Hardware-Independent Clipmapping

Antonio Seoane
antonio.seoane@videalab.udc.es

Javier Taibo
jtaibo@udc.es

Luis Hernández
lhernandez@udc.es

Rubén López
ryu@videalab.udc.es

Alberto Jaspe
jaspe@udc.es

VideaLAB
School of Civil Engineering - University of A Coruña, Spain
Campus de Elviña 15071 - Spain

ABSTRACT

We present a technique for efficient management of large textures and its real-time application to geometric models. The proposed technique is inspired by the *clipmap* [12] idea, that caches in video memory a subset of the texture mipmap pyramid. Based on this concept, we define some structures and a different management allowing its implementation on a personal computer without specific graphics hardware. Finally, we present the results of the application in a terrain visualization system, using several simultaneous textures with a detail up to 0.25 meters per texel, covering a 60,000 km^2 area.

Keywords
Terrain visualization, multiresolution visualization, large data set visualization, level-of-detail techniques, texture mapping, clipmap, mipmap, texture caching.

1 INTRODUCTION

Applying textures to digital terrain models is the classical solution to simulate the missing geometry details. When we want to apply a large amount of texture to the terrain surface, the storage problem arises. Although both the system memory and the video memory are extremely fast, there is limited storage capacity. Therefore the use of a paging technique that allows an efficient management of the texture data in order to visualize the terrain with a very high quality is essential.

In contrast to the abundance of terrain geometry management algorithms, there is little work focused on handling the texture. Moreover, most of the systems allowing the visualization of digital terrain models establish a strong dependency between both geometry and texture data bases.

Usually, texture tiles are bound to the geometry with a pre-established mapping. This makes it difficult to modify or to replace the geometry or texture data in a transparent way, without rebuilding the database.

One of the best and most used approaches that allows the handling of big textures is the *clipmap* [12]. This technique separates the handling of the texture and the geometry, allowing independence between both databases. The main problem of this technique is the requirement of specific hardware.

We propose a new technique that allows the handling of a large amount of texture without any requirement of specific hardware. Only OpenGL or Direct3D fixed function pipeline is required to implement this technique. It uses a two level cache composed of a texture stack stored in the video memory as the first level and a set of buffers stored in RAM as the second level. The contents of both levels are updated depending on camera movement.

This technique is inspired by the clipmap idea. It is also based on the caching in video memory of a huge mipmap pyramid [13]. Nevertheless, its structure, the management of the video memory and the way the texture is applied are different, which allows its implementation in a personal computer using a graphics API like OpenGL [11].

Our texturing technique provides the following advantages:

- It can be implemented using an API like OpenGL without special necessities in the graphics hardware.
- It keeps the independence between geometry and texture databases.

- Texture coordinates can be computed in the GPU (although it is not necessary), avoiding their transference to the graphics system. This allows modification of the geometry in real time, while keeping the right texture mapping without recomputing the texture coordinates.
- Texture aliasing is avoided using trilinear filtering hardware capabilities.
- It allows the visualization of high resolution textures with the possibility of including higher resolution insets.
- It allows the use of several independent large textures which can be combined to show different information types simultaneously on the terrain.

2 PREVIOUS WORK

Historically, the strategies for the terrain texturing problem have been based on paging systems. With these kind of techniques it's usual to solve the high texture volume problem applying high resolution information to the terrain regions closer to the camera, and less detail to those which are further away. Thus we only need to store in video memory the higher resolution data for a small portion of the terrain. Moreover, due to the perspective, the further we are from a terrain area, the fewer pixels used on the screen, and therefore, the fewer texels needed for accurate representation.

Rabinovich [9] proposed in his visualization system the use of a single texture covering the whole terrain, with hardware mipmap. This technique has scalability drawbacks, as the maximum texture size allowed by graphics hardware is very limited, and greatly exceeded by the resolution needed to represent large terrain surfaces with an acceptable visual quality.

