INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Interactive GigaVoxels

Cyril Crassin — Fabrice Neyret — Sylvain Lefebvre
ARTIS/LJK-INRIA EVASION/LJK-INRIA REVES/INRIA

http://artis.imag.fr/Membres/Cyril.Crassin/GigaVoxels

N° 6567

Juin 2008

Thème COG

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6567--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

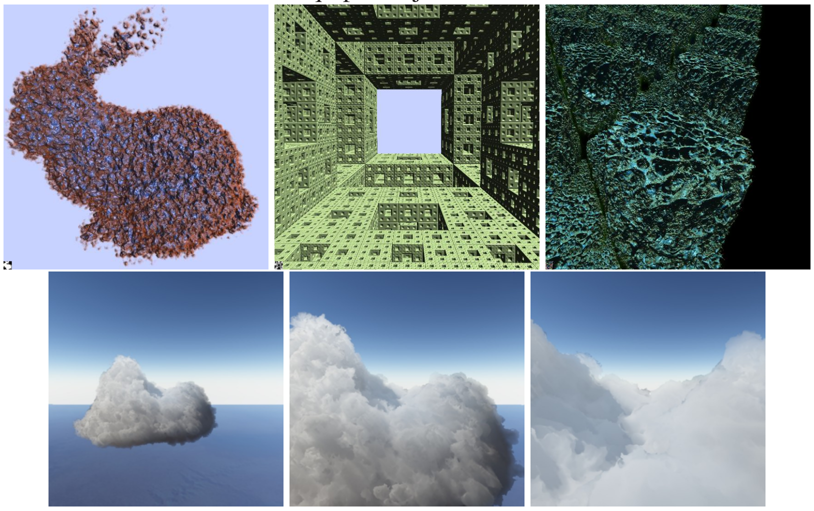
centre de recherche GRENOBLE - RHÔNE-ALPES

Interactive GigaVoxels

Cyril Crassin , Fabrice Neyret , Sylvain Lefebvre
ARTIS/LJK-INRIA EVASION/LJK-INRIA REVES/INRIA
`http://artis.imag.fr/Membres/Cyril.Crassin/GigaVoxels`

Thème COG — Systèmes cognitifs

Équipe-Projet ARTIS



Rapport de recherche n° 6567 — Juin 2008 — 20 pages

Abstract: We propose a new approach for the interactive rendering of large highly detailed scenes. It is based on a new representation and algorithm for large and detailed volume data, especially well suited to cases where detail is concentrated at the interface between free space and clusters of density. This is for instance the case with cloudy sky, landscape, as well as data currently represented as hypertextures or volumetric textures. Existing approaches do not efficiently store, manage and render such data, especially at high resolution and over large extents.

Our method is based on a dynamic generalized octree with MIP-mapped 3D texture bricks in its leaves. Data is stored only for visible regions at the current viewpoint, at the appropriate resolution. Since our target scenes contain many sparse opaque clusters, this maintains low memory and bandwidth consumption during exploration. Raymarching allows to quickly stops when reaching opaque regions. Also, we efficiently skip areas of constant density. A key originality of our algorithm is that it directly relies on the ray-marcher to detect missing data. The march along every ray in every pixel

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

may be interrupted while data is generated or loaded. It hence achieves interactive performance on very large volume data sets. Both our data structure and algorithm are well-fitted to modern GPUs.

We demonstrate our approach with several typical situations: exploration of a 3D scan (8192^3 resolution), of hypertextured meshes (16384^3 virtual resolution), and of a Sierpinski sponge ($8.4M^3$ virtual resolution), all rendered at an interactive frame-rate of 10 to 20 fps and fitting the limited GPU memory budget.

Key-words: real-time rendering, volumes, GPU, hypertextures, visibility, ray-tracing

Interactive GigaVoxels

Résumé : Nous proposons une nouvelle approche pour le rendu interactif de scènes vastes et extrêmement détaillées. Cette approche est basée sur une nouvelle représentation et de nouveaux algorithmes adaptés à ce type de données. Cette représentation est spécialement adaptée aux cas où les détails sont concentrés sur une interface entre de l'espace vide et des zones denses, par exemple penser un ciel nuageux, un paysage, ou encore des matériaux actuellement représentées avec des cartes de relief, des hypertextures ou des textures volumiques. Les approches existantes ne permettent pas de stocker, gérer et rendre efficacement de telles données, particulièrement à hautes résolutions ou sur de vastes étendues.

Notre méthode est basée sur l'utilisation d'un octree généralisé stockant dans es feuilles des briques de textures 3D mip-mappées. Seules sont stockées les données des régions visibles pour le point de vue courant, à la résolution appropriée. Du fait que les scènes qui nous intéressent sont composées de groupements de matière opaque éparses, ceci permet de maintenir une consommation mémoire ainsi qu'une faible utilisation de la bande passante pendant l'exploration. L'utilisation d'un algorithme de rendu de type ray-marching permet d'arrêter rapidement le rendu lors ce qu'une région opaque est atteinte. De plus, notre méthode permet de parcourir efficacement les régions de densité constante.

Une contribution importante de notre algorithme est le fait qu'il utilise directement le ray-marching pour déterminer les données nécessaires pour un point de vue donné. De plus, notre méthode permet d'atteindre des performances interactives pour de très larges volumes de données, notre structure de données aussi bien que que notre algorithme tirant parti intensivement des GPU modernes.

Nous illustrons notre approche dans différentes situations typiques: l'exploration de données scannées 3D (résolution 8192^3), de maillages hyper-texturés (résolution virtuelle 16384^3), et d'une éponge de Sierpinski (résolution virtuelle $8.4M^3$), le tout rendu à une cadence interactif de 10 à 20 FPS et se contentant de la quantité limitée de mémoire présente sur le GPU.

Mots-clés : rendu temps réel, volumes, GPU, hypertextures, visibilité, ray-tracing

1 Introduction

Special effects and games are now required to model and render scenes covering very large areas and containing increasing amounts of details. In special effects, volume representations are often considered for complex fuzzy data such as clouds, spray, smoke and foam, as seen in recent movies (*e.g., XXX, Lords of the Ring, the day after tomorrow, Pirates of the Caribbean*...). These are known under the generic name of *voxel engines* [Kis98, BTG03, Kap02, Kap03, KH05]. Sometimes, volumes are also used for complex shapes such as vegetation, fur and pseudo-surface[1] details.
Our goal is to improve efficiency of this approach in the context of special effects (at least for preview), and to bring it to the context of interactive rendering: Our works aims at making volume rendering a rich and convenient primitive for very detailed CG scenes.

