GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering

Cyril Crassin, Fabrice Neyret
LJK / INRIA / Grenoble Universities / CNRS

Sylvain Lefebvre
INRIA Sophia-Antipolis

Elmar Eisemann
MPI Informatik / Saarland University

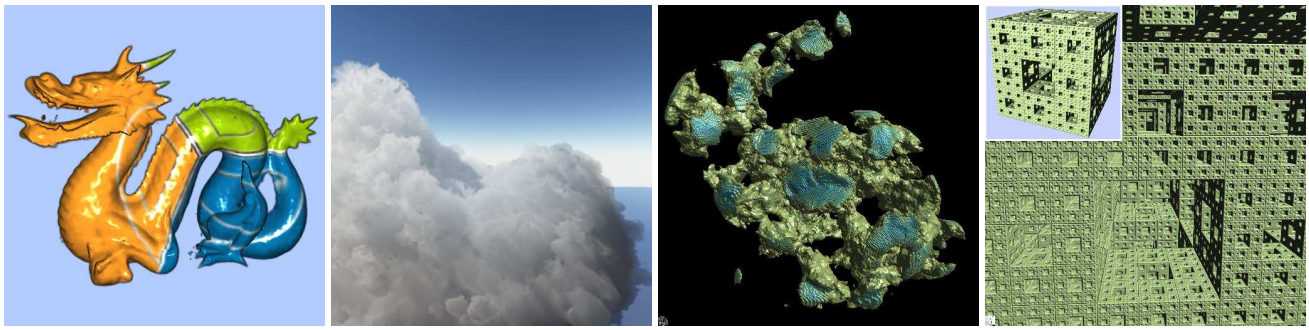

Figure 1: *Images show volume data that consist of billions of voxels rendered with our dynamic sparse octree approach. Our algorithm achieves real-time to interactive rates on volumes exceeding the GPU memory capacities by far, tanks to an efficient streaming based on a ray-casting solution. Basically, the volume is only used at the resolution that is needed to produce the final image. Besides the gain in memory and speed, our rendering is inherently anti-aliased.*

Abstract

We propose a new approach to efficiently render large volumetric data sets. The system achieves interactive to real-time rendering performance for several billion voxels.

Our solution is based on an adaptive data representation depending on the current view and occlusion information, coupled to an efficient ray-casting rendering algorithm. One key element of our method is to guide data production and streaming directly based on information extracted during rendering.

Our data structure exploits the fact that in CG scenes, details are often concentrated on the interface between free space and clusters of density and shows that volumetric models might become a valuable alternative as a rendering primitive for real-time applications. In this spirit, we allow a quality/performance trade-off and exploit temporal coherence. We also introduce a mipmapping-like process that allows for an increased display rate and better quality through high quality filtering. To further enrich the data set, we create additional details through a variety of procedural methods.

We demonstrate our approach in several scenarios, like the exploration of a 3D scan (8192^3 resolution), of hypertextured meshes (16384^3 virtual resolution), or of a fractal (theoretically infinite resolution). All examples are rendered on current generation hardware at 20-90 fps and respect the limited GPU memory budget.

This is the author's version of the paper. The ultimate version has been published in the I3D 2009 conference proceedings.

1 Introduction

Volume data has often been used in the context of scientific data visualization, but it is also part of many special effects. Companies such as *Digital domain, Cinesite*, or *Rhythm 'n Hues* now massively rely on so-called *voxel engines* [Kis98, BTG03, Kap03, KH05] to render very complex scenes.

Examples can be found in many recent movie productions (*e.g., XXX, Lord of the Rings, The Day After Tomorrow, Pirates of the Caribbean, The Mummy 3*). Clouds, smoke, foam, and even non-fuzzy but extremely detailed geometric data (*e.g.,* boats in *Pirates of the Caribbean*) are all represented with voxels and rendered via volume rendering. The scene size and resolution is so large that voxels often do not even fit in the computer's memory. In addition to storage, the rendering of such data is also extremely costly, even for previsualization.