A solution to the previous limitation is the clipmap [12], which caches a subset of a mipmap pyramid. Terrain is mapped at every point with the finest available level of detail. As the camera moves, the system updates the pyramid cache with the information corresponding to the new area. The clipmap, nevertheless, needs special hardware for its implementation.

Another solution is MPGrid [6], that uses several pyramids instead of only one. Each pyramid is a mipmap that should fit completely in memory. If the geometry is not aligned with the texture pyramids, it is necessary to either clip the triangles in real-time or to use a special hardware.

Döllner [5] proposes to store a tree containing a set of texture patches that belong to one texture pyramid. Each texture patch is bound to a geometry patch of another multiresolution model, which must cover it completely. This introduces a dependency that forces the adaptation of texture quality to the loaded geometry level and vice versa.

Blow [2] developed a similar system, in which the tree is a quadtree. For every triangle, the more adequate texture is sought in the quadtree. To achieve efficiency in this algorithm in spite of doing the above mentioned computation once for every triangle, he introduces certain restrictions in the way of doing the clipping and the shape of the triangles. These restrictions create, also in this case, a strong dependency between the geometry and texture systems.

3 STRUCTURE OVERVIEW

The system proposed in this paper is based in the clipmap concept described by Tanner, caching a subset of the texture pyramid in video memory (Figure 1). We introduce a different memory structure of the algorithm, described in this section.

Texture information is structured in three storage levels: disk, system memory (RAM) and video memory (VRAM).

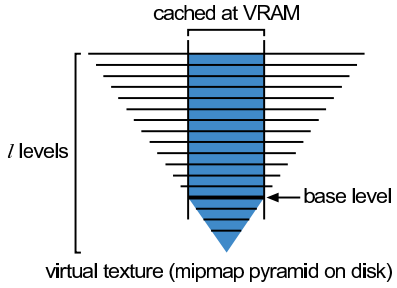

Figure 1: Virtual texture.

3.1 Disk

Texture is completely stored on disk using a mipmap pyramidal scheme. This texture is called *virtual texture*. The highest detail level of this pyramid is formed by $2^{l-1} \times 2^{l-1}$ texels, where l is the number of levels (Figure 1). Levels are numbered from 0 and up, being $2^i \times 2^i$ the size for level i. Higher levels can be incomplete, allowing incrementation of detail for special interest areas (*insets*).

Every level is stored on disk being divided in square fragments of $t \times t$ texels, called *tiles*, except those levels which dimension is smaller than the tile. Tile size is chosen so as to maximize the speed of disk to RAM transfer, while avoiding fragmentation. Tiles are addressed using three coordinates: x (column), y (row) and z (level).

The disk database is fully compatible with the OpenGL Performer [10] clipmap format.

The amount of disk space used by the texture is then estimated with the equation:

$$D \simeq 2^{2l-2} \cdot \frac{4}{3} \cdot b \text{ bytes}$$

where b is the texture color depth measured in bytes per texel.

3.2 System memory

Disk stored tiles are cached in a set of RAM buffers, one buffer per tile.

Cache requests address the tiles by column, row and level. Tile loading is done asychronously, and an LRU algorithm is used to choose the buffer where the tile is to be stored.

Requests are prioritized on a level basis, the coarser levels being those with higher priority. This facilitates having information of the zone of interest as quickly as possible, since lower level tiles cover larger areas. The detail is then being progressively and orderly refined as higher level tiles become available.

RAM usage is determined by the number of buffers (n) assigned to the cache, being calculated as follows:

$$R = n \cdot t^2 \cdot b \text{ bytes}$$

3.3 Video memory

A subset of the virtual texture pyramid stored in disk is stored in video memory (Figure 1).

Clip size (c) determines the maximum size stored in VRAM for each level of the virtual texture. We will choose a *base level*, this being the level of the pyramid with a size equal to the clip size. This base level and lower ones are stored entirely and permanently in VRAM. For each level higher than the base level, a region of $c \times c$ texels around the center of detail is cached. The base level (l_b) is calculated as follows:

$$l_b = \log_2 c$$

Information placed in VRAM is organized in a graphics system texture stack, being those textures independent of each other. The stack is composed by $l - l_b$ textures of $c \times c$ texels each. Texture t_i caches level $l_b + i$ in the pyramid (Figure 2).