The main issue with volume data is that it usually requires a *lot of memory*, thus *limiting the scene extent and the resolution of details*, or the *streaming of data* from larger and slower memory units. This is even worse for real-time applications since the GPU memory budget is a lot more limited than for software image production. The transfer time of this amount of data (640MB on modern GPUs) from the CPU already prevents real-time performance so brute-force streaming is not a solution. In addition, the *rendering of volumes is costly* due to the amount of voxels to visit, shade and blend. This strongly limits its use in real-time applications as well as for the previsualization and adjustment of volume-based effects in production. Dedicated representations such as bidirectional textures, hypertextures or volumetric textures embed explicit or implicit volume data within a limited interface layer upon objects surface. Current approaches achieve efficient storage and rendering under the condition that this volume layer remains small on the screen (*i.e.,* no zooming).

Still, in usual scenes *details are mostly concentrated* at some locations such as interfaces between dense and clear regions (e.g. cloud, water, landscape, fur). For a lot of pseudo-surfaces, *light rays stop quickly* once inside the dense body. Also, *only visible details are needed* for a given frame, at no more than the sufficient resolution. *I.e., our target scenes have lot of empty space, core regions, lot of occlusion, and mostly the pseudo-surface of objects is seen*.
We propose a new representation and rendering scheme taking advantage of these properties. Our approach manages and renders such volume data at huge resolutions with interactive frame rates, allowing to zoom and explore through details. This is achieved through lazy and adaptive data transfer and rendering: we determine on the fly the minimal necessary data to consider, accounting for empty or low frequency regions, level of detail (LOD) and occlusion information.

Our contributions are as follows:

- A new adaptive spatial data structure for voxel data, allowing efficient dynamic update and traversal. It is especially well fitted to the exploration of large detailed scenes characterized by clusters of density, a frequent case in CG.
- A new ray-marching algorithm that can be seamlessly interrupted and restarted whenever new data must be loaded, over the course of a single frame. It is designed for efficient implementation on SIMD architectures.

[1] A pseudo-surface is an interface which resemble a surface at distance, but which appears to be locally non-heightfield, non-connected or non-opaque at close view. *E.g.,* fur, tree foliage, cumulus clouds...

- A new way to take advantage of occlusions without complex anticipated determination of visibility.

Together, these contributions result in a new framework for interactive rendering of large detailed volume scenes. This includes the management of high amounts of multiscale details, on-demand generation, procedural generation and data amplification. Our framework is is highly inspired by voxel engine tools used in special effect production: it brings to interactive time features which are known to be especially time and memory consuming even in the scope of production. A key to the success of our framework is the early design choice to focus on scenes with clusters of densities, a case very common in Computer Graphics.

2 Previous Work

Voxels
Besides classical volume rendering for scientific visualization, special effects companies such as *Digital domain, Cinesite* or *Rhythm 'n Hues* now massively rely on *voxel engines* [Kis98, BTG03, Kap02, Kap03, KH05] to render very complex scenes. Clouds, smoke, foam and even sometimes non-fuzzy but extremely detailed geometric data (*e.g.,* the boat of *Pirates of the Caribbean*) are all captured with volume rendering. For instance, modeled or scanned meshes, hypertextures on top of particles (*e.g.,* avalanche of *XXX*), sprays and gaseous data are all converted into voxel data before rendering, possibly *amplified* with volume details (*e.g.,* the river of LoTR). The scene size and resolution of details is so large that voxels sometimes don't even fit in the computer memory. In addition to storage, the rendering of such data is also extremely costly, even for preview.

Apart from applications based on full grids of voxels, various specialized explicit or implicit volume-based representations have been proposed to benefit from the visual complexity allowed by volumes: Fur, vegetation, pseudo-surfaces, etc. To avoid the issues mentioned in introduction, they often rely on the assumption that the volume data is embedded in a layer at the interface between empty space and filled space. Hypertextures [PH89] evaluate the procedural opacity on the fly, which trades memory for computation cost. Volumetric textures and shell maps [KK89, Ney98, PBFJ05] rely on a tiling of a volume pattern within the layer. Bidirectional textures [TZL*02] store the local aspect for all view and light conditions. Some recent extensions of relief maps are indeed ray-marchers [BD06b] and rely on optimization structures [CS94] to accelerate empty space traversal.
In addition, volume data structures have also been used to encode non volume data: e.g., texture information along a surface without the need for a planar parameterization [BD02, DGPR02, LSK*06, LHN05b, LH06]. In particular, the *brick maps* data structure of Per Christensen [CB04] captures surface irradiance data in an octree storing 3D volumes in its nodes.
Among all these approaches, several are well adapted to GPUs and provide interactive to real-time performance [DN04, LHN05b, LH06, BD06b, BD06a, LD07].

Work has also been done to bring traditional volume rendering to real-time: factorized rendering using slicing [LL94], compression of empty space [KE02], ray-marching algorithms adapted to modern GPUs [Sch05]. Performances are currently still low, and the volume that can be managed is seriously limited by the GPU memory budget (typically 512^3, or 384 MB). Slices and rays have opposite advantages and drawbacks. Accumulating volume slices yields a lot of overdraw, and numerous hidden voxels are rasterized anyway. Conversely, ray-casters allow to sample and stop each ray at different depths but they are more complex and less data-coherent, so that the occurrence of GPU ray-casters has not yet reached its promises in terms of performances. In [Sch05], the ray-traversal is done on the GPU after determining the closest and furthest fragment for each pixel. Empty space is compressed in memory using an indirection table as in [KE02], but the ray is sampled regularly even in empty or low-frequency regions. Several hierarchical data-structures have been proposed for ray-tracing of polygonal scenes on the GPU [HSHH07]. However, the complexity and cost of marching along rays is greatly reduced by the fact that meshes are opaque in their case.

Our approach builds on the compact hierarchical structures on the GPU by [DGPR02, LHN05b] and adapts them to the storage of sparse volumes. Our structure shares similarities with brick maps [CB04] but ours is dynamic, and as a consequence its content is view-dependent (through LOD and visibility). We hence encode complex volumes efficiently and adaptively, enabling fast ray-marching directly on the GPU.

Dynamic loading
Real-time rendering of very large terrains has the same requirement of having to manage a large scene visible at all levels of details through perspective. Here, level of details (LOD) approaches adapt the memory requirement to the visual needs. Several methods have been proposed to dynamically load terrain patches on demand [LH04, AH05, LDN04].
An aspect of our work is the extension of such approaches to volume data. Adding a third dimension is of course much more involved in terms of memory management.

In 2.5D (terrains) and even more in 3D scenes, a lot of data is occluded: detecting and rendering only visible data is a crucial source of resource saving. Conservative prediction of visibility in scenes is an important and difficult topic in Computer Graphics.

Since we rely on volume ray-tracing, taking advantage of occlusion comes for free: the ray-casting marches the data in depth order, which order is directly provided by the volume structure. Our dynamic ray-tracing will directly point out the necessary tiles to be loaded.

