Dynamic Virtual Textures

Javier Taibo
University of A Coruña, Spain
jtaibo@udc.es

Antonio Seoane
University of A Coruña, Spain
aseoane@udc.es

Luis Hernández
University of A Coruña, Spain
lhernandez@udc.es

Abstract

The real-time rendering of arbitrarily large textures is a problem that has long been studied in terrain visualization. For years, different approaches have been published that have either expensive hardware requirements or other severe limitations in quality, performance or versatility. The biggest problem is usually a strong coupling between geometry and texture, both regarding database structure, as well as LOD management.

This paper presents a new approach to high resolution, real-time texturing of dynamic data that avoids the drawbacks of previous techniques and offers additional possibilities. The most important benefits are: out-of-core texture visualization from dynamic data, efficient per-fragment texture LOD computation, total independence from the geometry engine, high quality filtering and easiness of integration with user custom shaders and multitexturing. Because of its versatility and independence from geometry, the proposed technique can be easily and efficiently applied to any existing terrain geometry engine in a transparent way.

Keywords: Terrain rendering, virtual texture, clipmap, dynamic data, GIS, GPU.

1 INTRODUCTION

Terrain visualization typically involves both high resolution geometry and texture management. There are numerous works on these two areas, whether on one or other or on both. Focusing on the texture management, existing techniques require expensive hardware or establish a strong coupling between geometry and texture, both in databases and LOD management systems. We propose a different technique based on the programming capabilities of new GPU generations, that efficiently solves these limitations as well as provides new features for emerging applications to interactively visualize geographic information. The technique enables real-time rendering of high resolution textures composed of dynamic data that is periodically updated. A working subset of the whole dynamic database is cached in TRAM, in a clipmap-like structure [22], allowing arbitrarily large textures to be mapped over any geometry.

This virtual texturing engine can be applied to terrain visualization as well as to other applications that require a high detail 2D texture focusing on a center of interest. The textured dynamic data can be raster or vectorial in its source.

We start with a state of the art review, mentioning the weaknesses and limitations of previous related work, which have constituted our starting point for working on and improving. We then present a new approach, describing its features, architecture and design throughout section three. Finally, we present some quantitative results from the current test implementation, in order to analyze the performance of the system, followed by the conclusions and future lines of research we are currently working on.

2 RELATED WORK

The use of wide area detailed textures for real-time terrain rendering was first studied by Michael Cosman in 1994 [7]. Two approaches were described: to use a mosaic of small textures of a size supported by the hardware or to use a unique virtual texture managed by specific graphics hardware.

The ideas given by Cosman were later retaken by Tanner et al. when they described the clipmap architecture [22]. This approach was similar as it was based on specific graphics hardware to support arbitrarily large virtual textures with a very limited amount of texture memory. The clipmap technique became quite popular and nowadays it is one of the most important references in the field of real-time terrain texturing.

Later approaches to the large textures problem were published, that avoided the strong hardware requirements of the previous ones. They were mainly based on the texture mosaic approach, and so they suffered most of the limitations described by Cosman. Some of the most important techniques were described by Hüttner (MP-Grid) [13], Rabinovich [19], Cline [6], Döllner [9], Klein [14], Cignoni (BDAM/PBDAM) [4, 5], Brodersen [3] and DachsBacher [8].

The biggest drawback of these techniques is the strong coupling between terrain geometry and texture regarding database structure and run-time LOD selection. Geometry must be tessellated and tile boundaries must exactly match those of the tiles of the texture mosaic.

Lefebvre [15] et al. described a generic texture management system for arbitrary meshes but that does not comply with all the requirements we state, detailed in next section. They use a tile pool texture that does not keep continuity, what difficults high quality filtering.

The clipmap approach has also been adapted to geometry rendering, as in the works of Losasso [16], Holkner [12] and Asirvatham [1]. The use of texturing is proposed in some of these works, but texture LODs are fully coupled with geometry LODs and so they are not chosen based on the texture space projection of each fragment. This is an important drawback not only to LOD selection, but also to texture filtering. Some techniques have been proposed recently, as in Ephanov [10] and Seoane [21], based on the idea of clipmapping and trying to provide its advantages while not requiring expensive specific graphics hardware.