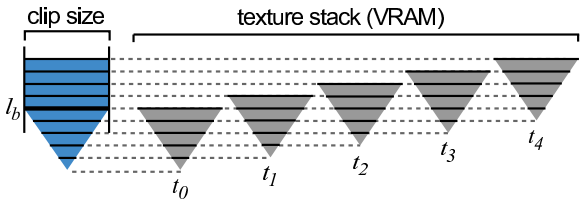

Figure 2: Texture stack.

In order to allow the graphic system to perform a trilinear filtering to avoid aliasing, mipmap levels for every texture are needed. Let t_{ij} be the mipmap level j of the texture i in the stack (Figure 3).

Texture t_0 has all its mipmap levels, corresponding with the base level and coarser levels. For the rest of the textures, level t_{ij} caches level $l_b + i - j$ of the virtual texture. In these textures, the number of mipmap levels can be limited to save bandwidth.

We can visualize this stack as a set of rings representing different resolutions (Figure 4). In the Figure 5 we can see the application of different levels of detail to the terrain, represented by a color code.

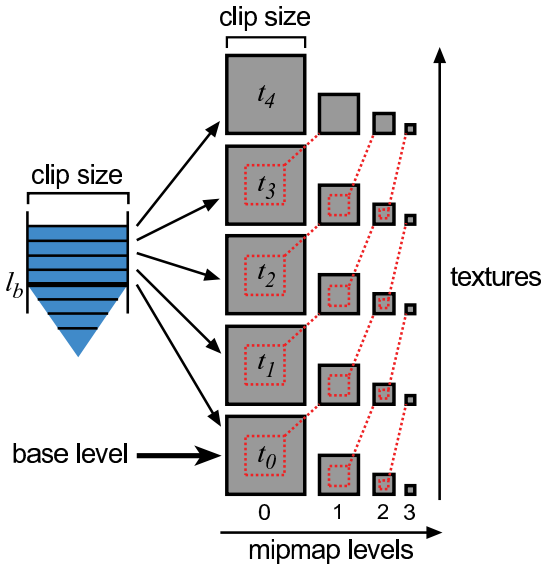

Figure 3: Texture stack. Mipmap levels correspondence.

Figure 4: Rings of detail.

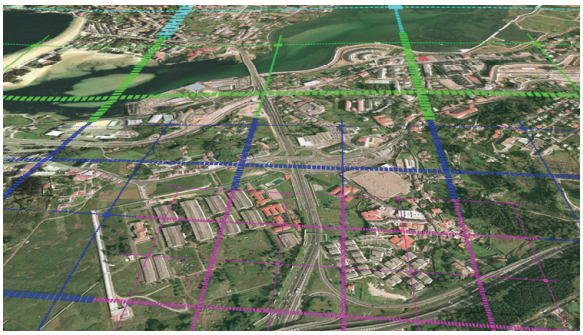

Figure 5: Virtual textures applied to the terrain. Levels of detail are shown using a color code.

To compute the usage of video memory let us supose that m mipmap levels are used for the textures corre-

sponding to those levels of the disk pyramid above the base level. It is computed as

$$V = \left((l - l_b - 1) \cdot \sum_{i=0}^{m} \left(\frac{c}{2^i} \right)^2 + \sum_{i=0}^{l_b+1} 2^{2i} \right) \cdot b$$

4 UPDATE

The data stored on the texture stack, as described in the previous section, corresponds to a zone of the virtual texture around the center of detail. As the center of detail position is moved, it is necessary to update the contents of the stack and other related structures.

Center of detail. For every frame, the application must place the center of detail in the location where higher quality is desired. Several strategies can be used, usually computing it as a function of the camera position and orientation.

