolution range scanning [20], scientific simulation [22], terrains [27]. On the other side, advances in graphics hardware and software have shifted end-user expectations in image quality and relevance. In many fields such as GIS, models often consist in data of various origins: topographic, infrastructural, geological, hydro-graphic. These different data are mixed in order to obtain a better representation of the model allowing a more precise interpretation. Thus, for such area today's scientific visualization needs can be expressed as follows: ensure an interactive rendering in spite of the data size while preserving, with enough detail, essential features in the rendered model. Whereas a large variety of methods are proposed for addressing the first aspect, few methods process both aspects.

Before the advent of level of detail (LOD) algorithms, the only solution for visualizing complex 3D models or high-resolution images was to resort to high-performance rendering systems such as SGI InfiniteReality [23]. These systems consist in high-end computers with several tightly-coupled processors. The main memory of these architectures amounts in tens of gigabytes which makes them good candidates for large datasets rendering. However, the main drawbacks of these custom-designed systems are their lack of extensibility and their price. To address large data sets rendering on low-end hardware, earlier LOD algorithms have been designed [17, 21, 9] (see [16, 24, 13] for detailed surveys). These algorithms being of in-core type, they require storing the entire full resolution mesh in main memory. Many of today's data sets are several orders of magnitude bigger than even a high-end workstation memory, making in-core algorithms impractical. To solve this issue, several techniques for out-of-core LOD have been developed recently [7, 5, 18, 26, 11, 29]. The problem of such mechanisms is the latency due the external memory access. This is often solved by implementing complex prefetching mechanisms. However, any extra time spent in LOD algorithm inexorably results into frame rate degradation which leads to interaction disruption. Thus, for interactive rendering of huge data sets, both LOD techniques and high-performance computing are required. An alternative to external memory methods is then to adapt LOD algorithms to inexpensive parallel architectures such as PC clusters. PC clusters present many advantages over custom-designed high-end systems: a favorable price-to-performance ratio, flexibility, extensibility, ubiquity. Using these architectures, the two limiting factors, CPU and memory, can be solved by developing efficient parallel algorithms that evenly partition model data and LOD computations workload among cluster nodes. Their main disadvantage is the interconnecting network, although currently efficient solutions emerge: Gigabit Ethernet, Myrinet, Infiniband. This limitation can be discarded by minimizing the induced communication overhead either using culling or implementing efficient compression techniques. Several work based on this approach [12, 10, 4, 14] showed promising results.

Most of previous work only address the geometry data, ignoring surface appearance information, for example color and lighting parameters. However, real-world data sets are often composed of several layers representing thematic or surface appearance information. Few algorithms propose an appearance-preserving support. The paper of Hoppe [17] is among the pioneer work. It defines an optimization procedure of which goal is to preserve not only the geometry of the original mesh but also its overall surface appearance attributes. The procedure measures a maximum or average deviation of these attributes over the model. More close to our work is the mipmapping approach described in [8]. The method initially parameterizes the input polygonal mesh. From this parameterization, texture and normal maps are created. A texture deviation metric is then used to guarantee that these maps shift by no more than a user-specified number of pixels on the screen. Cignoni and co-workers [6] present a general method for preserving detail on simplified meshes. The initial detail is encoded in texture and bump maps. The preservation is performed as a post-simplification process and it consists in sampling each simplified mesh face to build an associated triangular texture patch. These patches are next packed into a standard rectangular texture. Other approaches such as [3, 19] deal with integrating texture and geometry LOD. The main differences of our method compared to these approaches are: first, it processes any per-vertex data and second, it does not require out-of-core schemes for attribute data as they are distributed.

Our work aims at defining a distributed framework for a generalized LOD rendering of large terrains. It uses the parallelization scheme defined in [14] where only elevation data were managed. We still rely on recent PC clusters high-performance computing capabilities to distribute terrain data among cluster nodes. But now, the distribution concerns not only geometry data but also surface appearance attributes and more generally, any per-vertex data. In real-world applications, different types of information are usually encoded in texture images. For large-scale terrains, these textures are typically huge and often do not fit in video or even in system memory. A comprehensive LOD framework must provide support for both geometry and texture data. We do this via a generalized mipmapping scheme. Texture mipmapping [28] is a widespread technique used to reduce aliasing while mapping a texture at different scales. Mipmapping is a quadtree-like multi-resolution representation for texture data and so particularly fits to the quadtree-based LOD algorithm [25] used in our framework. By extending this technique to other per-vertex data such as normal vectors, we are able to provide a general distributed framework for LOD rendering of multi-modal terrain models. In addition, the framework benefits from recent developments in cluster middlewares for real-time applications such as FlowVR [1]. FlowVR dramatically eases distributed real-time applications implementation. It simplifies many communication and synchronization details generally hard to manage in parallel libraries such as MPI. The major contributions of this paper over our previous work include:

- **Integrating** levels of detail for both geometry and texture data in one parallel framework
- Providing **a generalized mipmapping** support applicable to any per-vertex data
- Providing **a morphing support** addressing the visual artifacts due to the mipmap-

ping

- **A novel data encoding** aiming at reducing the network bandwidth consumption
- Using **recent software development tools** for efficient parallel real-time applications deployment

The rest of the paper is organized as follows: in section 2 we describe the general framework overview. Color mipmapping and color morphing are explained in sections 3 and 4 respectively. We detail the generic mipmapping scheme in section 5 where some applications are also discussed. The networked data compression is given in section 6. Benchmark conditions and results are detailed in section 7. Finally, in section 8, conclusions and perspectives are drawn.

2 Framework Overview

In previous work [15], we defined a sort-first parallel framework for interactive rendering of large height fields. It consists in a parallel implementation of Roettger's algorithm [25]. We choose this algorithm because of its efficient use of the memory. In addition, the runtime operations locality of this algorithm makes it a good candidate to parallelization. Nevertheless, most of the ideas in the framework are enough general for being applied to other LOD algorithms. Our method is summarized in the Figure 1. At a processing stage, terrain height data are partitioned into tiles and for each tile roughness values (called $d2$ values) are computed and stored. These values are a key-feature for allowing non-uniform simplification as they influence the decision variable f governing the hierarchical structure (a quadtree) generation. Tile size is carefully chosen such that minimal achievable resolution is acceptable while possible load imbalance in LOD computation due to the user interaction is kept minimal. Additional information is added to each tile regarding its neighborhood. This information is used to solve the border simplification problem and to implement an accurate distributed geomorphing mechanism. Tiles are next assigned in a round-robin fashion to a large subset of cluster nodes devoted to LOD computations. This set of nodes is depicted as computation cluster in Figure 1. At runtime, using Roettger algorithm, a quadtree is constructed from each tile at each frame. Simplified data from computation nodes are sent to a limited subset of cluster nodes devoted to rendering called visualization cluster in Figure 1. A compression mechanism that exploits frame-to-frame coherence allows to further reduce the communication overhead.