Seoane's technique provides a cached window of the virtual texture, for each texture level, as big as desired up to the texture size limit of the hardware. As it is a roaming window instead of a mosaic of textures, geometry tessellation is not so constrained as in other techniques. Geometry boundaries can be anywhere and as soon as the texture cache window contains a geometric primitive, it can be textured with this level of detail.

This decoupling between geometry and texture management is one of the main advantages of this technique. It can be used with different tessellations, even non-rectangular ones, i.e. TINs. The one important aspect is about geometry batch size, not shape. The problem of this technique is that, in order to achieve the highest texture LODs, geometry must be highly fragmented, which may result in performance loss.

Ephanov et al. propose two alternatives for their Virtual Textures: using the OpenGL fixed function pipeline or using pixel shaders. The first choice is similar to Seoane's approach in its relation between texture LODs and geometry tessellation and in that it does not require programmable hardware. But Ephanov does not use the toroidal update like Tanner and Seoane. Instead of the continuous update of a center of detail, the paging centers hop to positions matching the geometry. This means an undesired coupling between texture and geometry management systems: the texturing engine must know the geometry structure.

The second approach proposed by Ephanov extends the previous scheme by using the programmability and multi-texturing capabilities of new graphics hardware. This way, they can map a geometry primitive using several textures to achieve a higher texture detail for this geometry than was possible with the fixed pipeline. The drawback is that it implicates the use of several texture stages, making it difficult to combine several Virtual Textures or to use them in other advanced rendering techniques without expensive multiple render passes.

Also, the use of mipmaps for level tiles imposes a memory usage overhead for redundant data. Moreover, the use of double buffering duplicates the graphics memory assigned to cache the Virtual Texture.

Ben Garney [11] published a technique that emulates clipmaps with programmable hardware. The main limitation of this technique is that it requires one texture stage for each clipmap stack level used. This drastically limits the number of stack levels that are simultaneously usable and, specially, it difficults the combination of several textures with programmed effects.

Later works on the topic are Mittring's Advanced Virtual Textures [17] and Barret's [2] Sparse Virtual Textures. Both of them need at least an additional texture for cache addressing and do not keep the texture space continuity, what difficults high quality filtering. Moreover, none of these techniques address the management of dynamic textures.

3 VIRTUAL DYNAMIC TEXTURING SYSTEM

3.1 Objectives

The goals established in this work were focused on solving the main drawbacks and limitations described in previously mentioned works, supporting some features required for new applications. The technique complies with the following characteristics:

- It is based on a standard, such as OpenGL 2.0, avoiding the requirement of any vendor specific hardware.
- Texture LOD is computed and applied per fragment, resulting in an efficient cache usage and no drops of detail in geometry boundaries.
- Efficient memory usage. Unlike in some of the previous techniques, no redundant data is to be needed for hardware mipmaps or double buffering.
- Allows for several texture filtering types, including dynamically configurable anisotropic filtering.
- High performance, real-time update and texturing.
- Compatibility of virtual texture with any custom vertex or fragment shader.
- Use of only one hardware texture stage, allowing its combination with custom shaders and high performance multitexturing. This minimizes the number of texture binds (only one) without need to sort the geometry batches.
- Total independence from geometry. Possibility to apply an arbitrarily large texture to a single quad and reach its maximum LOD when the camera approaches the textured surface. This way, it allows the geometry engine to dynamically modify the meshes in size, shape or topology.

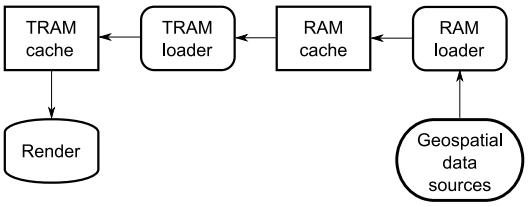

Figure 1: Overall architecture.