The simplest approach is to place the center of detail in the vertical projection of the camera location over the ground. More adequate approaches place the center of detail on a point of the visible geometry close to the camera.

Texture stack update. Each texture of the VRAM texture stack is updated independently, caching a level of the virtual texture. VRAM textures are considered to be divided in square patches, called *subtiles*. The subtile is the texture updating unit. The subtile size (s) must be a divisor of the clip size and the tile size, with

$$s = 2^i,\ t = 2^j,\ c = 2^k, \text{ with } i \leq j \text{ e } i < k$$

After the center of detail is moved, some subtiles will retain useful data, but other tiles must be updated. Each texture has a state matrix that keeps the state of its subtiles. Subtiles that must be loaded with new data are marked in the state matrix as invalid.

Processing the textures of the stack in ascending order, each invalid subtile is loaded from the tile that contains the information it needs at that moment. If the requested tile is in the cache in RAM, the subtile is updated. Otherwise, the subtile remains invalid, waiting for the needed tile to be loaded from disk. In case of incomplete levels, subtiles in the areas where there is no information will never be updated.

To keep the coherence of data, the described subtile update implies updating the related area in every mipmap level of the texture. Update of levels t_{ij}, where $j > 0$, can be made from lower textures in the stack because this data is replicated in several textures (Figures 2 and 3). In this way data can be transferred inside VRAM, faster than loading it from RAM.

There is a toroidal structure of the subtiles in VRAM due to efficiency reasons. Considering the virtual texture levels divided in subtiles, subtile (x_a, y_a) from level i on disk is placed in position (x_b, y_b) inside the texture in VRAM, where

$$(x_b, y_b) = (x_a \bmod K, y_a \bmod K) \text{ and } K = \frac{2^i}{s}$$

Load control. The speed of the center of detail affects the amount of texture that must be updated. The higher detail textures cover a smaller area than the lesser detail ones and they must be updated more frequently. The time required to update the texture stack in VRAM can cause the time available for rendering the frame to be overrun. To avoid this potential problem, it is necessary to restrict the amount of time available for updating the texture stack.

The stack update is made from the coarser texture in ascending order. A texture from the stack must be completely updated before the update of the next texture begins. No matter how quickly the center of detail is moving, there will always be a set of textures fully updated (at least the base level).

Computing the subtile size, it is important to find a tradeoff between an adequate load control and a good transfer rate. The less the subtile size is, the higher the accuracy to measure the update time will be. Even though the subtile update time is strongly dependent on hardware used, usually the smaller sizes have a very poor efficiency.

Concentric rings updating. As the textures in the stack are not always completely updated, it is necessary to decide when a texture is appliable. A simple approach is to discard a texture from the stack until it is completely updated. The problem here is that every time the center of detail is moved the distance of a subtile, it will be invalidated until being completely updated again. This problem is reduced by applying the texture when a partial area of the full texture area is loaded. This subarea is called texture *coverage*.

We update the subtiles in concentric rings, innermost to outermost, so the coverage grows as the subtile rings are updated (Figure 6). This way the texture is useful from the moment it begins to have valid subtiles. Beginning from the center, the highest interest zone is available sooner. Also, the center subtiles are the ones with higher life expectancy.

5 RENDERING

To use the described technique in a real application, we have to do the following tasks

- Apply the texture
- Compute the texture coordinates
- Draw the geometry

There are two possible approaches to doing these tasks:

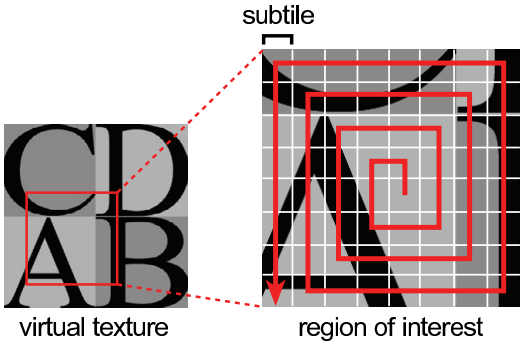

Figure 6: Circular update.