The significant advantage of voxels is the richness of this representation and the very regular structure which makes it easy to manipulate. In particular, filtering operations are well-defined, making it a good candidate to address aliasing issues that are hard to deal with for triangulated models.

This is one of the reasons voxel data is often used to represent pseudo-surfaces, which is an interface that resembles a surface at a certain distance, but appears complex (non-heightfield, non-connected, or non-opaque) at close view. An example is the foliage of a tree that can be well approximated with volumetric data [RMMD04], but this observation holds for complex surfaces in general. Thus volumetric rendering is also a graceful way of dealing with the level of detail problem.

In this paper, we show that the current hardware generation is ready to achieve high-quality massive volume renderings at interactive to real-time rates. Benefits such as filtering, occlusion culling, and procedural data creation, as well as level-of-detail mechanisms are integrated in an efficient GPU voxel engine. This enables us to obtain some of the visual quality that was previously reserved for movie productions and enables the technique to be used to previsualize special effects.

There are two major issues before making detailed rendering of massive volumes possible: overcome the memory limitations (and propose related update schemes) and the usually costly rendering.

Volume data can require *large amounts of memory*, thus *limiting the scene's extent and resolution of details*. The fact that the scene can no longer be held entirely in memory implies the need for *intelligent*

data streaming from larger, but slower memory units. This point is particularly crucial for real-time applications because the memory restrictions on the GPU are more severe than for CPU software productions. Even if we were able to iteratively fill the GPU memory in a brute-force manner, the transfer of 512MB each frame (which is the standard memory size on current GPUs) already prevents real-time performance.

The second challenge is the actual *rendering of volumes*. Displaying data with pure fixed-step ray marching can lead to alias artifacts and is costly due to the large amount of voxels that needs to be visited, shaded, and blended.

Fortunately, most of the time, only parts of the volume are needed at the full resolution. Others are invisible or distant enough to be approximated without quality loss. Occlusion can also cover parts that in consequence can be omitted. This can happen even if the scene has no opaque entity. Transparent materials can quickly accumulate to denser, thus opaque, elements which block the view.

This paper aims at making voxels an alternative rendering primitive for real-time applications. Our framework is highly inspired by voxel engine tools used in special effect productions: it lifts features known to be time and memory consuming (even in the scope of offline production) to interactive and real-time rates.

2 Previous Work

Much work has focused on volume visualization, too much to be covered in this paper. For a recent overview of real-time volume rendering techniques, we refer the reader to [EHK*06]. Also, many approaches focus on overcoming discretized representations to recover a smooth signal. This also includes high-quality normal reconstruction (e.g., [MMMY97]). Our focus lies more on data management and efficient artifact-free rendering. The high resolution of our input also hides, to some extent, aliasing issues.

Full voxel grids have been used in many applications to benefit from the visual complexity allowed by volumes for, e.g., fur [KK89], vegetation [DN04], or pseudo-surfaces [Ney98]. Because of the very detailed and realistic appearance that arises from the use of voxels, much effort was spent on accelerating volume rendering. Recently, many GPU-based approaches were published. Earlier solutions relied on volume slicing [LL94], whereas newer approaches perform a ray marching in the fragment shader.

Many efficient methods assume that the entire volume is present in the graphics card's memory, thus highly limiting the possible detail level. Therefore, in the context of real-time rendering, voxels remained mostly a valuable representation for distant models (e.g., [MN00, GM05, DN09]) because resolution can be kept low, counteracting the usual memory cost.

Most recent methods rely on some ray marching procedure where opaque models are of particular interest as they allow for an early ray termination. Height fields share this concept. It also aims at improving performance via sophisticated preprocessing methods [BD06] and intelligent ray marching [PO07], but are mostly limited to this restricted type of data.

It is true though, that a full volume is not always needed. For computer graphics scenes, it often suffices to have detail located mostly in a layer at the interface between empty- and filled space. The observation has been used in rendering in form of specialized representations such as shell maps [PBFJ05], relief textures [OBM00], bidirectional textures [TZL*02], and hypertextures [PH89]. In all cases, volume data is represented in a limited interface attached to an object's surface, or to the space around, as is the case for some recent GPU structures [LHN05b, LH06, LD07]. Nonetheless, the storage and rendering of these previous methods are only efficient if the volume layer remains small on the screen. Instead, we will deal with general volumes.