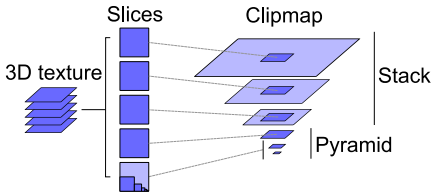

Figure 2: Clipmap structure on a 3D texture.

- As texture contents may be dynamic, cache is updated not only in space but also in time.

In summary, the proposed texturing system provides the features present in systems such as Cosman [7] and Tanner [22], also allowing for new capabilities such as dynamic update, multitexturing and its combination with custom-made shaders. Moreover, the only hardware requirement is a consumer graphics card.

This system can be applied to static or dynamic data and, in both cases, data sources can be raster as well as vectorial rendered to a procedural texture.

3.2 System architecture

The texturing system proposed is based on a two-level cache hierarchy, following the clipmap[22] structure, adapting it to currently available consumer hardware and improving it to support dynamic textures with contents that are continuously updated as a function of time.

The first cache level is the texture subset stored in TRAM, while second level is stored in RAM. Each cache level has an associated component whose task is to load or generate the contents stored in this cache. The overall architecture is illustrated in Figure 1.

Following the clipmap structure, the *pyramid* (complete levels) as well as the *stack* (incomplete levels) are cached on TRAM.

Because the use of a single texture stage is a requirement of the system, as previously mentioned in the objectives, only a single texture could be used to store the cache contents. We chose to use an OpenGL 3D texture to store all cache contents, because it offers several advantages to both quality and performance of the system. Storing each stack level on its own texture slice, continuity is kept so bilinear filtering in the level can be automatically done by hardware without noticeable overhead. Moreover, this continuity simplifies texture addressing. Both advantages assume the toroidal updating and addressing of the stack levels, as described by Tanner. The clipmap pyramid can be stored completely inside two slices in case of square virtual textures or one slice in rectangular ones. The storage of the clipmap structure in a 3D texture is illustrated in Figure 2.

This storage in a single texture is a big advantage over the mosaic of textures approach followed by many other techniques, critical for the independence between texture management and geometry management, regarding structure as well as LOD management. It also frees precious texture units that can be used for other purposes in a custom shader that use the dynamic virtual texture proposed combined with other textures of any kind. This way, a user can access several virtual textures and use them for any purpose beyond just applying them as fragment color. Some examples of interesting applications for virtual textures beyond color are described by Ephanov [10].

The NVIDIA SDK white paper [18] proposes a structure using texture arrays, that can be an alternative to the use of 3D textures, and the algorithms described in our paper can be applied to this structure as well.

3.3 Updating

The cached region of the virtual texture that will be used in the render is computed from the center of detail supplied by the user. As the center of detail is moved over the virtual texture space, the contents of the caches must be replaced to keep the right information.

The update process implies several decisions affecting cache efficiency and so the final visual quality, though they must never affect the real-time performance of the system. To guarantee this premise, update process is divided in both a synchronous and an asynchronous update. The synchronous update uploads texture data to TRAM, competing for time with the render tasks, and so it has time restrictions to avoid frame drops. These time restrictions are dynamically changed in function of the render load.

The synchronous update, that feeds the first level cache, involves several important decisions. First of all, which levels to load, which regions inside each level and, in both cases, the loading order. Apart from that, there are also some other aspects we must take into consideration, such as temporal updating of dynamic data and asynchronous, predictive RAM cache updating to minimize second level cache misses.

Level loading order Regarding the order of loading, the classic bottom-up approach has the advantage of making the larger areas available first and then refining detail as soon as possible in progressively smaller areas around the center of detail. This is the strategy followed by the great majority of systems, including Cosman [7] and Tanner [22]. Although it is the best solution for static data, in a dynamic data texturing system such as

the one proposed, it is not. This scheduling scheme can lead to situations in which the camera is focused on a small detail of the texture that will never be available because coarser levels covering bigger areas are continuously reloaded even though they are not needed at all. This problem is more severe as the updating frequency of the data increases.

Frequently updated dynamic textures require a different approach for the update scheduling. We need to know in the update stage what texture LODs will be required (not in the fragment shader, as before), and that introduces an extra computation for the CPU. The scene must be examined to determine what range of texture LODs will be needed.