As we used the hardware texture support available on modern GPU in [15], all texture data have to be loaded on each rendering node video and/or main memory. Unless non trivial out-of-core mechanisms for texture data are implemented, this limits the size of LOD-able data for multi-modal applications. To overcome such a limitation, we distribute these additional data using our parallel framework. This distribution scheme presents a double advantage. First, it can be extended to any per-vertex data. Second, besides the hierarchical view frustum culling and the compression due to the frame-to-frame coherence, it benefits from a novel data encoding to minimize the induced increase in communication overhead. This encoding processes the output of the core LOD algorithm which consists in independent triangle fans. The processing aims at discarding the vertex redundancies present in this kind of data structure.

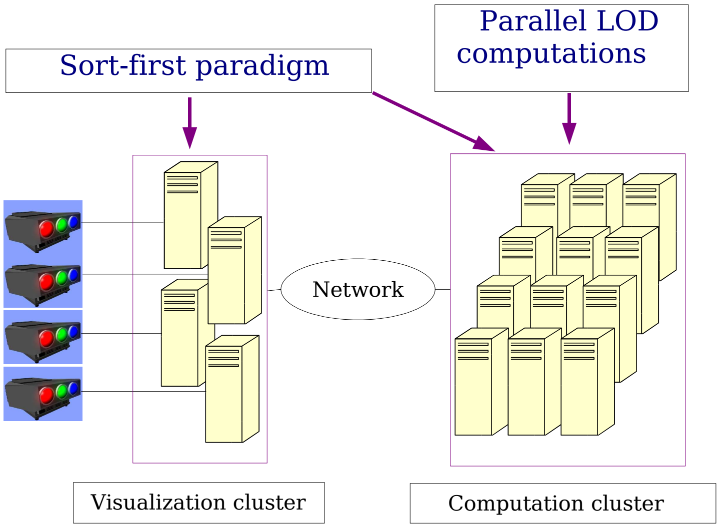

Figure 1: Our parallel framework model.

A trivial solution to handle additional layers of data would be to pack every height value in the tiles with its corresponding extra data. Next, at each frame, not only the height values but data of all types are sent from LOD processing nodes to rendering ones, where each vertex of the simplified model is rendered taking into account the extra parameters(color, alpha value, normal vector). The drawback of this approach is that the LOD algorithm simply picks in the original map data the values associated to the remaining vertices after LOD processing. As the latter is driven by geometrical parameters, the approximation of the locally simplified information is only good for the geometry. It may not be suitable for some parameters such as color data. For typical color information, the obvious choice for computing a correct approximation value would be to average all the neighboring data lost by simplification.

Starting with color data, which represents the most common extra terrain data encountered in rendering applications, we present an approximation technique that works in pair with Roettger algorithm, and gives a more accurate result than simple value picking. The mechanisms will be generalized to any other type of extra data, and will be shown to be compliant with our parallel framework.

3 Color Mipmapping

In order to maintain the scalability of our framework in a multi-modal context, we need to move texture data from the rendering nodes to the larger computation cluster. The counterpart is that we have to perform ourselves a part of the GPU job: colors management. Actually, the basis of our method is closely related to a very common texture rendering technique, *mipmapping*, which is used to generate and manage a color map at various levels of approximation. Mipmapping solves texture LOD problem by generating a pyramid of textures called mipmaps. As illustrated in Figure 2, the pyramid starts from level 0 with highest resolution down to a texture consisting of just one pixel. Each level have half of the size of the prior level. At runtime, the appropriate mipmap level is chosen for each object according to its screen-projected size. Mipmapping is very useful in view-dependent LOD rendering of large terrains. Indeed, when using one-resolution texture in such a context, aliasing effects appear at farther simplified areas. Using mipmapping allows to correct these artifacts. So, integrating this technique in the Roettger LOD algorithm is done in a natural way. As both are quadtree-based, an intuitive way to affect to each quad-node (referred as block for a sake of clarity) vertex a color from a mipmapped extra color layer is to address the same mipmap level as the block level.

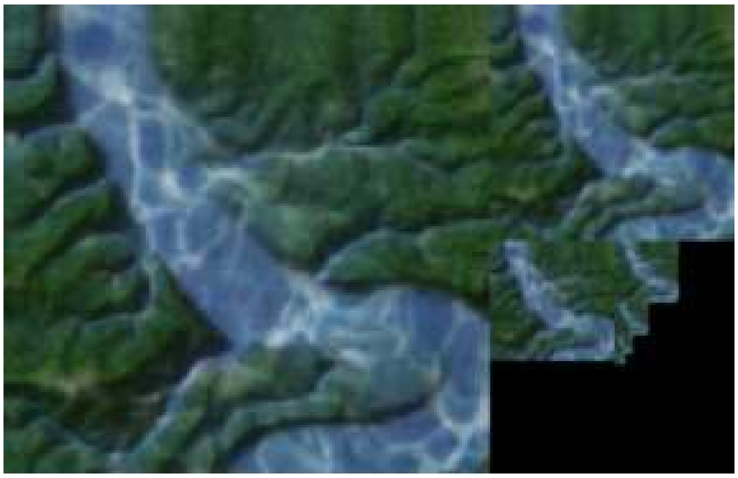

Figure 2: A mipmap hierarchy.

A block is made of a middle point, four corner points and up to four mid-edge points. All of them require to fetch a specific color at the correct mipmap level, but except for the middle point they all also belong to neighboring blocks which may be of different levels of detail due to non-uniform LOD simplification (see Figure 3). In order to ensure consistent color affectation on shared vertices, an effective way is to average the contribution of each block to a shared vertex color. A primary attempt would be to average the level of detail of each block, addressing the mipmap at the resulting level, but smoother transitions are achieved by fetching for the shared point each block respective color contribution, which are next averaged. Thus, whatever the shape of the simplification, the coloring takes into account the whole color map, with a correct per-vertex color approximation.