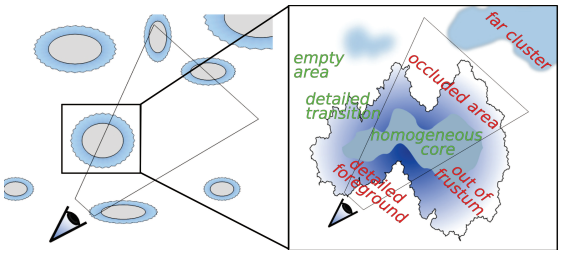

Figure 2: *In CG scenes, details tend to be concentrated at interfaces between dense clusters and empty space.*

General volumetric data sets also often consist of dense clusters in free space (compare Figure 2) so-called *sparse-twice*. One can benefit from clustered data, e.g., by compaction [KE02] or fast traversal of empty space, avoidance of occluded regions [LMK03], or by stopping rays when a certain opacity level is reached [Sch05]. We also want to make use of regions of constant density for acceleration and compaction purposes.

Although inspired by the aforementioned work, this paper focuses on out-of-core voxel rendering to achieve the aim of richness of the representation, in conditions where the memory of the GPU is typically orders of magnitude smaller than the data, which can be very detailed. Much research has focused on the topic of massive volume rendering. Boada et al. [BNS01] analyze the volume data by computing a mipmap-like structure based on an octree. They then choose a cut through the tree and use the leaves' mipmap data during rendering. LaMar et al. [LHJ99] keep all these different resolution levels in memory and choose the blocks according to the distance from the observer during rendering. Our goal is a combination of both: view-dependent rendering for massive volumes. This is close to [GS04], where data is compressed, used in a view-dependent manner, and empty space is skipped efficiently. Although relatively efficient, the approach involves much CPU work, does not achieve temporal continuity, nor does it include occlusion tests to cull hidden parts of the volume.

Volume decompositions into blocks have been used in many other approaches, e.g., [HQK05], but without filtering, the block structure remains visible and aliasing artifacts can occur. Filtering was discussed in [VSE06] where two approaches are presented. One based on [LHN05a], but leading to a more difficult filtering scheme, and one with higher precision based on [LKS*06], but that tends to overrefine some parts of the scene. The structure is view-independent though, whereas our approach adapts resolution continuously. This makes discontinuities disappear and temporal transitions to a different subdivision level are smooth.

Our structure shares similarities with brick maps [CB04], but is dynamic, and, as a consequence, its content is view-dependent (through LOD (level-of-detail) and visibility). We thus combine an adaptive version of the structure from [LHN05a] with regular constant-sized memory blocks [LKS*06], which allow for very efficient hardware-based filtering.

The most related work has very recently been published by Gobbetti et al. [GMAG08] and we will take a closer look at their solution in the next section.

Preliminaries

Our approach was developed in parallel to the work of Gobbetti et al. [GMAG08]. While it shares similarities with this work, our approach provides better quality / performance and offers new possi-

bilities such as procedural content and level-of-detail management thus pushing forward the idea of making volumes a rendering primitive. To better understand our contributions and make this paper self contained, we will first give an overview of what our approach shares with Gobbetti et al.'s one.

Let's start with a naive consideration. If the volume is small, GPUs allow for an efficient rendering by simply stepping through the data set stored in a 3D texture and accumulating the voxels' colors. In this way, one can benefit from several hardware related advantages, like direct 3D addressing, tri-linear interpolation, and a 3D coherent texture cache mechanism. For larger volumes, even if the GPU had enough memory, there are two problems: first, the algorithm would be slow due to many steps that need to be taken in large data sets and, second, the whole dataset will not fit into the GPU memory.