The computation of the exact LOD range needed in the render can be complex and costly, and so affect the system performance. We compute a conservative estimation of the LODs needed in function of the distance of the viewer to the textured geometry, the field of view of the camera and the screen resolution. The distance is estimated by casting some rays through the corners and the center of the viewport (and optionally some other samples, depending on the available computing power) to intersect the terrain. This approximation results quite adequate and needs reduced CPU time.

Inter-level loading order Concerning the update inside a texture level, there are two possible approaches for updating the TRAM texture cache of one texture level: updating variably sized regions of invalidated data according to center of detail displacement, as described by Tanner [22], or tessellate the virtual texture space in square tiles of fixed size, as described by Seoane [21].

Although the first approach can gain a bit of performance in some circumstances, while introducing more complexity to the update and specially to the load time control, the second approach guarantees that loaded texture blocks have an adequate size to maximize transfer rate, avoiding inefficient small block transfers and leaving this time available for other tasks in this frame.

For this reason we use the second approach, loading fixed-size square tiles of texture. Tile size is either specified in the configuration or automatically computed at startup time. The adjustment of this parameter is very important for tile loading, as the transfer rate from RAM to TRAM is strongly affected by it.

In case of procedural textures (tile render on-demand), the tile size is also critical because it affects the render efficiency. The usual behavior is that the bigger the block to upload or render, the higher the transfer rate or the rendering efficiency. However, for an adequate load time control, the load/render quantum must not be too big. In the special case of very dynamic textures, modified every frame, the best option is to render each whole level in one pass (tile size = clip size) as long as the render time of a level does not exceed the available update time.

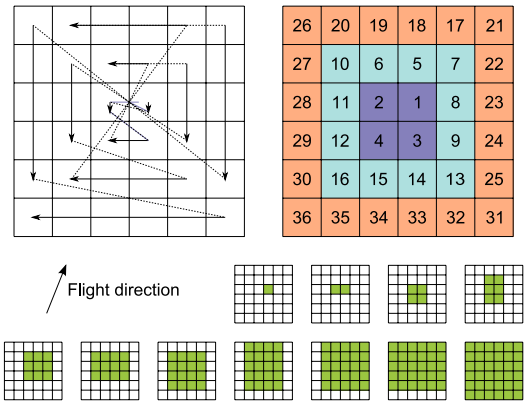

Figure 3: TRAM tiles loading order. Example for a given flight direction.

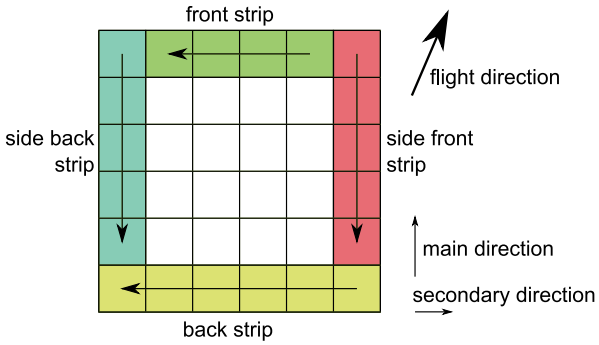

Figure 4: TRAM tiles loading order for a single quad around center of detail. Example for a given flight direction.

Following with the case of loaded tiles (instead of rendered ones), our tests showed that the optimum tile size for loading texture is dependent on the graphics hardware, but it is usually around 128 square texels. It is the smallest size before a severe drop of transfer rate.

Whatever the source of data, raster loaded or vectorial rendered to texture, the tiles loading order inside a texture level is defined with the following goals:

- To prioritize the loading of tiles close to the center of detail, that contain the information located in the region that will capture the user's main attention.
- To progressively load rectangular areas around the center of detail, i.e. to construct, as soon and as big as possible, areas directly usable in the render.
- To prioritize those tiles that, for the current flight direction, will be kept valid for more time, i.e. load first the tiles in the direction of movement.

Following these goals, the sorted list of tiles to load is generated in concentric squares around the center of detail (innermost to outermost), as showed in the example in figure 3. The order in each of these concentric squares is illustrated in figure 4.