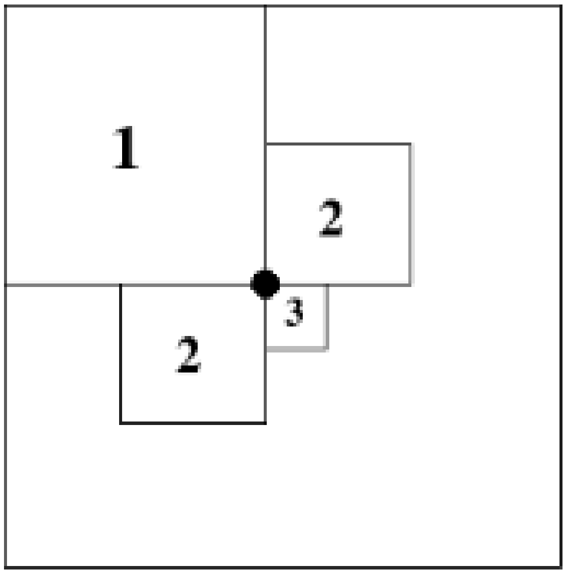

Figure 3: A vertex shared by several blocks of different levels.

In our parallel framework, each height map tile is associated with its equal sized corresponding color layer, of which mipmap hierarchy is computed at preprocessing step. Let's note that the additional memory footprint required for mipmapping is only 33%. As for each mipmap computation the averaging process must take into account up to half part of neighboring tiles, it is easy to design a simple out-of-core preprocessing algorithm for mipmaps. Each tile data along with its corresponding neighborhood are loaded for mipmaps generation and at once removed from the memory. As usual, a border problem may occur, but our previous work about this issue enables us to find block levels across borders, which gives us all necessary coefficients for averaging. As we preprocess mipmaps over the whole terrain, their values are consistent from one tile to another, so finally the resulting averaged colors match perfectly for every distributed tiles.

4 Color Morphing

As for geometry data, popping may occur for color data during the navigation; so, we need a kind of color morphing, or more generically a data morphing concept that acts like geomorphing. Thanks to the mipmapping and the fact that Roettger algorithm insures that the level of detail difference between two neighboring blocks is never more than one, colors changes are seldom abrupt. But the artifacts remains clearly noticeable. Actually, morphing occurs at two levels: at newly created vertices by refinement, *i.e* the block middle vertex and mid-edge vertices, and at vertices shared by evolving blocks. The latter case is new compared to Roettger geomorphing which only applies to block height values.

In [25], a blending factor b in [0,1] derived from the decision variable f is used to perform a smooth transition from an approximated to a real position. For our color morphing, the idea is to use this blending operator to smoothly pass from one mipmap level to another. In the graphics domain, this process is related to *trilinear filtering*, which adds to the traditional texture bilinear filter an "in-depth" interpolation between two consecutive mipmap levels. For our purpose, we have seen in previous section that any vertex in the simplified model belongs to up to four blocks of different levels. Let b the blending factor and l the level of the leaf block, we have the following relation:

$$C_f = (1-b) * C_{l-1} + b * C_l \tag{1}$$

where C_f is the filtered mipmap color and C_i the mipmap color at level i. So, the final vertex color is the average of the color contribution of each block it is connected to.

Concerning the color morphing at newly created vertices, the scheme initially developed for geomorphing is directly applicable: the blending factor b is used to move from the interpolated color previously visible at the location the new vertex is created, towards the actual color defined for this vertex. Note that this target color evolves slightly due to the mipmap filtering, but as the morphing depends only on the viewpoint, it is constantly re-evaluated at each frame and then stay consistent.

5 A Generic Mipmapping Framework

In previous sections we have focused on color layers, encoded as per-vertex vectors of 3 scalar components (intensity of red, green and blue). In order to support a richer set of data types, we define in this section a more general mipmapping scheme for our framework which can virtually handle any kind of per-vertex information, and with the adequate semantics. To be able to process a particular new data type, the following operations must be defined in adequacy with the nature of the data and their characteristics we want to visualize.

5.1 An approximation function

It is the key point that should guarantee a coherent representation of global information from any viewpoint during navigation. This function is used to successfully compute each mipmap level, and generally consists in a four-points approximation into a single value (since a mipmap at level l has one-fourth the surface of the one at level $l-1$). For a photographic layer, we calculate the average of each 3 color channels. For a normal map, a more specific averaging function must be chosen in order to keep unit-length the approximated vectors. But the actual strength of the approximation function is to be able to take into consideration not only the nature of the data, but also its semantics. For example, there are multiple ways to approximate colors. As stated before, the common way is averaging, but for some kinds of abstract color maps, it would be better to use a more meaningful statistical function such as a max, a min or a median. A typical example is a visual application dedicated at finding gold deposits, starting with an overall view of the terrain. With an approximation function working by an average over a gold density gray-scale map, at large scale the few gold mines would be totally unnoticeable because totally overwhelmed by the *no gold* information. On the other hand, with a max function, every local approximation will indicate the greatest gold density in the corresponding zone, driving the user towards the information he is searching for.

5.2 A rendering technique

With the support of additional information over geometry, vertex parameters are not limited any more to the 3D coordinates but also could be associated with colors, alpha value, normal vector. To match the flexibility gained on the computation side, the rendering part highly benefits from latest GPU programmability: *shaders*. Thanks to vertex and pixel shaders, the programmer can render most kinds of data in the way it appears the most meaningful on the screen. Furthermore, shaders allow a tight integration of multiple data types in a single and optimized rendering pass. And finally they make available new rendering effects such as Phong shading for accurate lighting or custom filtering. For example, we use a four-points bilinear filter to improve vertex colors Gouraud shading over quads. This shader may be used to bypass a common problem with quads rendering on graphics cards that work with triangle primitives only: a quad is necessarily split up into 2 triangles, meaning the smooth shading is operated over 3 points only rather than four. This leads to non uniform and non deterministic quad drawing, and often let appear a disturbing visible diagonal (see Figure 4, left). Our four-points bilinear filter sends for each of the two quad triangles the four colors at each corner, and a coordinate to fetch from a 2D precomputed texture the four blending coefficients (stored in the R,G,B,A channels) to apply respectively to the corner colors (see Figure 4, right).