3 Overview

A lot of scenes in Computer Graphics – and especially the ones we target – are composed of sharp or fuzzy objects lying in mostly empty space. Our scheme is optimized for such an assumption: We expect details to be concentrated at interfaces[2] between dense clusters and free space, *i.e.,* “sparse twice”. More generally, we deal with high-frequency interfaces between regions of low frequency within the density field: We treat constant areas (*e.g.,* core regions) just like empty areas. (see Figure 1). Our key idea is to exploit this assumption in the design of our storage, rendering, and visibility determination algorithms, together with the usual assumptions concerning the exploration modalities so as to ensure reasonable time coherency.

[2] Note that all real interfaces are fuzzy at some point since opacity is also a question of scale: Physically, very thin pieces of opaque material are transparent, and light always enters a few steps in matter. Moreover, in terms of LOD the correct filtering of infra-resolution occluders yields semi-transparency.

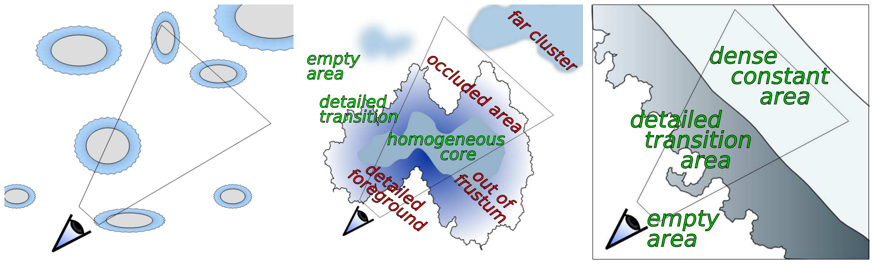


Figure 1: *Details tend to be concentrated at interfaces between dense clusters and empty space.*

Our spatial data structure is a N^3-tree (a generalized octree) storing MIP-mapped 3D texture bricks in its leaves (see Figure 2,Left). The tree hierarchy is mirrored on the CPU and the GPU, the 3D bricks are stored in a pool (*i.e.,* a big texture) on the GPU. We describe it in Section 4. Rendering is performed by directly ray-marching the data structure on GPU. It can interrupt and resume when some data is missing, on a per-pixel basis (see Figure 3). Each pass of rendering results in a list of missing tiles to query. This is described in Section 5 and Figure 5. Bricks are dynamically generated on demand by a *producer* either on CPU or GPU. An *oracle* based on priori knowledge on the scene layout allows to predict empty (or constant) tiles which require no brick production (without it, rendering sparse scenes would be very costly). Producer and oracle are described in Section 6.

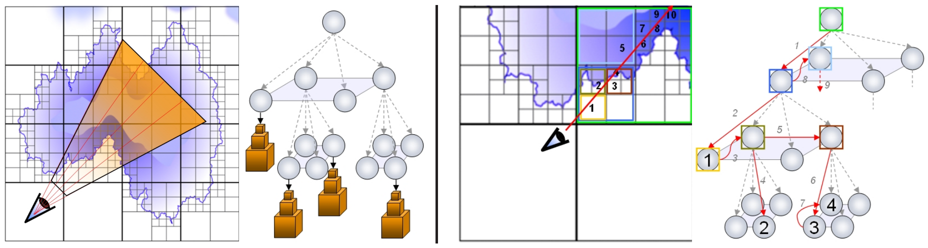


Figure 2: (Left:) *Our hybrid spatial structure combines a N^3 tree and MIP-mapped 3D texture tiles.* (Right:) *Traversal of the hybrid structure along a ray (illustrated in 2D with recursive DDA).*

Terminology

In the following, *tiles* refer to the spatial regions associated to tree *nodes*. These are aligned with the regular subdivision of the object or scene bounding box. Volume data is represented only at tree *leaves*, knowing that the tree subdivision is adaptive (to distance, visibility and content). Thus at any time, all regions of space are represented somewhere in the tree at a given subdivision level: leaves tile the space.

Bricks refer to voxel grids storing the volume density of leaf tiles whenever necessary (*i.e.,* when visible and not constant). Note that not all leaves contain bricks, since we deferred the production of these up to the moment where their visibility is certain. For brevity, we might anyway speak of “parent of a tile” or “producing a tile”.

Nodes and bricks are stored on GPU in two large *pools*. These resemble indirect textures packing tiles [KE02, LN03, CB04]. But differently to the static use of these, the content of our pools change dynamically so as to store all the necessary data in a context where all the existing data cannot fit the memory. Even more, we use these pools as *caches* through an LRU[3] mechanism.

Overall algorithm

At any time, the leaves of our tree structure tile the bounding box of the scene (adaptively, at the minimum useful resolution according to distance, visibility and content). Some contain data, some not (these are marked "invalid"). The content of our structure is inherited from frame to frame. At the beginning of a new frame, the tree leaf nodes intersecting the view frustum are re-validated in order to mark the ones which no longer fit the new viewing conditions (frustum, LOD). The later will have to be regenerated (leaf node simplification or subdivision, brick production) if they happen to be visible, which is deferred up to the moment where ray-tracing queries it (*i.e.,* they are proven visible). At the end of the frame, the tree can be purged from unused bricks (*e.g.,* occluded). This is done lazily (LRU table and occasional tests) for efficiency. The pseudo-algorithm is given on Figure 6.

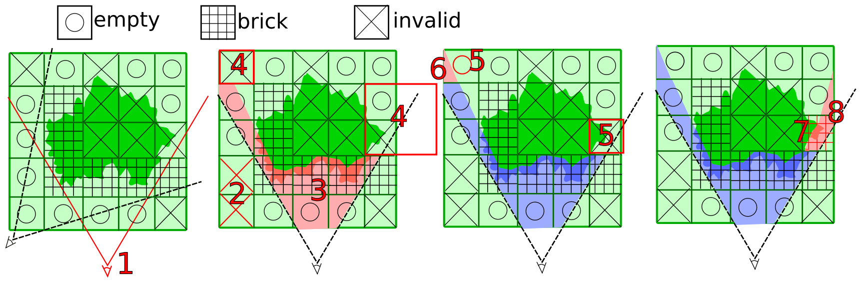


Figure 3: *Interleaved progression of the 3 tasks (rendering, tile queries, production).* From left to right*: At a new frame after a camera move (1), the tree is inherited. After revalidation of the tree, some tiles (area #2) are invalidated (*e.g.*, need simplification or subdivision). A first rendering pass traverses the valid data. Each ray goes as far as possible (area #3) and stops once opacity is full or data is missing. The list of missing tiles (areas #4) is gathered and transmitted back to CPU to query them. Some will be declared as constant by the oracle (#5,left). Some will yield a brick production (if not already available in cache) and its loading to GPU (#5,right). Some will require a subdivision. Then rendering is resumed (rays area on #6), and the cycle repeats: query and production of missing tiles (area #7), rendering resumed (area #8).*

4 Our volume data structure

4.1 Structure

kd-trees are known to be one on the most efficient structures to organize polygonal scenes for ray-tracing [Hav00]. But their efficiency depends on a complex analysis of data spreading. Moreover, these trees tend to be very deep. This yields costly traversal. The regular subdivision of N^3 trees (*i.e.,* generalized octrees) is more convenient to build and traverse, and is more GPU-friendly [LHN05b, LHN05a]. Also, it allows to easily trade between memory efficiency (low N, deep tree) and walk efficiency (large N, shallow tree).