One insight is that, for a given point of view, not the entire volume needs to be in memory. By organizing the data in a spatial subdivision structure, empty parts can be let unsubdivided and distant parts can be replaced by lower mipmap levels, leading to a lower resolution (a different level-of-detail, *LOD*), thus less GPU memory requirements. In addition, for a given point of view, hidden parts do not need to be loaded at all (see Figure 2). As Gobbetti et al., we chose an octree structure, which is convenient to represent and traverse on the GPU [LHN05b, LHN05a], and is well adapted to store regular data like voxels.

The octree is refined as to reflect the needed precision in the scene. Each tree node contains a pointer to a so-called *brick* or is indicated as constant/empty space. A *brick* is a small voxel grid of some predefined size M^3 (usually $M = 32$) that approximates the part of the original volume that corresponds to the octree's node. For example, the brick for a root node would be an M^3 voxel approximation of the entire data set. The data representation thus combines the memory efficiency of an adapted structure with small 3D texture units that can be efficiently evaluated through ray marching.

All bricks are grouped in a large 3D texture, the *brick pool*, stored on the GPU. This memory is limited and so its content needs to be chosen wisely and updated when the viewpoint changes. At all times, the algorithm makes sure that the bricks of all visible leaf nodes are in the pool. This enables an appropriate rendering of the entire volume from the current viewpoint. For other nodes, bricks might be missing, in which case pointers are invalidated.

When the viewpoint is moved, nodes in the tree are fused or collapsed based on the needed resolution and visibility. An update is triggered if data in the brick pool is missing. Each brick in the pool has a certain timestamp that is reset upon usage. If an octree node needs a subdivision and new bricks are transferred to the GPU, the algorithm will use the memory locations that were previously reserved for the oldest, thus unused, bricks (this concept is referred to as *LRU* - Last Recently Used).

To keep track of the current data organization and facilitate updates, the structure is mirrored on the CPU. Structure modifications can thus be done on the host with all its general computational capacities, and only changes need to be transferred to the GPU. This facilitates some of the operations, such as the LRU ordering on the CPU. It also allows for a brick cache in the main memory, which is important if the volume is so large that hard disk accesses are necessary.

3 Overview and Contributions

With respect to the work described in the previous section, we introduce several important improvements. In Section 4, we present our generalized data structure and discuss its condensed representation on the GPU. It reduces memory requirements, simplifies implementation, accelerates computations, as well as updates, and allows us to exploit the sparse-twice data structures often encountered in computer graphics. In Section 5, we explain a basic rendering strategy that ensures a correctly produced image. This section also details the basic processes involved to traverse the grid.

We also address aliasing, which appears because the value of a pixel should not be defined by accumulations along a ray, but by the integration along a cone defined by the eye and the pixel's extent. Computing this value with several rays (FSAA methods) increases computation time significantly. As for 2D texturing, a way to overcome this issue is to use mipmapping. Equivalently, a voxel hierarchy can be built by iteratively downsampling and averaging neighbors. Due to the filtering process, reading from a mipmap delivers the integrated value of a small volume. If the step size during the ray marching is chosen accordingly to the distance from the observer, it is possible to obtain a good approximation of the actual cone integral and accelerate computations. Mipmapping thus addresses aliasing to a large extent, smoothes the results and can increase rendering performance, but it increases memory consumption and makes tree updates more challenging.

We then discuss a more advanced solution that relies on an LRU mechanism in Section 6. Here, we provide a unified rendering/visibility/LOD framework that is efficient, eliminates much of the previous CPU interaction, and is easy to implement.

Finally, Section 7 will show several results and possibilities to amplify and increase the detail level of the original data procedurally. This enables higher rendering quality than previously possible.