Temporal update In case of dynamic textures, cache tiles must be updated, not only when the center of detail is moved, but also in function of time. The time of life of a tile can be determined in two ways. An asynchronous mechanism allows the client of the texturing engine to invalidate any single tile, level or the whole cache. Nevertheless, the usual temporal update is managed through a synchronous mechanism based on an expiry timestamp assigned to each tile. When this expiry time is reached, the data is no longer valid and must be requested again to the next cache level.

This expiry scheme has a problem that produces visible flickering on the render because, when some texture tiles reach their expiration time and so become obsolete, those tiles that could not be updated in this frame will cause a drop of quality in the render.

We solved this problem with a two-level expiration scheme, with hard and soft expiration times. Soft expiration refers to that there is new data available and its loading should be scheduled, but old data can be used meanwhile. Hard expiration implicates that this data is obsolete and while new data is not available, this tile or buffer cannot be used. With this technique, dynamic data will be updated smoothly, keeping the highest level of detail.

The expiration time is established by the data source as part of some metadata attached to each tile, so this information is propagated and taken into consideration in every cache level.

One of the other parameters in this metadata is the "absent" flag that avoids continuous request to the cache pipeline of elements that are not available in the data source. This allows to efficiently manage texture with incomplete levels or heterogeneous detail in the virtual texture. This concept was introduced by Tanner as "high resolution insets".

The dynamic virtual texture can vary the available information through time, so the absent state also has a lifespan assigned and, once expired, this data can be requested again. Apart from supporting dynamic data that change, not only in contents, but also in spatial and detail availability, it allows the use of existing information in a texture database while it is still being generated.

Asynchronous predictive updating First level cache (TRAM) is updated by its loader from the information available in the second level cache (RAM), that structures the virtual texture space in square buffers with a size that is a multiple of TRAM tile size.

As the transfer rate from secondary storage or even network to RAM is much slower than from RAM to TRAM, this will be the bottleneck of the updating. This is why a prediction system, instead of an on-demand approach, must be designed to minimize cache misses and so avoid loss of detail or a slow refinement.

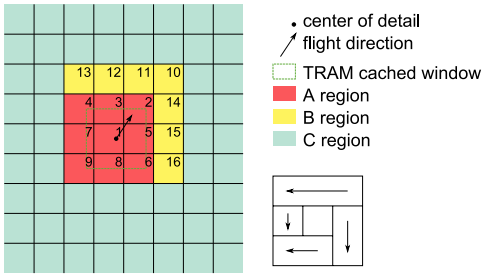

Figure 5: Loading order of RAM cache buffers.

Buffer loading priority is critical. In our experience, the best results have been achieved with the following algorithm (illustrated in Figure 5).

For each texture pyramid level (from coarser to finer), the set of buffers conforming the region that contains the current clipmap stack is computed. This set is called "region A". The L-shaped set of buffers surrounding the region A, following the movement of the center of detail is called "region B".

The buffers in each region are sorted and loaded by priority. Region A buffers are loaded innermost to outermost in concentric rings and each ring is loaded beginning with the buffers in the position towards which the center of detail is moving, in a similar way as described for TRAM tiles. Buffers in region B are loaded from the center to the edge of each arm, beginning with the arm in the main direction of movement. This way, buffers closest to the center of detail are loaded first to have valid data as soon as possible. Also, higher loading priority is assigned to those buffers that presumably have a longer life expectation, because they are in the direction of movement.

Region A contains the buffers immediately needed by TRAMCache, so they have the highest priority. But, there is an important design decision about how to do the loading along the clipmap levels.

Loading region A first for each clipmap level from coarser to finer and then beginning with region B in the same order gives the user the highest detail as soon as possible, but the camera movement can make the clipmap levels invalid very soon.

The other alternative is to load region A and then region B for each level in the clipmap, from coarser to finer. It takes a little more time to reach the highest levels of the clipmap, but once they are loaded they will be much more stable and suffer little or no drops of detail.