[3]For *Last Recently Used*: each entry has a time stamp reseted when data is used. Allocation use the LRU entry.

On the other hand, regular grids are known to be the best structure for even distributions. Indeed, the best GPU data structure for dense blocks of voxels is a 3D texture: storage is compact, traversal is done simply by regular sampling of a ray, linear interpolation is ensured by the hardware.
Our representation combines the best of each: We encode our scenes with an N^3 tree, storing blocks of M^3 voxels – which we refer as *bricks* – in its leaves. . We rely on MIP-mapped bricks, since the LOD level may arbitrarily vary along a ray depending on the viewing conditions. As illustrated in Figure 2,Left, bricks naturally tend to be concentrated at interfaces between regions of constant density.

4.2 Implementation of the structure

The N^3 tree is stored both on the GPU side – walked during rendering – and on the CPU side – to monitor the updating of the data on the GPU. Similarly to [LHN05a], our tree nodes are stored on the GPU in a large 3D texture, which we refer to as the *node pool*. Each node is a N^3 block stored somewhere into the node pool. Each block entry stores (see Figure 4):

- A bit to tell whether the node is a leaf;
- A bit to tell whether the leaf is valid (*i.e.,* currently subdivided at the correct LOD and containing up-to-date data);
- A bit to tell whether a leaf content is constant (*e.g.,* empty or core regions) or described through a brick;
- the data, which is either:
  - A pointer toward a child node for non-leaves;
  - The average value at this location for constant leaves;
  - A pointer toward an M^3 brick for non-constant leaves.

Pointers are encoded as (u,v,w) coordinates within 3D textures. Similarly to the nodes, the MIP-mapped M^3 bricks are all stored within another large 3D texture which we refer to as the *brick pool*. If the producer is on CPU, we optionally maintain a larger version of this pool on CPU side to be used as a cache.

In order to avoid wrong interpolation of samples close to the border of the bricks, a band-guard of 1 voxel is added (*i.e.,* border voxels are duplicated in neighbor bricks). This principle is not friendly with usual MIP-map pyramids since adding a band of 1 voxel at level l would require to store a band of 2^l voxel at base level. We prefer to manage the MIP-map pyramid "manually": We store l MIP-map levels as l separate 3D bricks of resolution $(2^i+2)^3$ stored in l different brick pools. We need only one pointer in the node since the (u,v,w) is the same in the l pools. See Figure 4. Only 3 levels of MIP-map are required since the leave is already an LOD.

4.3 Structure update

Our data structure is updated dynamically: only visible bricks at useful LODs are stored. The LOD of visible non constant tiles (*i.e.,* containing a brick) is calculated in function of the viewpoint according to the usual MIP-mapped LOD calculation. If more resolution is requested for a brick, this is obtained through subdivision of the node (a reminder that our brick size is constant). Constant density regions (no brick) and low frequency regions are never subdivided. Note that while several of the N^3 children of a given leaf node might be constant; only the non–constant ones will require to store a M^3 data brick. Update of the tree structure is done simultaneously on the

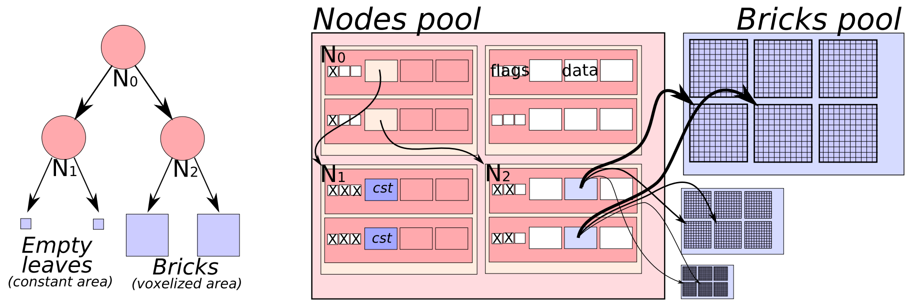


Figure 4: *Our hybrid N^3-tree+brick structure (illustrated as a 2^1 bin-tree instead of N^3 for clarity).*

CPU and the GPU. The CPU tree contains a bit more information to facilitate data structure maintainance and might rely on a larger pool to be used as a cache. The pools counterpart on GPU are updated through `subTexture` operations.

Checking and cleaning old tiles out of the GPU pools immediately could incur a large penalty, especially as they might be visible again soon. Instead, we rely on a LRU mechanism and we use the pools as a cache: for each pool we maintain on the CPU a list of block pointers sorted by date of last use. The time stamp of a tile is reseted when its visibility is confirmed by the ray-tracer (see Section 5). The last pointer in the list is recycled for the next allocation so that during exploration the oldest tiles are recycled first.

Inserting or updating a data brick is done in the same way as nodes in the corresponding pool. However, preparing the voxel data for a brick is not trivial and can be a slow process. It is ensured by a *producer* (see Section 6).

Initialization at a new frame
At every new frame if the point of view has changed, new tiles become visible while some are no longer. Moreover, the LOD to be used for leaves at a given location might require simplification or subdivision of the leaf node so that the current tree is no longer correct.

Before rendering we visit all leaves intersecting the new view frustum. We mark the leaf as "invalid" if its LOD has changed: It must be subdivided or simplified. (In the first case, subdivision is deferred. In the second case, we can directly identify the parent node at the right LOD, make it a leaf, and mark this one as invalid). Recall that the tree covers the entire scene extent: There is no concept of missing region, only leaves at the wrong LOD or not yet provided with a brick of data. So we don't need to worry about leaves leaving the view frustum since the LRU mechanism handles them. Similarly, leaves entering the view frustum will already be invalid if not at the right LOD or if the brick has been recycled. The valid tiles remaining correspond to bricks already on GPU and likely to be used during rendering. (If really used, their time stamp will be reseted in the LRU).

This makes our rendering conservative as it prevents any out-of-date region to be traversed: During rendering, rays reaching invalid leaves will be stopped. Data will be loaded, and rendering restarted.