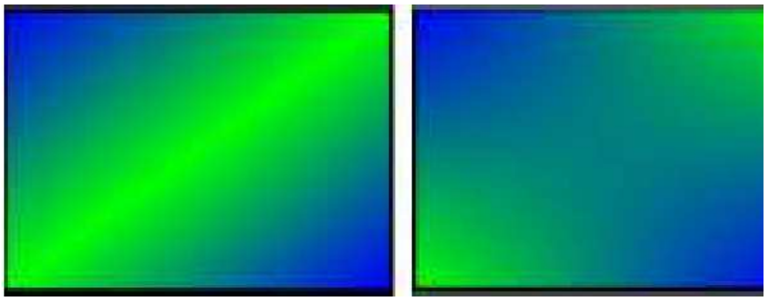

Figure 4:
Left: a quad rendered with traditional Gouraud shading
Right: the same quad with our four point bilinear filter

5.3 A morphing function

As we seen in section 4, the morphing process operates in two areas: at every existing vertex when moving from one mipmap layer to another, and in conjunction with geomorphing where new vertices are appearing due to LOD refinement. Both have a common operation: the morphing function that takes several data values and blends them together according to contextual coefficients. This operation must preserve the characteristics of data types (for instance unit-length normal vectors), and produce transitions as smooth as possible. Specifically for newly created vertices, we must also be able to determine a starting value for the morphing process. This problem is closely related to the rendering part: the starting value must be computed in such a way that when rendered, it matches perfectly the appearance of the previously interpolated value at its location. This is required to guarantee the absence of popping. For this, the use of shaders on the rendering side helps to know exactly how pixel colors are computed, so the same computation must be repeated on the compute side.

5.4 An application: missing values

Large geoscientific data often suffer from local lacks of actual data, leading to undefined areas of any shape that are difficult to render in a LOD context. Indeed, the latter must preserve the terrain topology in spite of the simplification. In order to manage the holes due to the missing values, we apply our generic mipmapping method. We define the three previously stated functions as follows. Vertices alpha component in the range [0,255] is used to generate mipmapped alpha layers. At level 0, the mipmap is filled with 255 when the corresponding height value is valid, and 0 otherwise. For mipmap sub-layers creation, a standard averaging operation is used as approximation function. The alpha values can become semi-transparent proportionally to the amount of missing values in the approximated area. So no special morphing function is needed for transitions between mipmap levels. Finally, the rendering function must draw the transparent triangles back-to-front in a second pass.

5.5 Other possible uses

The generic mipmapping method can be used for other data types such as lighting parameters through normal maps or thematic information given as texture samples. Normal maps allow dynamic lighting by associating a normal vector to each vertex. A mipmapped scheme can be used by choosing an angular interpolation as approximation function rather than independent coordinates interpolation. A per-pixel Phong-like shading is required to control morphing operations which also have to take care of keeping normal vectors normalized. Similarly, texture samples may be used in soil description. Soil data layers can be built as a vector of n coefficients (a single byte), each of them corresponding to the proportion of a particular soil type at each vertex location. On the rendering side, a shader should be used to apply on each pixel n repeated seamless textures modulated by these coefficients. The texture samples technique also benefits from hardware texture mipmapping that acts in pair with our technique at distant view.

6 Exchanged Data Encoding

Despite the fact that both height values and extra data layers share many common points in the way we handle them in our extended LOD algorithm, the respective output data representing the simplified model mainly differs in the following aspect. From one frame to another, large parts of the simplified height map geometry corresponding to non-morphing quadtree leaves remain unchanged, whereas other data processed by our filtered mipmapping scheme generates new values for every vertex in the model. As a consequence, in our distributed framework, no opportunities appear for extra data to benefit from frame-to-frame coherence in order to lighten data communication. Henceforth, the amount of data to transfer, inflated by processing other data layers in addition to height values, becomes a crucial problem.

In our previous work, geometry information was sent to visualization nodes in the form of a set of fans, *i.e* a series of ordered vertices data. A major issue with fans encoding is a strong data redundancy since a vertex can be shared by up to four fans. A simple indexed vertex buffer method would solve the redundancy problem but at the cost of some additional overhead. In order to economize network bandwidth, we have developed an efficient data encoding that gets rid of any redundancy and drastically reduces vertices positioning or indexation thanks to a highly compact header structure that relies on Roettger LOD specificities. The key point in our solution is near-implicit vertex positioning due to a careful ordering of data.

We first used this idea for optimally storing $d2$ values. Instead of a large matrix with unused cells, we build a *sequential quadtree*: for a tile quadtree of fixed depth n, node values are stored in a buffer of size $\sum_{i=0}^{n} 4^i$ in the order corresponding to a breadth-first traversal of the tree. The index of a value in the buffer has a unique equivalent quad (size and position) over the tile. When stored in a two-dimensional matrix, quad attached values were directly accessed by their row and column indices ; whereas in a sequential quadtree, indexation parameters are the quad level l and an offset which identifies the quad among the 4^l possible ones at this level. In order to index quickly the first value at level l in the sequential quadtree buffer, we use the simplified formula $\sum_{i=0}^{l-1} 4^i = \frac{2^{2*l}-1}{3}$ implemented with bits shifting for a fast calculation. This structure is convenient for the manipulation of perfect trees as for $d2$ values.

Our goal being communication data lightening, the ideal case would be to send height, color and other per vertex/fan information in such an order that they could be identified at the reception and the quadtree easily reconstructed. The problem is that such a simplified quadtree is of any kind, *i.e* not necessarily perfect. So sequentially storing its nodes would lose its structure. Our solution consists in sending a buffer per tile, with all block data sequentially encountered in a breadth-first traversal of the simplified quadtree, taking care to skip any redundant data. We append to this buffer a very light description of the quadtree structure called *Simplified Sequential Quadtree* (SSQ), which allows to decode the data buffer and build the simplified model. Actually, breadth-first traversal is not required: the key is simply that the receiver must process the buffer in the exact order as the sender. The SSQ is a highly compact structure encoding the state of each simplified quadtree node in a single byte, or more specifically 8 independent bits which are associated to the following meanings:

- **1st to 4th bits**: they indicate whether the quadtree node own a top-left, top-right, bottom-left or bottom right children. Indeed these bits are used to encode the refinement state. Thanks to this information, it is possible to identify which nodes have been recorded into the data buffer paired with the *SSQ*, and thus to read the correct count of data samples in the appropriate order.
- **5th to 8th bits**: these bits record an information more specific to the Roettger LOD. When set, they indicate the presence of a vertex in the middle of a quad left/top/right/bottom edge, which is dependent of the quad neighborhood. This information is very important. First it allows to rebuild the triangle fans matching exactly the Roettger LOD simplification and second, it indicates the number of per-vertex data samples to read for the block in the data buffer.