The first strategy is more adequate when the camera movement is very slow or when the application needs to show the information very quickly in some points, instead of showing a smooth visualization of a flight. As the behavior is application dependent, we decided to make it configurable and support both strategies. Moreover, this behaviour can be dynamically changed in run time, so RAM cache loading schedule can be set as a

function of the speed and/or kind of movement of the camera.

After completion of regions A and B, as described, the surrounding buffers are prefetched (region C), sorted by proximity and clipmap level, until the memory allocated to the cache has been filled.

Once all the available buffer storage space is filled, an LRU policy is applied to discard old buffers and load new ones. Only region C buffers are in the LRU queue; regions A and B buffers are always kept in RAM.

It is important to state that the RAM cache can be shared between several TRAM caches in applications with several views of the same scene with different centers of interest. Regions A and B are dependent on the client (TRAM cache) and they can be overlapped. To keep these buffers locked in cache, a reference count mechanism is used, so no buffer is sent back to the LRU until it is out of every client region A and B.

3.4 Rendering

The kernel of the texturing system is in the fragment shader code. A GLSL function (VTfetch) is provided to access the dynamic virtual texture inside any user shaders for whatever purpose.

The GLSL source code for this function is generated at run-time, during the initialization phase of the TRAM cache, depending on its configuration. This way, this performance critical code is highly optimized for the exact intended use.

The fragment shader uses several parameters, being the most important the texture coordinate set, received from the vertex shader.

Some of the parameters used are the texture sampler used by TRAM cache, the virtual texture size in texels, the number of levels in the clipmap pyramid and stack, the clip size, the tile size, the wrapping mode of the virtual texture, the LOD offset and the limit of anisotropic samples allowed.

The function, as well as these uniform parameters are prefixed with a texture stage identifier in order to allow several instances of virtual textures to coexist in different texture stages and so be combined in a user custom shader.

The main tasks of VTfetch are to compute the texture levels of detail needed for the fragment, check the availability of the needed texels in the cache and select the most adequate level for the fragment, address the real 3D texture to fetch the needed samples and combine them performing the desired filtering scheme.

The computation of texture LOD is done using the partial derivatives of texture coordinates in screen space. In case of isotropic filtering, it is done in a similar way to the one described in the OpenGL specification [20].

After calculating the LOD, it is clamped down to the immediate equal or lower level present in cache for the texel. The fragment shader needs to know the status of the TRAM cache, i.e. the availability of each texel in the virtual texture full pyramid. This information is communicated to the fragment shader through a set of parameters containing the rectangular valid area for each level, computed by the update process described before.

How many texels and in what coordinates they are fetched depends on the filter used for the virtual texture. To avoid interpolation errors because of hardware bilinear filtering on the 3D texture, a border of half a texel around the valid area is considered unavailable, which means that immediate coarser level texels will be used instead. Supported filters include nearest neighbor, bilinear, trilinear and anisotropic.

As texture fetches are the most time consuming operation in the fragment shader, the number of anisotropic samples is limited by an uniform parameter that can be dynamically updated each frame, depending on the current system stress.

3.5 Stress management

A stress management system was developed with the objective of sustaining the frame rate of the visualization. It has, among others, the responsibility of distributing the available update time for each frame between the virtual textures. This way, the update time adapts to the time available until the next screen buffers swap and textures can be prioritized in function of their relevance or update status.

It can also dynamically adjust some of the quality parameters, such as the number of anisotropic samples, in function of the system stress to avoid frame drops. As the limit of anisotropic samples is a uniform parameter to the fragment shader, it can be changed every frame with no cost.

Monitoring the camera movement enables the rendering engine to increase quality (e.g. by improving the filtering) when the camera stops, in applications that are not frame rate critical and can allow frame drops in this situation, or to change the update scheduling strategy as described before.

3.6 Scalability issues

When virtual texture size is big (for example texturing the whole planet with submetric detail), several problems arise, affecting quality as well as performance and required resources:

- Memory requirements increase due to the amount of stack levels in the cache.
- Update time increases for the same reason.
- 32-bit numeric precision of graphics hardware is insufficient to accurately address the virtual space, producing visible deviations in the texture coordinates.