5 Our rendering algorithm

5.1 Ray-casting our structure

Our rendering consists of marching the data in the structure along the view rays while cumulating color and opacity. Hierarchically, rays have first to traverse the N^3-tree, then the M^3-bricks if any (see Figure 2,Right). The most efficient method to traverse the tree along a ray should be recursive DDA (*i.e.,* generalized Bresenham) through the N^3 tree nodes. This algorithm relies on a stack which implements on GPU as indexed memory [SM4], but this is very inefficient on current GPUs. Instead, we locate the leaf node corresponding to the next ray segment by a direct descent from the tree root similarly to the *kd-restart* algorithm [HSHH07]. A brick is traversed by a regular sampling along the the ray. The sampling rate and the relative MIP-map level are set depending on the viewpoint. Recall each brick has its own MIP-map pyramid.
The optical operation per voxel is the same as with any volume tracer: Light and opacity are accumulated, and we consider a pre-integrated transfer function as in [EKE01] for quality reconstruction. A phase function or a pseudo-Phong lighting (using the density gradient as normal) can easily be accounted for.

As usually done for GPU ray-tracers, the process is initialized by rendering a large single primitive on screen (called *proxy surface*) with the ray-caster implemented in a pixel shader. We use a simple quad covering the screen.

5.2 Interrupting and resuming the rendering

In our case, after a camera move all the necessary bricks to render the frame may not be present in GPU memory when launching the ray-casting. We do not want to produce and fetch all the volume data present in the view frustum since we expect the rendering of foreground bricks to occlude background bricks before we ever have to generate and store them. Instead, we stop all rays reaching a missing tile. This is of course done on a per-pixel basis. The consequence is that we may need to perform several passes to complete the ray-casting of the volume for a given frame, each pass resuming each ray where it stopped after the missing tiles get produced and stored on GPU. Thus each rendering pass has to provide a list of missing tiles to query to the producer (see Section 5.3) and the necessary information to resume the ray-casting.

To process efficiently these multiple passes of rendering, we treat only the rays which need to be resumed when rasterizing the (large) proxy surface. This is done by adding a *mask buffer* and a *ray-state buffer* (see scheme on Figure 5):

- The *mask buffer* flags the pixels for which rendering must be resumed. These are the pixels for which data was missing. It is encoded as a Z for efficiency (`early-Z` test will prevent for the evaluation of useless fragments).
- The *ray-state buffer* stores the state of stopped rays (due to tile miss) and is generated by the previous rendering pass. This is done by using a dedicated `render target` in addition to the regular RGBA and Z outputs. The ray-state simply consists of the distance to eye of the interrupted sample.

During the next rendering pass (*i.e.,* when rasterizing the quad proxy surface), these buffers are used as input texture. Note that they always remain on GPU side. Only rays selected by the mask are thus launched[4] (*i.e.,* the not already opaque for which data was missing), starting at the provided location.

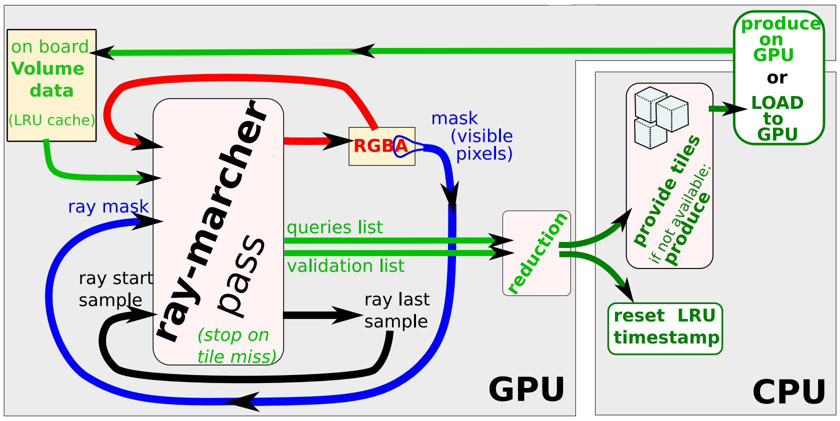


Figure 5: *Interaction between rendering and production passes during the processing of one frame(see also Figure 4 for the data structure and Figure 6 for the overall algorithm). Mask flow is in blue, tile flow is in green. The ray-marcher marches a set of selected rays and stops on tile miss. Visible pixels are kept in a mask to point the rays to be resumed once the data will be completed. Tile numbers are registered in a compact* tile query buffer *provided to the CPU. Tiles already stored in the GPU brick cache or in the optional CPU cache are immediately available. For other tiles, the CPU produces and uploads missing tiles.*

5.3 Returning tile information from a rendering pass

Getting the list of missing tiles to query
A ray stops if it traversed the whole scene, if it gets opaque, or if during the tree traversal the shader has reached a tile with "invalid" flag – *i.e.,* a leaf not at the correct LOD for which the brick is not available. It is easy to produce an extra output buffer storing these tiles numbers, but it would be costly to transfer such a large buffer back to the CPU. Moreover, it is very redundant as large areas will tend to reach the same missing tile. Instead, we reduce it to a small *tile queries buffer* (typically, 32×32) in which each pixel summarize the information of a *pixel block* (typically, 16×16). This reduction is done with an extra pass which rasterizes the small buffer. Its shader scans the tiles numbers of the small block corresponding to the current buffer pixel and stores the first four different tile numbers. Of course some information might be discarded, but this will simply require more passes since rays will reach an invalid leaf again.

Providing the queried tiles
The small *tile queries buffer* is uploaded in CPU memory and parsed. Redundant queries are ignored. 4 cases of query can occur: constant tiles identified by to the oracle are treated immediately. Similarly for tiles which bricks are already available in the GPU brick cache or in the optional CPU cache. Subdivision requests are treated immediately. However, the production requests for child tiles are not generated immediately since it is not yet proven that a ray will reach every child. The only exception are constant bricks, since their update and required memory is very low. Remaining queries correspond to bricks which must be generated by the producer.

[4]The `early-Z` test guarantees that only these fragment shaders are evaluated.

Updating tiles timestamp
We need to know which tiles were traversed during rendering in order to reset their time stamp in the LRU cache. (As long as a tile remains on GPU, the CPU side has no information about its use). Here again we must transfer tile information to CPU, but this one is not critical and can be done lazily. We take advantage of the holes in the *tile queries buffer*: if there are less than four missing tiles in a pixel block, we store validated tiles in the free entries instead, with a flag to identify them. In the worst case, that is screen regions with no tile miss and no tile LOD change during several frames, this will reset the time stamp of at least 4 tiles per pixel block and per frame.

5.4 Practical issues

Mixing volumes and meshes
Our representation can handle a complete scene. Still, in CG it is often required to combine several representations (meshes, billboards, particles...), possibly organized in a scene graph. In such case, we embed the volume rendering inside the proxy surface of an object (*e.g.,* see our hypertexture examples below). This is done by using this surface as the proxy primitive on which to evaluate the ray shader. Note that this also provides an optimization of the volume rendering since voxels and tiles out of the shape are not even considered.