Special cases must be taken into account. For example, a quadtree partially in the viewing frustum will have some part culled away, making it incomplete. Since a quad cannot own mid-edge vertices if not refined, we have identified that any bits combination beginning with *0000* then at least one bit set in the remaining part should never occur; so values encoded in that way can be used as special identifiers. We set such value (for example *00001111*) to mark a quadtree node pruning in the *SSQ*. Finally, another special case is the four corner vertices of the root quad: in the general case, a quad acquires its corner vertices from its parent, except, naturally, the root quad. The data for these four vertices must be written to the buffer at first.

Our data encoding scheme is illustrated in Figure 5. The strength of our data structure is clearly its lightness: a set of up to 5 vertices data is described in only one byte. For instance in the case represented in Figure 5, with 5 bytes per data sample, the SSQ is built with 3 bytes where the raw data buffer, with no redundancy, is made of 55.

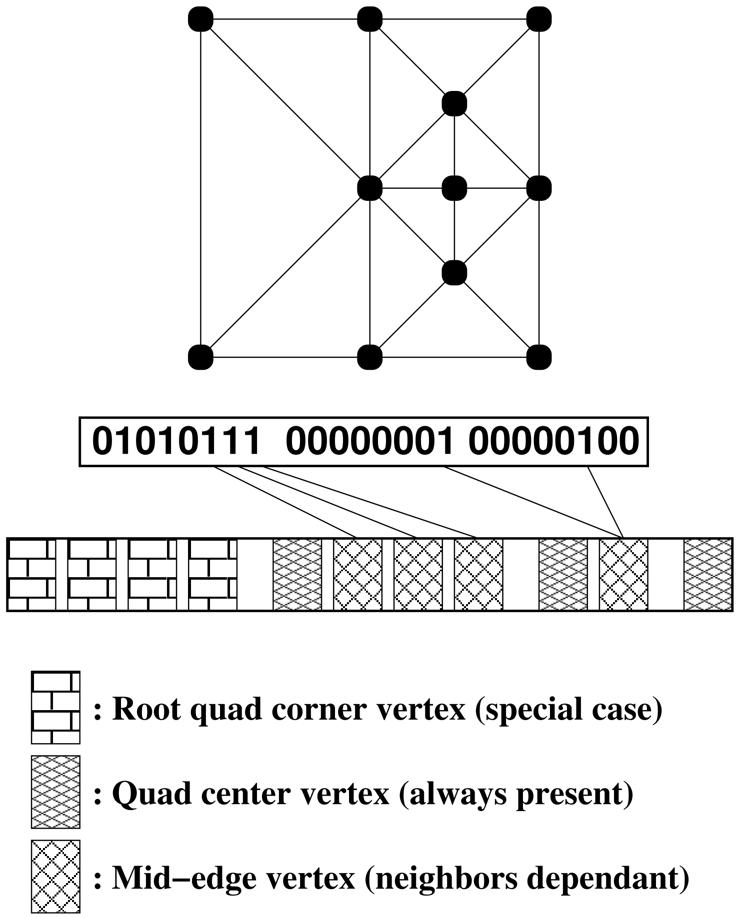

Figure 5: A simple quadtree data encoded with a SSQ. At the bottom is represented the buffer containing the per-vertex data samples related to the SSQ

7 Performance Analysis

7.1 Data and hardware setup

We performed our tests on a cluster of 8 PCs each equipped with two Pentium 4 processors, 1 GB of RAM and 1 NVIDIA GeForce FX 5900 graphics card with 256 MB of video memory. The nodes are connected by a Gigabit Ethernet network. The same screen resolution of 1024x768 is used in all configurations. We use a publicly available dataset for all tests (see ftp://edcsgs9.cr.usgs.gov/pub/data/srtm). The flythrough on the terrain is fixed in order to keep the interaction identical in all cases. The elevation data size amounts to 3.5 GB. Texture data, lighting parameters and missing values are dealt together with elevation data. As our framework is a tile-based system, we can load a variable number of tiles in order to change our dataset size, a tile being of size 257x257.

7.2 Software environment

Concerning the software environment, we used FlowVR [1]. FlowVR is a middleware aiming at distributing Virtual Reality (VR) applications on cluster and grid architectures. From FlowVR point of view, an application is a set of possibly distributed modules exchanging data. A module is a entity equipped with input and output ports. It receives data on its input, performs some computation on them and then writes the results to the output. A consequence of module definition is that any software can run inside a module, allowing unlimited codes coupling possibility. FlowVR carries out inter-module communication thanks to configuration files and special components called connexions and filters. Connexions materialize communication channels whereas filters are used to perform different kinds of operations on data. The modular approach of FlowVR allows to easily test different deployment scenarii with various communication schemes.

In Figure 6, we give a simplified graph of our application. In this figure, modules correspond to boxes, filters to diamonds and connexions to lines. The points in the upper side of the figure are broadcast filters. For our purpose, only two kinds of modules are needed: computation module and rendering module, each exclusively running on a processor. Every rendering module (*visu* in the figure) has a mandatory output port, *frustum*, for sending its viewing frustum to all computation modules. The viewpoint which is unique is broadcasted by one of the rendering module. Simplified data from computation modules are first gathered on each rendering node using the filters *mergeData* before transferring them to the input port *data* of the local rendering module. Analogously, the filters *mergeLOD* gathers quadtree structures *SSQ* on the input port *lod*. Input and output ports on computation modules, called *heightmap* in the figure, are defined as follows. Their outputs *lod* and *data* are used to send their SSQ and their buffers, respectively, to the corresponding ports on the rendering modules through filters. Similarly, their input ports *frusta* and *viewpoint* receive the view parameters written on rendering modules output ports.