Filter	Render time
Nearest neigbour	1.6 ms
Bilinear	1.6 ms
Trilinear	1.6 ms
Anisotropic (1 sample)	1.7 ms
Anisotropic (2 samples)	3 ms
Anisotropic (4 samples)	4.4 ms
Anisotropic (8 samples)	5.4 ms
Anisotropic (16 samples)	7.3 ms
Anisotropic (32 samples)	8.5 ms
Anisotropic (64 samples)	9.9 ms
Anisotropic (128 samples)	11 ms

Table 1: Rendering performance.

In such situations, we solve these problems by not caching all the stack levels, but only a subset (floating stack) that is located depending on the proximity between the camera and the textured objects. This placement of this window of levels is computed in a similar way to the one mentioned for the level loading order in very dynamic textures. Apart from the reduction of levels stored, the texture area managed by the texturing system is reduced to an extension between the addressing limits of 32-bit arithmetic precision of the hardware.

Therefore, the application updates the location of this floating stack, as well as the center of detail position. The active texture area can be supplied by the geometry engine (that will suffer the same precision problems for the vertex coordinates and so it must work in local coordinate systems) or it can be automatically computed by the texturing engine following the position of the center of detail. The update of the floating stack as the cached window is moved, is performed in a circular way to minimize memory transfers and optimize performance.

4 RESULTS

The current implementation of the described technique has been integrated in our real-time terrain visualization system in order to test it in a real production environment. For debugging, testing and verification purposes, a standalone texture visualization tool was also developed, that provides detailed graphic information about the internal state of the texturing engine. All the tests presented in this section have been performed on an NVIDIA GeForce 8800 Ultra with driver version 1.4.0.90 for Linux 32-bit.

For the rendering performance and quality analysis, we used a 1048576×1048576 texels (20 levels) texture with its cache fully updated, so no texture loads could affect performance. The cached window for each level (clip size) was 2048×2048 texels. This virtual texture was mapped to a frame filling plane viewed from a shallow angle to force a highly anisotropic situation. Screen resolution was 1280x1024 pixels. In table 1 we can see performance measurements for the different supported filter types and, in the case of anisotropic filtering, different number of samples.

For the testing of update performance, a dynamic texture with high update ratio was used. Texture virtual size was 131072×65536 (17 levels), with a clip size of 1024×1024 and a TRAM cache tile size of 128×128.

The data source was taken from NASA Blue Marble data set, showing the Earth appearance along a year. Texture update rate was set to one second, with a grace period of another second to reload the information before invalidating it.

This dynamic texture was applied to a model of the Earth globe and moved around, zooming in and out to examine different places as in a typical usage.

The update time assigned for each frame was 3 ms, data was accessed through a network, and cache contents were completely replaced every second. The highest detail was available for the most time, especially with still camera or slow movements. Drops of detail matched the fast movements of the camera (and so the center of detail) but they were barely noticeable because in these situations the user is far away and higher levels are not applied.

The size of the cached window or clip size is an important decision. With static data it is beneficial to have a higher size, because it allows to make fast moves, keeping the maximum detail, but with dynamic data it can be counterproductive, because update time is wasted with information that will expire before being used. In highly dynamic information, it is very critical to adjust the clip size to the minimum required for the screen resolution. We usually chose 1024 for high update ratios (near to a second) and 2048 for low update ratios or static data.

The scalability of the system was successfully tested with a static virtual texture of $2^{27} \times 2^{26}$ texels covering the whole planet, reaching a resolution of 0.25 m/texel in the area of highest detail.

5 CONCLUSION AND FUTURE WORK

We have developed a geospecifical, dynamic, virtual texturing engine that fulfills the objectives proposed in section 3.1 and we are beginning to successfully test the system in real environments. We have focused on terrain texture or similar applications, where detail is located around a unique area, and not on other applications that need sparse detail textures. Planetary sized dynamic textures with submetric resolution are supported through the virtualization mechanisms described.