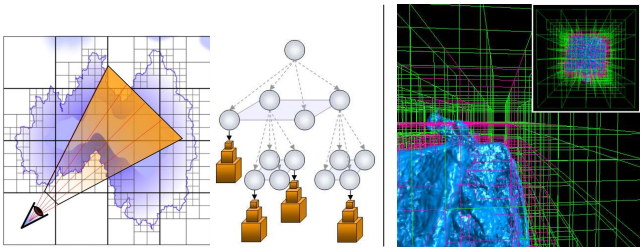

Figure 3: (Left:) *Our spatial structure combines a N^3 tree and mipmapped 3D texture tiles.* (Right:) *Hierarchical structure with constant nodes (green) or bricks (purple).*

4 Our Structure

As mentioned in Section2, an adaptive space subdivision is key to render large volumetric models. Our algorithm makes use of N^3-*trees* (similar to [LHN05a]). Each node in an N^3-tree can be subdivided into N^3-uniform children, hence its name. In the case of $N = 2$, this results in a standard octree, but using a different N can modify the algorithms behavior. A trade-off between memory efficiency (low N, deep tree) and traversal efficiency (large N, shallow tree) is easily possible and can be adapted to the repartition of the input data at each scale.

We further allow each node of the N^3-tree to store a brick pointer or a constant value. Storing a single value in the case of almost homogenous regions (empty or core) reduces memory requirements enormously and we can avoid ray marching by directly computing the voxel's contribution. We store brick pointers also in the interior nodes of the tree, as it will enable us to perform high-quality filtering via mipmaps. Our new update mechanisms, introduced in Section 6, will take this into account.

4.1 Implementation of the Structure

Figure 4 summarizes the data structure of our approach and might be helpful during the following discussion of the implementation.

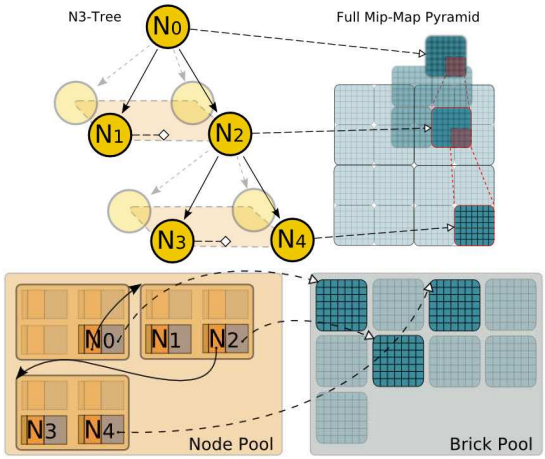

Figure 4: *Our N^3-tree+brick structure (illustrated as a 2^2 quad-tree for clarity).*

In our structure, only a single pointer is used per node, where Gobbetti et al. [GMAG08] needed eight such pointers for the child, as well as pointers to neighboring nodes. To make this possible, we keep all nodes in a 3D texture, which we refer to as the *node pool*. The texture is organized into blocks of N^3 nodes. Grouping the children of a node in one such block makes it possible to access all N^3 child nodes with a single pointer. In this way, not only coherence is largely improved during the traversal (3D texture caches on modern GPUs help drastically), but also memory requirements are largely reduced. Even though our structure does not exhibit neighbor pointers between adjacent nodes, we will show in Section 5.1 that an efficient traversal remains possible.

For each node, we reserve data to either store a constant value (homogenous volume), or a pointer towards a brick. With this information, we reduce the previous eight RGBA (32 bit/channel) texels [GMAG08] to two 32 bit values. This results in a 16 times storage improvement. It may seem less important because the N^3-tree memory occupation is much lower than for the brick pool, but the amount of data read during the traversal of the tree is critical for the GPU rendering performances and updates to the structure can be achieved with significantly less information transfer from the CPU.

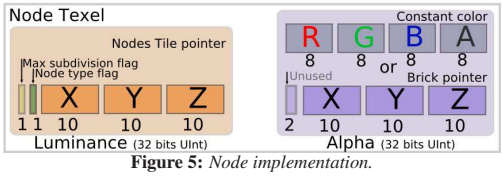

Figure 5: *Node implementation.*

The 64 bit node data is repartitioned as follows (see Figure 5):

- 30 bits - encode a pointer to the child nodes for non-leaves (zero meaning that there is no child);
- 1 bit - indicates whether the node is refined to a maximum, or whether the original volume still contains more data;
- 1 bit - stores whether the content is a constant RGBA8 value (*e.g.,* empty or core regions) or described by a brick;
- 30/32 bits - accordingly represent either:
  - A pointer to an M^3 brick for non-constant leaves (30 bits);
  - The average value at this location for homogenous leaves (4*8 bits).