Guaranteeing the frame rate
The number of tiles to update depends on the camera motion, its velocity, the proximity of the surface, its sparsity, etc. A given frame might require too many updates and cause a sudden irregularity in the interactive frame rate. If this is not acceptable for a given application, we can set a time budget for a frame and postpone all tile productions after the delay is passed. The highest available LOD will be displayed in place of the required data. For this, we need to slightly modify the last pass so as to keep track of the last available data when descending in the tree hierarchy.

6 Implementation of Producers and Oracles

6.1 Producer

The source data describing the scene can be stored on the disk or in memory, possibly compressed, or described procedurally, or a combination of those (*i.e.,* data amplification). The *producer* is in charge of providing bricks of data on demand, for a given region of space at a given resolution. If this region is constant, it returns the constant value instead. It may also determine that a lower resolution is sufficient for this brick. These tile status can be precomputed (for explicit data), evaluated dynamically trough analysis of the produced brick, or known a priori by the oracle (*e.g.,* bounding box, procedural data). Note that despite the oracle, it is often interesting to make the producer analyze the resulting brick before uploading it to GPU: if empty, constant or low-frequency this can save both storage and ray-tracing performances.

For explicit volume data the producer is on the CPU and its task is simply to access the data from disk or RAM and upload it on GPU. For procedural data, it is sometimes possible to implement the producer totally on GPU (see examples in Section 6.3). The first case costs in CPU time and in bandwidth due to transfers from the CPU to the

```
//--- init frame ---------------
foreach tree leaf in frustum
  if (LOD(tile)!=MAX(distLOD(tile),contentLOD(tile)))
     if too_subdivided(tile) setLeaf(parent at right LOD)
     invalid(tile)

forever // --- interleaved render/query/production passes
  // --- render ----------------
  bind mask; bind ray-state; draw quadProxy
    shader: (for pixels in mask)
       march tree from dist=ray-state(pixel)
         if (tile=invalid) output RGBA,dist,tile; exit fragment
         if (tile=const) cumulates RGBA(length)
         if (leaf) march brick
                     cumulates RGBA
                     if (opaque) ouput RGBA; exit fragment
  generates mask from tileQueriesBuffer (set if non empty query)
  draw tileQueriesBuffer
    shader: reduce missingTileBuffer, insert tiles to confirm
  // --- query and production ------------
  get and parse tileQueriesBuffer
    if "empty" end of frame
    if "to validate"  resetTimeStamp(tile)
    if "require brick(tile)"
      if (oracle(tile)="const") update node
      if isInGPUcache(tile) restore links
      if isInCPUcache(tile) upload brick; update tree (*)
      else brick=producer(tile); upload brick; update tree (*)
    if "to subdivide"
       subdivide(tile); update tree
       foreach subtile
          if (oracle(subtile)="const") update node
          else invalid(subtile)
```

Figure 6: *The overall algorithm. Uploading uses the oldest pool slot (*i.e., *LRU mechanism).* **(*):** *if the producer is on GPU there is no upload and no CPU cache.*

GPU, the second case costs in GPU time.
Procedural descriptions can be managed either by the producer (creation of a brick) or they can be used directly on the fly during the ray-casting (*e.g.,* amplification of low resolution data). The second solution saves storage but it is more costly if the voxels of the brick are evaluated more than once on average.

6.2 Oracle

The *oracle* is in charge of delivering information allowing to skip treatments for a tile (*i.e.,* subdivision or brick production) using some a priori knowledge: for a given region of space, it may tell conservatively if this region is invisible, or constant (*e.g.,* empty space or in the cluster core).

A priori knowledge is available for most scenes of interest for Computer Graphics. *E.g.,* at least the bounding box of objects can be known, or an optimization structure might be precalculated (*e.g.,* [CS94]). In other cases, where volumes are used to locally

a

Bone			
depth	resol	ms	fps
5	512	99	10
7	2048	70	14
9	8192	41	24

b

Sierpinski			
depth	resol	ms	fps
2	729	10	95
4	(hole)	18	56
4	6561	24	41
10	4.8M	30	33

c

Bunny		**pseudo-mesh**		**ramp**		**transp**		**hypertexture**		**perlin**	
depth	resol	ms	fps	ms	fps	ms	fps	ms	fps	ms	fps
5	512	19	52	38	26	54	19	26	38	31	32
7	1024	30	33	77	13	149	7	41	24	52	19
9	8192	83	12					91	11	59	17
10	16384							139	7		

d

Exploration example	**render** (ms)	**update** (ms) prod	#nb	#passes	upload	tree	**tot**	**fps**
Bone	116	1.4	27	3.6	7.5	38	60	5.4
Sierpinski	25	0	0	0	0	0	0	40
Hypertex	218	277	83	4	0	37	329	1.8

e

extra costs	bone	Sierpinski	pseudo mesh	ramp	hypertexture	Perlin
prod 1 tile	0.3	0	0	0	3.3	0
shading cost	16	38	19		28	

Figure 7: *Performances with our 3 examples scenes: bone data, Sierpinski sponge, hypertextured bunny.* **a,b,c:** *ray-casting cost at 3 typical distances (no animation, no shading). See Figure 8 for the corresponding images. 'Hole' in b is the variant shown in the teaser. The grid marching represent 66% to 75% of this cost. The shading cost in given in* **e,bottom:**. *Most of it is due to the on-the fly calculation of normals from density gradient.* **d:** *We show the decomposition of the rendering time and the update time during exploration. Note that these explorations go deep in the matter, thus challenging our method. Timings in a,b,c give the best fps one can expect at a given depth. The production cost of one single brick is given in* **e,top**. *For update performance (measured on the videos), the camera velocity was set to 4 closest voxel length per frame (*i.e., *we adapted it to the resolution of the closet tile, with time relaxation). Performances for other velocities scale linearly.*

enrich surfaces (as for hypertextures, relief maps, bidirectional textures, volumetric textures), it is known by construction that the details are embedded between an interior and an exterior surfaces.

6.3 Applications

We tested a simple explicit volume producer, a procedural producer, and an hypertexture producer.

• The explicit volume producer is trivial: raw volume data already exists and bricks are precomputed by regular subdivision of the data. Precomputation can also analyze these bricks to mark constant ones. The CPU producer simply upload them on demand to the GPU. We also tested amplification of the data with procedural noise to increase the resolution of details. If no precomputation is possible (*e.g.,* output from another program) and no oracle is available, it might still be interesting to analyze bricks on CPU before transferring them to GPU to replace them on the fly by constant tiles when it applies.

• We tested two types of procedural producers: density defined by a noise function (see next item), and fractal geometry illustrated by a Sierpinski sponge. For the later we simply always refer to the same unique brick at every location and scale during the exploration.