1. For each geometry set, apply the finest available texture covering this region.

2. Apply each texture from the stack, asking for its coverage and drawing the geometry covered by this level but not for the finer ones.

For texture mapping the geometry, we need the virtual texture coordinates bound to each vertex. Texture coordinate computation is made in exactly the same way, no matter which one of the two approaches we choose.

Virtual texture coordinates are converted to coordinates of the texture selected from the stack. Texture coordinates are scaled because each texture from the stack covers half the virtual space of the previous one. Thus, the scale factor for the level i from the stack is computed as $S = 2^i$. We only need the texture matrix of the fixed function pipeline to automatically scale the texture coordinates. Once the coordinates are scaled, the toroidal organization assures that repeated application of the texture [3] will be correctly mapped over the covered surface (Figure 7).

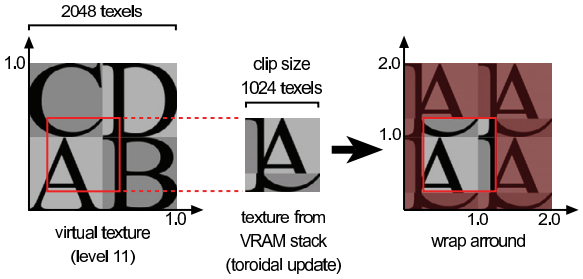

Figure 7: Mapping a texture from the stack.

5.1 Automatic generation of texture coordinates.

In terrain visualization, the usual way of mapping the geometry is to have the texture coordinates precalculated.

The computation of virtual texture coordinates can be done by the texturing system. The only information the texturing system needs to know about the geometry database is the coordinate system used. There can even be different projection systems for texture and geometry.

The real-time computation of these texture coordinates gives us some advantages, one of them being the ability to dinamically modify the geometry while keeping the mapping right.

The programmable capabilities of new graphics pipelines allows us to move these computations from CPU to GPU. This way, we achieve a significant reduction of bandwidth consumption between RAM and VRAM, and avoid the need of VRAM space to store texture coordinates. We can generate the virtual texture coordinates and scale them using a vertex program.

5.2 Multitexturing

An interesting feature of our technique is that several virtual textures can be managed simultaneously. Each one is bound to a texture stage in the graphics system. The maximum number of virtual textures is determined by the graphics hardware used.

In terrain visualization, we can check different types of information over the ground by mapping several overlapping textures (Figure 8).

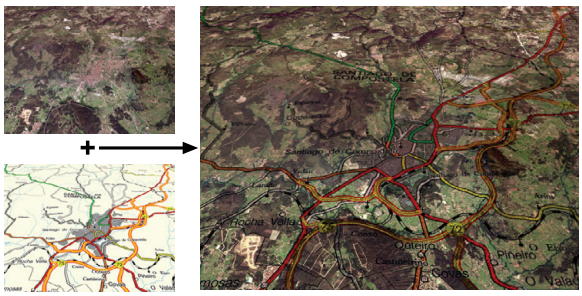

Figure 8: Virtual textures combination.

6 EXAMPLE OF APPLICATION: SANTI

An OpenGL [11] implementation of the technique described above has been applied in the last version of SANTI, a terrain visualization system [8] (Figure 9). SANTI is an application that was developed to display a very large area of Spain as part of a permanent exhibition from 1999 on. It was initially implemented on an SGI Infinite Reality architecture using clipmapping for terrain texture management. Currently it runs on a common personal computer.

This application combines several virtual textures simultaneously, using a vertex program to dynamically compute the texture coordinates.

The system uses an De Boer's [4] inspired algorithm to render the geometry dividing the total mesh into equally sized patches that can be drawn using different levels of detail. There is no constraint on the way the patches are generated. The texture level of detail

is calculated on a block basis, and it has no relation at all with the geometry level of detail. Distant patches are considered as a block in themselves, while close patches are split considering every new subpatch as a block for texturing purposes. Using this approach, the geometry granularity is increased near the camera, to display higher textural detail.