7.3 Benchmark results

In order to check the algorithm data scalability we progressively increase simultaneously the terrain data size and the number of computation nodes by the same factor. In this experiment, all setups use 1 rendering module and a different number of LOD modules: 1, 2, 4 and 6 respectively. We note $n-m$ a setup with n rendering modules and m computation modules. Each computation module is loaded with 1000 tiles which

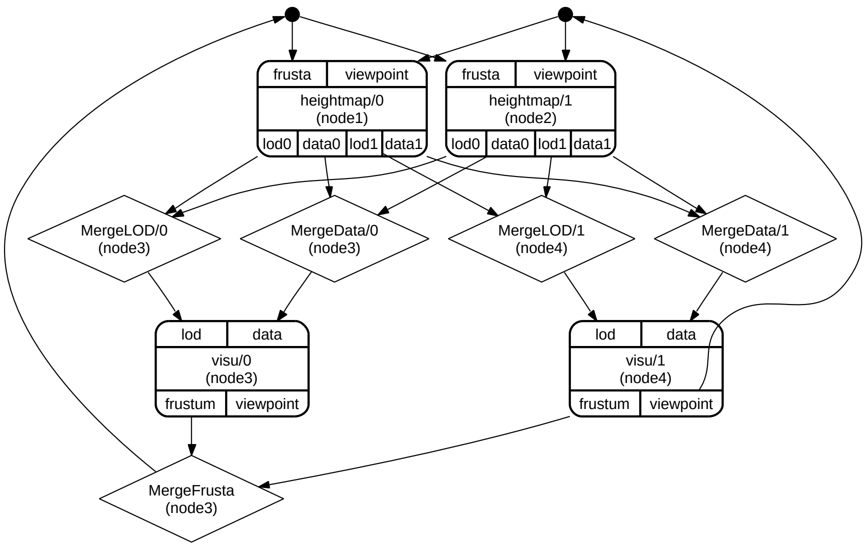

Figure 6: FlowVR diagram.

amounts to about 800 MB. This entirely saturates our 1 GB RAM machine without, however, requiring swapping. Data scalability is measured as frame rate speed-up per 1000 tiles. Before explaining the scalability results in Figure 8, we recall that the frame duration is determined by three major stages: LOD computation, communication and rendering. By distributing initial data , we decrease LOD computation duration (see full-line curve in Figure 7) but a communication overhead is created. Thanks to the LOD control, the exchanged data size augments only slowly whereas terrain data linearly increases (see doted-line curve in Figure 7). Thus, only a little increase in rendering duration is observed. At overall scale, the computation time reduction compensates the communication and rendering overhead. This allows the speed-up curve shown in figure 8.

Figure 8 shows a speed-up curve where 1-1 setup is the reference.It illustrates two different phenomena. First, we show the scalability of our framework. As the data size grows proportionally with the number of computation nodes, the frame rate per node load **L** (1000 tiles) grows in the same proportion. As a consequence, in our framework, when you need to enlarge the data size, you just need to add more cluster nodes and the real frame rate does not fall. The second phenomenon is illustrated by the super-linear speed-up showed in figure 8. For 1-2 setup (two computation nodes) we obtain a three time greater frame rate per **L** than for 1-1 setup (one computation node). This is due to an intrinsic characteristic of view-dependent algorithms: the LOD workload is concentrated in the areas close to the viewpoint, the farther ones being drastically simplified. Hence, in the 1-2 setup, we enlarge the data size but we do not enlarge the total LOD workload in the same proportion. Thanks to the good load balancing of our framework, two processors compute the LOD located near the viewpoint, and this gain is not compensated by the data size increase. Then, a part of the workload benefits from

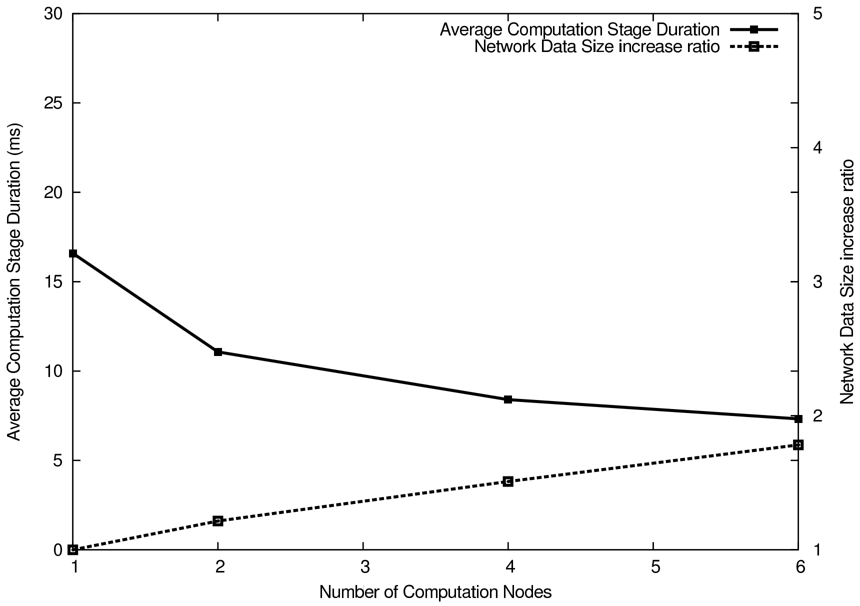

Figure 7: Average computation stage duration (full-line curve) and overall exchanged data size increase ratio (doted-line curve). The overall number of tiles is: the number of computation nodes X 1000 tiles.

node number growing, and so we obtain super-linear speed-up per **L**. In conclusion we show that we benefit twice of our parallel framework, we can scale data size without frame rate drop. Moreover, we obtain better frame rates.
Next, we use the codes coupling facility of FlowVR to add multi-display ability to our framework by coupling the Net Juggler [2] software to FlowVR. Net Juggler is an efficient parallel rendering tool based on a multiple copies approach. Using the couple FlowVR-Net Juggler, we parallelize not only LOD computations but also the rendering work. According to the latency of the network, using a single rendering node may be penalizing. In such a case, we benefit from the multi-display ability: the graphical workload can be distributed on all the available rendering nodes, ideally dividing the rendering duration by the size of the visualization cluster. Tests evaluating parallel rendering contribution are as follows: all setups comprise 4 computation nodes and the first has 1 rendering node, the second 2 and the third 4. A flythrough that almost ensures load balancing is used for the entire test. The results are summarized in Figure 9. We clearly observe a linear decrease of the rendering durations. In average, values for the 1-1 setup almost quadruple those of the 4-4 one. Finally, screen shots of our application running in a 2-2 setup is shown in figure 10.