One of the most important achievements of the proposed system is to offer all the previously mentioned features while keeping full geometry independence. This allows homogeneous, high quality aerial image to be mapped over irregularly tessellated terrain, enabling us to use this virtual texturing engine in applications like the one shown in Fig. 6, where terrain geometry

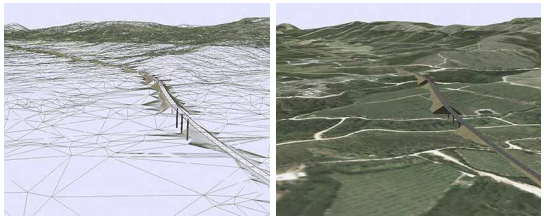

Figure 6: Texturing a TIN-based terrain with embedded 3D models.

Figure 7: Examples of use of the texturing engine.

cannot have a regular tessellation because it must be accurately adapted to a 3D model of a highway.

Multitexture capabilities have been tested blending raster virtual textures together and blending them with vectorial data rendered to another virtual texture, such as technical drawings or GIS layers. Figure 7 shows these examples, as well as the use of the texturing engine within a shader that desaturates some regions depending on their color.

Concerning dynamic update of textures, we have found that quantitative results have outperformed the needs of real applications managing dynamic geographic information, where usual update cycles do not fall below one minute.

We are exploring the benefits of the described technique in some fields of application, combining visualization of high resolution aerial image with dynamic raser and vectorial data over 3D terrain models. This includes projects in real-time traffic management, urban planning, infrastructure project analysis and fire extinction, among others.

REFERENCES

[1] Arul Prakash Asirvatham and Hugues Hoppe. Terrain rendering using gpu-based geometry clipmaps, 2005.

[2] Sean Barrett. Sparse virtual textures. http://silverspaceship.com/src/svt/, 2008.

[3] Anders Brodersen. Real-time visualization of large textured terrains. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 439–442, New York, NY, USA, 2005. ACM Press.

[4] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Bdam – batched dynamic adaptive meshes for high performance terrain visualization, 2003.

[5] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Planet-sized batched dynamic adaptive meshes (p-bdam). In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 20, Washington, DC, USA, 2003. IEEE Computer Society.

[6] David Cline and Parris K. Egbert. Interactive display of very large textures. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 343–350, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[7] Michael A. Cosman. Global terrain texture: Lowering the cost. In Eric G. Monroe, editor, *Proceedings of 1994 IMAGE VII Conference*, pages 53–64. The IMAGE Society, 1994.

[8] Carsten Dachsbacher. *Interactive Terrain Rendering: Towards Realism with Procedural Models and Graphics Hardware*. Universität Erlangen, 2006.

[9] 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, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[10] Anton Ephanov and Chris Coleman. Virtual texture: A large area raster resource for the gpu. In *The Interservice/Industry Training, Simulation and Education Conference (I/ITSEC)*. I/ITSEC, 2006.

[11] Ben Garney. *Game Programming Gems 7*, chapter Clipmapping on SM1.1 and Higher, pages 413–422. Charles River Media, 2008.

[12] Alex Holkner. Hardware based terrain clipmapping, 2004.

[13] Tobias Hüttner. High resolution textures. In *Visualization '98 - Late Breaking Hot Topics Papers*, pages 13–17, November 1998.

[14] Reinhard Klein and Andreas Schilling. Efficient multiresolution models for progressive terrain rendering. *it - Information Technology*, 44(6):314–321, 2002.

[15] Sylvain Lefebvre, Jerome Darbon, and Fabrice Neyret. Unified texture management for arbitrary meshes. Technical Report RR5210-, INRIA, may 2004.

[16] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM Press.

[17] Martin Mittring and Crytek GmbH. Advanced virtual texture topics. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 23–51, New York, NY, USA, 2008. ACM.

[18] NVIDIA. Clipmaps - white paper (wp-03017-001_v01), february 2007.

[19] 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, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[20] Mark Segal and Kurt Akeley. The opengl graphics system: A specification (versión 2.1). Technical report, 2006.

[21] Antonio Seoane, Javier Taibo, Luis Hernández, Rubén López, and Alberto Jaspe. Hardware independent clipmapping. In *WSCG '2007: The 15th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2007 - Full Papers Proceedings II*, pages 177–183. Eurographics Association, 2007.

[22] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM Press.