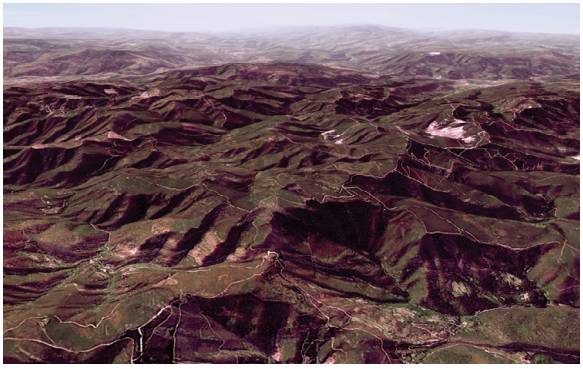

Figure 9: Real-time terrain visualization (SANTI).

This system has been successfully used to visualize real cases of digital terrain models covering more than 60,000 km^2. The most detailed texture used to map this surface has a virtual size of $2^{20} \times 2^{20}$ texels, composing a pyramid with 21 levels of detail. In this pyramid, levels 0 to 16 are generated for the full geographical area, allowing full coverage of the terrain with a 5 meters per texel detail, while for special interest areas levels 17 to 20 are also generated to reach a texture detail of up to 0.25 meters per texel (Figure 10).

Figure 10: View of a 0.25 meters per texel area.

Texure is fragmented in tiles, each of them of 512×512 texels. Clip size measures 2048×2048 texels and the subtile size is 128×128 texels.

The application maintains a steady 75 fps display rate over all the terrain using a personal computer with a regular hardware configuration.

At present, authors are preparing a new 5 terabytes database with 21 full levels of detail that will allow their users to visualize the above mentioned 60,000 Km^2 terrain with a continuous 0.25 meters per texel resolution.

7 RESULTS AND DISCUSSION

The system has been tested with a real data set. These tests were done in a low end personal computer with the following features: Intel Pentium 4 2.8 GHz with 512 MB DDR RAM, GeForce 4 Ti4800SE with 128 MB, AGP 8x, SATA disk 7200 rpm with an approximate bandwidth of 53 MB/s.

The rendered data set includes two combined virtual textures of Galicia (northwest of Spain), simultaneously showing satellite image and a road map (Figure 8) for a 250×250 km area. The resolution of the satellite image is 5 meters per texel over all the area, with one aerial image inset of 0.5 meters per texel, covering a 32×32 km area and other of 0.25 meters per texel, covering a 2.5×2.5 km (21 levels). The resolution of the road map image is 16 meters per texel (15 levels). These images are mapped over a regular 200×200 m cell sized terrain mesh.

Both virtual textures have a clip size of 1024 texels, a subtile size of 128 texels, a color depth of 24 bits per texel. There is a maximum allowed update time of 1 ms per frame for each virtual texture.

The test was a flight over the insets with 0.5 and 0.25 resolution. The speed of flight was 250 meters/s.

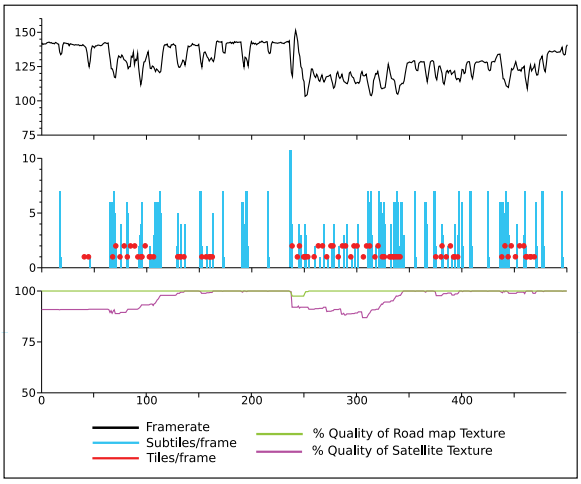

Figure 11: Test results.