In this paper, we also introduce a high-quality (quadrilinear) filtering based on mipmapping (Section 5.2). Without this anti-aliasing, the amount of data per node could be further reduced to only 32 bits because then only leaves need data pointers. Thus, child pointers, brick pointers, and constant values (stored as RGBA6) are all exclusive and can share the same 30 bits. The remaining two bits leave enough room for the rest of the necessary node data. Hence, volume data is not only needed at the highest resolution (leaves of the tree), but also for interior nodes. The performance cost of using 64 instead of 32 bits is minor because it still fits in a single texel of a luminance-alpha texture.

5 Basic Approach

In this section, we will discuss how to render and update the volume represented by the N^3-tree. It is useful to first consider the simpler case of standard rendering, where we assume that all necessary data is present. This will allow the reader to further familiarize with the data structure before we address the dynamic updates. In particular, our advanced approach in Section 6 combines this render algorithm and a visibility mechanism to trigger updates in a single unified method. Previous work relied on a separate step to analyze how to adapt the structure and which bricks should be loaded to next. This usually involved much CPU interaction, whereas most of our computations will be executed on the GPU.

5.1 Ray-Casting

Our rendering consists of marching the data in the structure along the view rays while accumulating color and opacity. Hierarchically, rays need to traverse the N^3-tree, and when reaching a leaf node, the M^3-bricks or homogenous region.

To initialize the rays, we use rasterization to draw some proxy geometry that delivers the origins and directions corresponding to the view rays to the fragment shader. One could use a screen-covering quad on the near plane, the bounding box of the volume data, or some approximate geometry that contains the non-empty areas of the volume. Such a proxy for the initial position can also be used to determine where the ray leaves the volume.

Both extremes can be computed in a single pre-rendering by activating an alpha blending set to a maximum blending and drawing the depth of the front-facing surfaces in the luminance, and one minus the depth of the back-facing ones in the alpha channel. Alternatively, we can tile two depth buffers in one texture and let the geometry shader move the back-facing triangles in the second tile while inverting their depth. In practice, to allow a fair comparison to previous work, all our tests were performed using the near plane to initialize the rays and the volumes bounding box to stop them.

One efficient method to traverse the tree along the initial rays would be a recursive DDA (*i.e.,* generalized Bresenham) through the N^3 tree nodes. This algorithm relies on a stack which can be implemented on GPU as dynamically indexed memory. This is inefficient on current GPUs (due to the lack of fast indexable memory within shader processors).

Instead, we use an iterative descent from the tree root similarly to the *kd-restart* algorithm [HSHH07]. It is particularly efficient because we highly benefit from the N^3-tree structure. We start with the origin of the rays and locate them in the N^3-tree (top-down). The descent stops when we reach a node with the appropriate level-of-detail (not necessarily a leaf). Such a node either represents a constant region of space, or contains a brick whose resolution is fine enough so that a voxel projects to at most one pixel. In case of a constant node, the value is simply integrated analytically along the distance the ray traversed in the node. In case of a brick, a standard ray-marching is applied until we leave the current node. The new position then serves as the origin for the next descent. One important observation is that our traversal does not need the structure to indicate correct level-of-details (as done previously [GMAG08]). This is determined in the shader. As we will see later, this is a key

feature to minimize update operations.

The descent is fast because the coordinates of a point can be used directly to locate it in the N^3-tree. Let $x \in [0,1]^3$ be the point's local coordinates in the volume's bounding box and c be the pointer to the children of the root node. The offset to the child node containing x is simply $int(x*N)$, the integer part of the multiplication between x and N. We fetch the child at $c + int(x*N)$ and continue with the descent by updating x with $x*N - int(x*N)$. Even though a new descent is needed every time a node is left, mostly the same pointers in the structure will be followed, thus the hardware texture cache is very well prepared.

This ray-casting is performed using one big fragment shader for both top-down traversal and brick sampling. It proved more efficient than making these two steps separate passes, probably due to local data storage and texture cache.