• For hypertextures [PH89], opacity is 0 outside of the outer surface and 1 inside of the inner surface (thus the oracle can easily avoid evaluating the procedural function in these regions). In the interface layer between both surfaces, a distance field $d(X)$ allows to define a gradient opacity from 0 to 1. The noise is described as a perturbation of this gradient. In our example we chose opacity=$f(d(X)+k.P(X))$ where $P(X)$ is Perlin noise [Per85], $f()$ is a sigmoid function and k is a gain factor. Since $d()$ corresponds to the distance to a surface its evaluation on the fly is complicated, so we charge the producer to build low-resolution bricks of the distance field. This is detailed in Appendix A. We chose to evaluate $P()$ on the fly since we expect the interior voxels to be invisible due to opacity occlusion. For comparison, we also tested generating high resolution bricks through a GPU producer of hypertexture.

7 Results and discussions

All our benches were done on a Core2 bi-core E6600 at 2.4 GHz and a 8800 GTS 512 graphics card (G92 GPU) with 512 MB. All images are rendered at resolution 512×512. See also the accompanying video.

Example 1: *Explicit volume (trabecular bone).*
We used a 1024^3 scanner data of a trabecular bone. We pre-subdivided the data into 16^3 bricks at every LODs. We pre-computed the MIP-map and we analyzed the data to mark constant blocks. In our tests, the data could fit the CPU memory (testing disk-to-GPU streaming is out the scope of this paper). We copied this volume 8 times in each direction in order to simulate a 8192^3 resolution. In our data structure, we used $N = 2$ and $M = 16$. See performances on Figure 7,a,d.

Example 2: *Procedural volume by instantiation (Sierpinski).*
We used one unique brick of size 81^3 which is instantiated at all non-empty tiles, so that there is no producer. We naturally chose $N = 3$ so that we can also rely on one unique node which is instantiated for all non-empty children (we benched with and without). The resolution is potentially infinite, but in practice the floating point precision of coordinates limits the zoom to 2^{19} so the maximum virtual resolution is $8.4M^3$. See performances on Figure 7,b,d.

Example 3: *Hyper textures and amplification of a mesh.*
For this example we use a volume clipped inside a bunny mesh: the ray-marching operates only between the first and last ray-mesh intersection. We implemented the distance field producer to generate a ramp in the vicinity of the surface, the hypertexture based on this distance field (either as a brick producer or as on the fly amplification, using 20 octaves of Perlin noise), and some other test configurations. In these examples we used $N = 2$ and $M = 16$. See performances on Figure 7,c,d.

Example 4: *Cumulus cloud.*
Our method was used to encode the cloud details in [anonymous] (a paper dealing with multiple scattering in clouds). The raw shape was described with a mesh which we amplified with hypertextures much like the bunny example above. The distance field was about 13% of the cloud size. We used $N = 2$, $M = 32$, and 5 octaves of Perlin noise. The virtual resolution was 2048^3.

Memory usage:
In most examples the node pool was small: 4 MB, corresponding to 64^3 entries. Using 16^3 bricks, this can index a 1024^3 volume pool. Our implementation incurs a memory overhead (using 16 bytes per entry which could be easily compressed), but it is not critical for this pool. The addressable amount of bricks is larger than what the GPU memory can contains. The brick pool used 430 MB, allowing for 42^3 bricks.

Our method handles both GPU producer for procedural noise and for on the fly computation of the noise during the ray-marching. Generally, which one to choose is a tradeoff between computational efficiency and memory efficiency. This essentially depends on the application. Moreover, it is interesting to use the on-the-fly methods to let the artist fine tune noise parameters, and then to later rely on a producer for the final visualization. Note also that if brick voxels are used less that 1 time on average (*e.g.,* sharp shapes, fast camera motion, animated noise), on-the-fly noise evaluation becomes the most efficient in terms of computations.

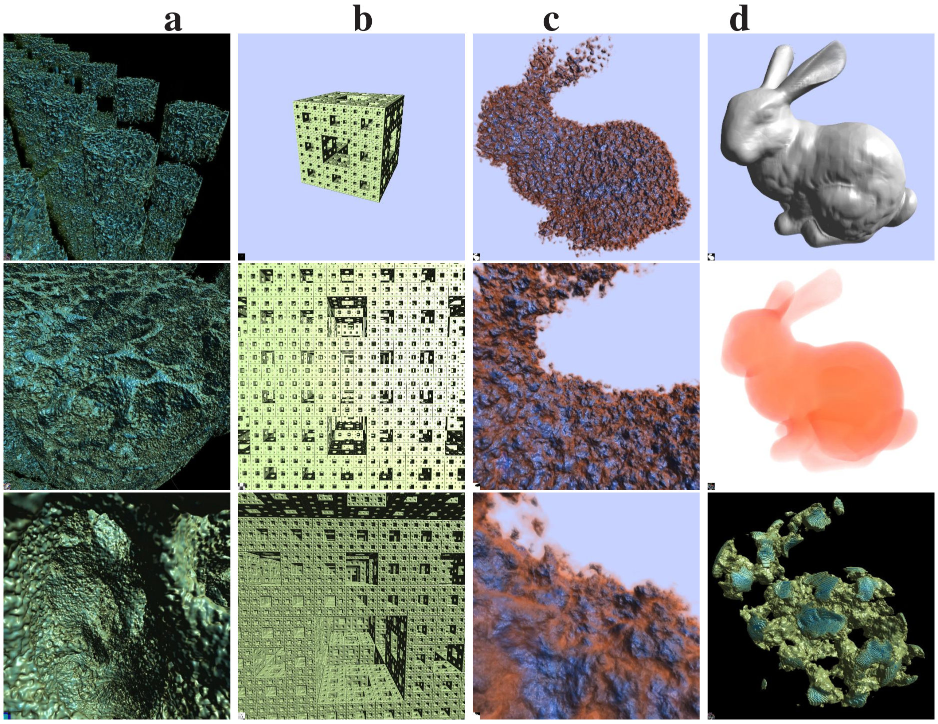


Figure 8: *Some result images corresponding to timings. (a): Trabecular bone data. (b): Sierpinski procedural sponge. (c): Hypertextured bunny. (d): Bunny with pseudo-surface, blurry silhouette, solid noise (see also the teaser).*

Another similar tradeoff is on-the-fly transfer function versus more complex voxel data (*e.g.,* RGBA instead of density). On interior views of the bone example, applying the transfer function during brick production instead of ray-marching would save 8*ms* to be compared with a 16 time increase in memory cost. Yet another similar tradeoff concerns normals calculation. In all our examples, we used on-the-fly gradient computation.

8 Conclusion and future work

We have presented a method allowing for interactive rendering of large and very detailed volumes. It is based on a fully dynamic workflow (data structures and algorithms) producing, loading and rendering on the fly only what is necessary for the current viewpoint. We presented a complete scheme to march the complex data structure totally on GPU.