The Figure 11 shows a graph for a 4 seconds interval. There we can compare the frame rate with the number of subtile and tile updates, as well as the quality available for both textures. The quality of a texture is measured as the percentage of updating of the texture stack.

As seen in Table1, the average quality of the virtual textures is never under 95%. The update process has little influence on the frame rate, keeping it always over 75 fps.

The update of the texture stack uses an average bandwidth of 7.25 MB/s, approximately 55.29 KB/frame, while the update of the RAM cache needs a transfer rate of 17.4 MB/s, being 32.84% of disk total bandwidth.

Globals	Min.	Max.	Avg.	Std. dev.
Frame rate (frames/s)	91.65	152.05	128.54	11.66
Subtile load (subtiles/frame)	0	14	1.17	2.32
Tile load (tiles/frame)	0	2	0.20	0.51
Tile load latency (ms)	0.96	126.15	15.83	17.62
Terrain texture				
Quality (%)	86.93	100.00	96.00	4.35
Subtile load (subtiles/frame)	0	7	1.13	2.19
Tile load (tiles/sec)	0	2	0.19	0.50
Road map texture				
Quality (%)	97.50	100.00	99.93	0.39
Subtile load (subtiles/frame)	0	7	0.05	2.32
Tile load (tiles/sec)	0	1	0.01	0.51

Table 1: Test statistics.

These results prove that our technique allows the management of multiple virtual textures with exceptional performance.

8 SUMMARY AND FUTURE WORK

We introduce a new technique for managing very large textures through a paging system. It keeps the texture and geometry databases independent of each other and it can be implemented on personal computers using standard graphic API's, like OpenGL, without the need of special hardware.

This technique has been applied on a real case displaying a 60,000 Km^2 terrain texture with a detail of up to 0.25 meters per texel with a steady 75 fps frame rate.

The dynamic, procedural generation of the texture is a research line that is currently open. This could be used for rendering vector data such as GIS information over a three-dimensional geometry, without the need of keeping the texture stored on disk.

Another line of research is to try different alternatives for real time texture decompression [7] [1] in order to reduce the bandwidth needed to transmit those textures through a network for LAN or Internet applications.

REFERENCES

[1] María José Abasolo and Francisco J. Perales López. Wavelet analysis for a new multiresolution model for large-scale textured terrains. *WSCG 2003*, 2003.

[2] J. Blow. Terrain rendering at high levels of detail. In *Proceedings of the Game Developers Conference*, 2000.

[3] Tom McReynolds David Blythe, Brad Grantham and Scott Nelson. Advanced graphics programming techniques using opengl. ACM SIGGRAPH 1998 Course #17 Notes, July 1998.

[4] Willem H. de Boer. Fast terrain rendering using geometrical mipmapping. 2000. Available in http://www.flipcode.com/tutorials/geomipmaps.pdf.

[5] Jürgen Döllner, Konstantin Baumman, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 227–234. IEEE Computer Society Press, 2000.

[6] T. Hüttner. High resolution textures. *Visualization'98 - Late Breaking Hot Topics Papers*, pages 13–17, November 1998.

[7] Young-Su Kwon, In-Cheol Park, and Chong-Min Kyung. Pyramid texture compression and decompression using interpolative vector quantization. In *ICIP*, 2000.

[8] L.Hernández, J.Taibo, and A.Seoane. Una aplicación para la navegación en tiempo real sobre grandes modelos topográficos. In *IX Congreso Español de Informática Gráfica, CEIG 1999.*, 1999.

[9] Boris Rabinovich and Craig Gotsman. Visualization of large terrains in resource-limited computing environments. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 95–102. IEEE Computer Society Press, 1997.

[10] John Rohlf and James Helman. Iris performer: a high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 381–394. ACM Press, 1994.

[11] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Editor (v1.1): Chris Frazier, Editor (v1.2): Jon Leech, March 1998. http://www.opengl.org/.

[12] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158. ACM Press, 1998.

[13] Lance Williams. Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11. ACM Press, 1983.