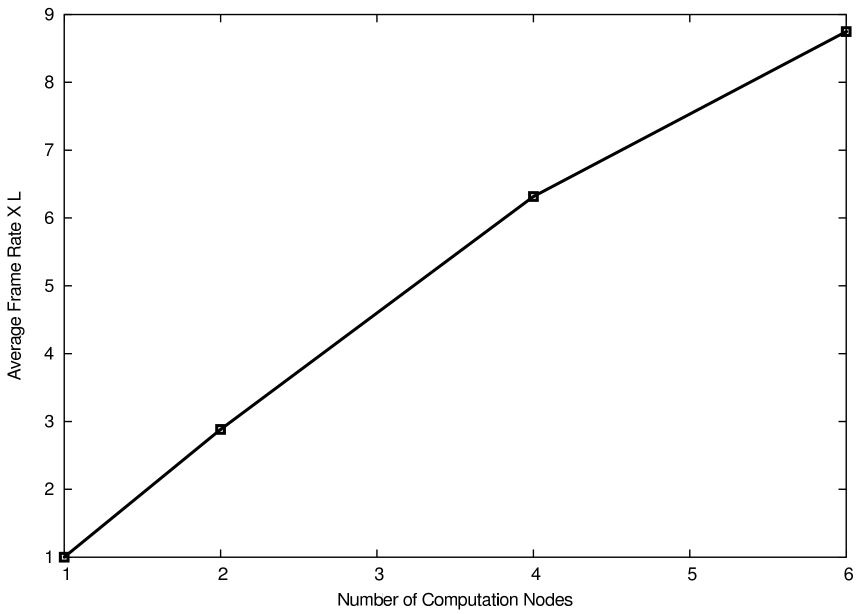

Figure 8: Average frame rate speed-up per node load **L**=1000 tiles for different number of computation nodes.

8 Conclusion and Future Work

We presented a generalized cluster-based framework for LOD rendering of large terrains. A generic mipmapping method combined to the core geometrical LOD made it powerful enough to allow LOD control over any per-vertex data. Not only the framework is proved scalable but it also improved the rendering quality. Indeed, the two limiting factors, which are the network bandwidth and the memory, are addressed. The memory issue is solved by distributing all data on an extensible cluster. Regarding the network bandwidth problem, besides the existing culling and compression mechanisms, a compact data structure is defined to lighten the data exchange overhead. Finally, the rendering is improved by defining morphing techniques for all attributes in addition to the mipmapping. Efficient cluster architecture software are introduced to ease application deployment. The strength of our approach lies in the complementarity of the provided solutions, each tackling a different part of the large problem of massive terrains visualization.

Despite the integration of extra data in our framework, the parameters directing the simplification process are still based on the geometry. However, there are situations where attribute or thematic information may be much more interesting than geometry. In such a case, it would matter to better weight the extra feature than the geometry so

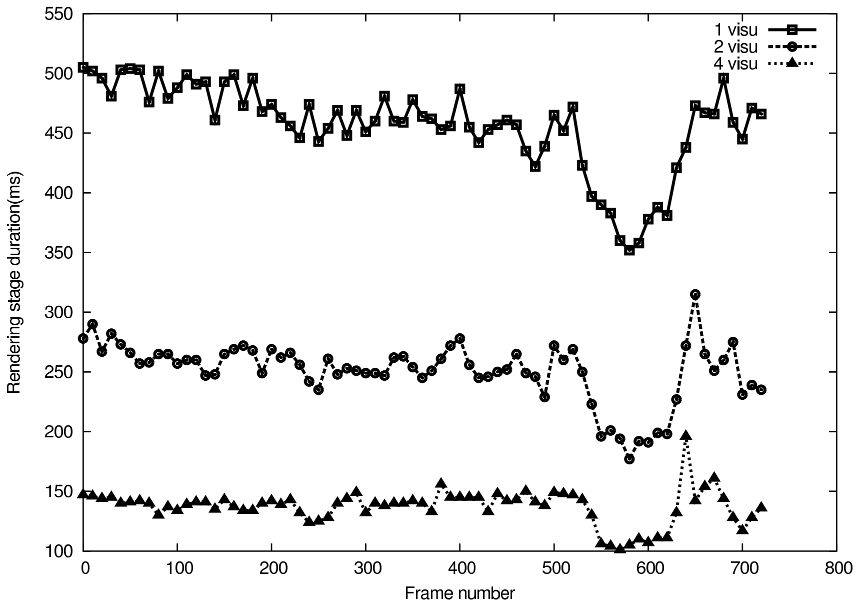

Figure 9: Rendering time for hardware setups with different numbers of rendering nodes.

that it remains preserved when decimating. This requires to incorporate extra data information into the core LOD algorithm. Moreover, an essential feature for an accurate perception of the geometry is lighting. As it depends on the shape of the terrain, it must follow the variation of the latter induced by the LOD. The way to do this is to use normal maps. They are proved mipmap-friendly [8]. So they are very appropriate for our framework. Although the scalability is only limited by the number of computation nodes in the cluster, it may be useful to implement triangle meshes compression techniques. This allow to further reduce network bandwidth use. In the same way, an occlusion culling support may speed hilly terrains rendering up. Moreover, we are far from exhausting FlowVR capacities. The flexibility given by this software may be used to implement adaptive load balancing strategies. Finally, regular grid based LOD are known for their runtime efficiency. It would be favorable to apply this to other types of data: triangulated irregular networks (TIN) data, time-varying data from scientific simulation.