For future work, we plan to investigate thinner performance tuning, for instance by pre-producing bricks very likely to be necessary soon, *e.g.,* the one close to the LOD threshold. Priority management of tiles to subdivide, and dynamic load-adaptation of global voxel resolution are other potential paths. We also want to manage instancing within the data, similarly to what is done in 2D by [LN03]. Our next goal is to account for shadows and animated data in the same spirit.

A Computing distance fields on GPU

We propose a method to calculate on GPU the approximate distance field to a surface in a volumes. The surface mesh is kept in GPU memory in a `VBO`. We want to generate the brick corresponding to the field in a given tile. The principle is to evaluate for each voxel its distance to the mesh in the 6 axis directions. The distance to the mesh is then approximated as the distance to the plane defined by the closest intersection along the 3 axis (see Figure 9).

In practice, for each slice of the brick we project the mesh to evaluate the distance along the `z` axis, keeping $abs(z)$. We simulate slices in `xz` and `yz` directions by rotating the mesh instead (so that we slice the volume only once), and we store the 3 corresponding distance in `R,G,B`. Then we calculate the distance estimate from these 3 values as described above.

This approximation is acceptable in our case since the volume layer thickness under the surface is small. (The lowest frequency of the proxy surface is supposed to be larger than the layer thickness). It yields very good performances as compared to other distance field evaluation methods.

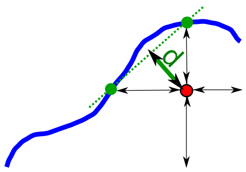


Figure 9

References

[AH05] ASIRVATHAM A., HOPPE H.: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. 2005, ch. Terrain rendering using GPU-based geometry clipmaps, pp. 109–122.

[BD02] BENSON D., DAVIS J.: Octree textures. In *SIGGRAPH'02* (2002), pp. 785–790.

[BD06a] BABOUD L., DÉCORET X.: Realistic water volumes in real-time. In *Eurographics Workshop on Natural Phenomena* (2006), Eurographics.

[BD06b] BABOUD L., DÉCORET X.: Rendering geometry with relief textures. In *Graphics Interface '06* (2006).

[BTG03] BARGE B. L., TESSENDORF J., GADDIPATI V.: Tetrad volume and particle rendering in X2. In *SIGGRAPH 2003 Sketches & Applications* (2003). `http://portal.acm.org/ft_gateway.cfm?id=965491`.

[CB04] CHRISTENSEN P. H., BATALI D.: An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)* (2004), pp. 133–142.

[CS94] COHEN D., SHEFFER Z.: Proximity clouds - an acceleration technique for 3D grid traversal. *Vis. Comput. 11*, 1 (1994), 27–38.

[DGPR02] DEBRY D., GIBBS J., PETTY D. D., ROBINS N.: Painting and rendering textures on unparameterized models. In *SIGGRAPH'02* (2002), pp. 763–768.

[DN04] DECAUDIN P., NEYRET F.: Rendering forest scenes in real-time. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)* (june 2004), pp. 93–102.

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (HWWS)* (2001), pp. 9–16.

[Hav00] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. `http://www.cgg.cvut.cz/~havran/phdthesis.html`.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree GPU raytracing. In *ACM Siggraph symposium on Interactive 3D graphics and games (I3D)* (2007), pp. 167–174.

[Kap02] KAPLER A.: Evolution of a vfx voxel tool. In *SIGGRAPH 2002 Sketches and Applications* (2002), p. 179. `http://portal.acm.org/ft_gateway.cfm?id=1242192`.

[Kap03] KAPLER A.: Avalanche! snowy FX for XXX. In *SIGGRAPH 2003 Sketches & Applications* (2003). `http://portal.acm.org/ft_gateway.cfm?id=965492`.

[KE02] KRAUS M., ERTL T.: Adaptive texture maps. In *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS)* (2002), pp. 7–15.

[KH05] KRALL J., HARRINGTON C.: Modeling and rendering of clouds on "stealth". In *SIGGRAPH 2005 Sketches* (2005), p. 85. `http://portal.acm.org/ft_gateway.cfm?id=1187214`.

[Kis98] KISACIKOGLU G.: The making of black-hole and nebula clouds for the motion picture `Sphere´ with volumetric rendering and the f-rep of solids. In *SIGGRAPH 98 Conference abstracts and applications sketches* (1998), p. 289. `http://portal.acm.org/ft_gateway.cfm?id=282285`.

[KK89] KAJIYA J. T., KAY T. L.: Rendering fur with three dimensional textures. In *SIGGRAPH'89* (1989), pp. 271–280.

[LD07] LEFEBVRE S., DACHSBACHER C.: Tiletrees. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)* (2007).

[LDN04] LEFEBVRE S., DARBON J., NEYRET F.: *Unified Texture Management for Arbitrary Meshes*. Tech. Rep. RR5210-, INRIA, May 2004. `http://www-evasion.imag.fr/Publications/2004/LDN04`.

[LH04] LOSASSO F., HOPPE H.: Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH'04* (2004), pp. 769–776.

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *SIGGRAPH '06* (2006), pp. 579–588.

[LHN05a] LEFEBVRE S., HORNUS S., NEYRET F.: *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*. 2005, ch. Octree Textures on the GPU, pp. 595–613.

[LHN05b] LEFEBVRE S., HORNUS S., NEYRET F.: Texture sprites: Texture elements splatted on surfaces. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)* (April 2005).

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94* (1994), pp. 451–458.

[LN03] LEFEBVRE S., NEYRET F.: Pattern based procedural textures. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)* (2003), ACM, ACM Press.

[LSK*06] LEFOHN A. E., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph. 25*, 1 (2006), 60–99.

[Ney98] NEYRET F.: Modeling animating and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (Jan.–Mar. 1998), 55–70. ISSN 1077-2626.

[PBFJ05] PORUMBESCU S. D., BUDGE B., FENG L., JOY K. I.: Shell maps. In *SIGGRAPH'05* (2005), pp. 626–633.

[Per85] PERLIN K.: An image synthesizer. In *SIGGRAPH'85* (1985), pp. 287–296.

[PH89] PERLIN K., HOFFERT E. M.: Hypertexture. In *SIGGRAPH'89* (1989), pp. 253–262.

[Sch05] SCHARSACH H.: Advanced GPU raycasting. In *Central European Seminar on Computer Graphics 2005* (2005), pp. 69–76.

[SM4] SM4: Shader model 4 opengl specification. `http://developer.download.nvidia.com/opengl/specs/GL_EXT_gpu_shader4.txt`.

[TZL*02] TONG X., ZHANG J., LIU L., WANG X., GUO B., SHUM H.-Y.: Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM Transactions on Graphics 21*, 3 (2002), 665–672. (Proceedings of ACM SIGGRAPH 2002).

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399