Acknowledgments

We would like to acknowledge the BRGM (French Geological survey) and Rgion Centre of France for supporting part of this work. We also would like to acknowledge

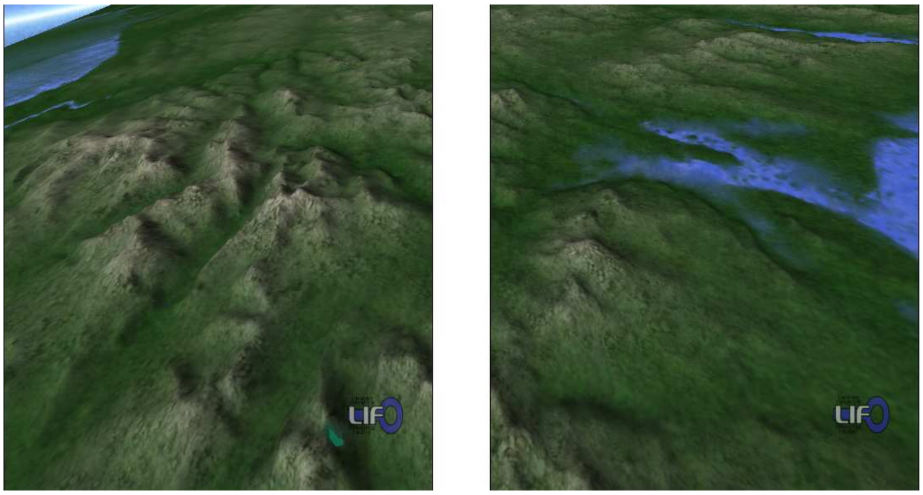

Figure 10: Screen shots of our application running in 2-display setup.

Sylvain Jubertie for setting up the environment for our tests.

References

[1] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. Flowvr: a middleware for large scale virtual reality applications. In *EuroPar 2004*, August 2004.

[2] J. Allard, V. Gouranton, L Lecointre, E. Melin, and B. Raffin. Net juggler: Running vr juggler with multiple displays on a commodity component cluster. *IEEE VR'02*, 2002.

[3] K. Baumann, J. Dollner, and K. Hinrichs. Integrated multiresolution geometry and texture models for terrain visualization. *Proceedings joint EuroGraphics-IEEE TCVG 2000*, May 2000.

[4] D. Brodsky and J. B. Pedersen. Parallel model simplification of very large polygonal meshes. *PDPTA'02*, pages 1207–1215, 2002.

[5] P. Cignoni, F. Ganovelli, E. Gobetti, F. Marton, F. Ponchio, and R. Scopigno. Bdam: Batched dynamic adaptive meshes for high performance terrain visualization. *Proceedings EG2003*, pages 505–514, September 2003.

[6] P. Cignoni, C. Rocchini, C. Montani, and R. Scopigno. A general method for preserving attribute values on simplified meshes. 1998.

[7] P. Cignoni, C. Rocchini, C. Montani, and R. Scopigno. External memory management and simplification of huge meshes. *IEEE Transaction on Visualization and Computer Graphics*, 2002.

[8] J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. *In SIGGRAPH'98 Computer Graphics Proceedings*, pages 115–122, 1998.

[9] L. DeFloriani, P. Marzano, and E. Puppo. Multiresolution models for topographic surface description. *The visual computer*, 12(7):317–345, 1996.

[10] F. Dehne, C. Langis, and G. Roth. Mesh simplification in parallel. *ICA3PP'00*, pages 281–290, 2000.

[11] J. El-Sana and Y.-J. Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 3(19):139–150, August 2000.

[12] J. El-Sana and A. Varshney. Parallel processing for view-dependent polygonal virtual environments. *Proceedings SIGGRAPH'99*, 1999.

[13] M. Garland. Multiresolution modeling: survey & future opportunities. *In Proceedings eurographics'99, State of the Art Reports*, pages 111–131, 1999.

[14] V. Gouranton, S. Limet, S. Madougou, and E. Melin. a scalable cluster-based parallel simplification framework for height fileds. *EuroGraphics/ACM SIGGRAPH, Proceedings Parallel Graphics and Visualization'04*, pages 59–65, June 2004.

[15] V. Gouranton, S. Madougou, E. Melin, and C. Nortet. Interactive rendering of massive terrains using pc cluster. *EuroVis 2005: Eurographics/IEEE-VGTC Symposium on Visualization, To appear*, June 2005.

[16] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification alogrithms. *Multiresolution surface modeling course notes, ACM SIGGRAPH'97*, 1997.

[17] H. Hoppe. Progressive meshes. *In proceedings SIGGRAPH'96*, pages 99–108, 1996.

[18] H. Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. *IEEE Visualization '98*, 31:35–42, October 1998.

[19] L. Hwa, M. Duchaineau, and K. Joy. Adaptive 4-8 texture hierarchies. *IEEE Visualization '04*, pages 219–226, October 2004.

[20] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital michelangelo project: 3d scanning of large statues. *Proceedings SIGGRAPH 2000*, pages 131–144, July 2000.

[21] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. Turner. Realtime, continuous level of detail rendering of height fields. *Computer Graphics, Proceedings SIGGRAPH'96*, pages 109–118, 1996.

[22] A. Mirin, R. Cohen, B. Curtis, W. Dannevik, A. Dimitis, M. Duchaineau, D. Eliason, D. Schikore, S. Anderson, D. Porter, P. Woodward, L. Shieh, and S. White. Very high resolution simulation of compressible turbulence on ibm-sp system. *Proceedings of Supercomputing 99*, November 1999.

[23] J Montrym, D. Baum, D. Digman, and C. Migdal. Infinitereality: A real-time graphics system. *Proceedings of ACM SIGGRAPH 97*, pages 293–302, 1997.

[24] E. Puppo and R. Scopigno. Simplification, lod and multiresolutio: principles and applications. *In EuroGrahics'97, Tutorial Notes*, 1997.

[25] S. Rottger, W. Heidrich, P. Slussallek, and H-P. Seidel. Real-time generation of continuous levels of detail for height fields. *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, pages 315–322, February 1998.

[26] T. Ulrich. Rendering massive terrains using chunked level of detail control. *SIGGRAPH Course Notes*, 2002.

[27] http://rocky.ess.washington.edu/data/raster/tenmeter/.

[28] L. Williams. Pyramidal parametrics. *In Proceedings SIGGRAPH'83 computer graphics*, 17(3):1–11, July 1983.

[29] S. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-vdr: Interactive view-dependent rendering of massive models. *IEEE Visualization 2004*, pages 131–138, October 2004.