5.2 High Quality Filtering

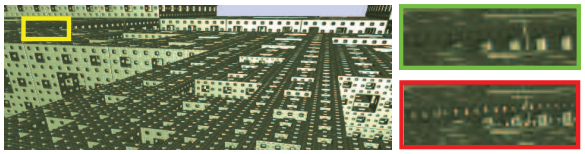

Figure 6: *Our method (top) does not show the noise of standard tracing (bottom).*

The rendering of the bricks should actually be performed differently than what we previously explained because we want to perform a better filtering of the values. Hence, we adapt the sampling rate and the relative mipmap level during the ray marching depending on the viewpoint.

As mentioned before, our traversal stops when it reaches the appropriate node. The idea behind it was twofold. On the one hand, it allows to increase rendering speed, on the other hand, it is a good approximation of a cone instead of ray tracing. For smooth transitions between different tree configurations and a better fit to the actual cone size, we need access to other mipmap levels of the bricks. Fortunately, these correspond to the bricks in nodes we encounter during the tree descent (see Figure 4).

One might think that many mipmap levels might be necessary to perform this internal filtering, but a brick only describes a small extent of space. It can be shown that for a small near plane offset three levels are enough. Three is also the minimum because, if the filter kernel at the entry point is only slightly below the next mipmap level, it might exceed it when the cone leaves the brick. For proper blending three levels are a must.

To make these bricks available for traversal, we collect them during the top-down traversal. We use a small queue of three elements implemented in shader registers without using dynamic indexing operations. Storing data only in leaves and using neighbor pointers to step through the volume are both elements that would make appropriate filtering very difficult.

The recovered color and opacity values are then accumulated with pre-integrated transfer functions (e.g., [EKE01]). A phase function and pseudo-Phong lighting (using the density gradient as the normal) can easily be accounted for.

5.3 Tree Updates via Interrupting and Resuming

So far, we have discussed the rendering procedure in the case that all data is present, but for complex data, not all of it might reside in GPU memory. Now, the simplest way to achieve a correct output image is to start the ray tracing process and to stop rays whenever data is missing. This is communicated to the CPU, which initiates the loading, then the algorithm resumes at the last ray position. The consequence is that several passes are performed to complete the output for a given frame. To only treat active rays, we use a standard technique in GPU ray tracing which is to block terminated pixels with the Z-Buffer, relaying on the z-cull and early-Z GPU features to prevent their execution.

Whenever we stop a ray, we need to maintain some status information about the ray: its preliminary output color accumulation, the missing node ID, and the current marching position in space. We can fit this information in two Multiple Render Targets (*MRTs*). The missing IDs are downloaded to the CPU that then adapts the data structure.

During the next rendering pass, initiated with a proxy surface, these buffers are used as input texture and the earlyZ technique will ensure that only active pixels will have the fragment shader executed. Even though this is very rudimentary, this algorithm already shows some nice properties. Loaded data is directly traversed by the ray, and no data lying outside the frustum will ever be loaded, even though we did not explicitly test for it. Further, the CPU does not need to track the needed LODs, the rays themselves will indicate through their traversal which precision is needed in each area of the screen. Nevertheless, the simple system as described so far would not know what data became useless. In the following, we will discuss a more complex LRU system and modifications of the basic algorithm that allow for real-time performance.

6 Advanced Algorithm

We discussed our implementation of the spatial subdivision structure that enables rendering of large volumetric models. We also explained how rendering is performed, and presented a simple method to update the data in the tree. This first approach in Section 5.3 interrupts ray tracing if data is missing, triggers an update and resumes once the new data has been uploaded to the GPU. This leads to accurately rendered images but costs much time.

The second display mode we will discuss now introduces some approximation. If data at the needed resolution is missing, a higher mipmap level is tried. Only if none is present or the available data is too coarse, we resort to the iterative method we have seen before. Using higher mipmap levels is often a good choice, as it has been shown that we perceive less details during strong motion, when it is most likely to encounter missing data. Once the camera settles, an accurate rendering is achieved within a few frames. Fixing a budget for the upload thus delivers a relatively stable frame rate.

Nonetheless, information about node usage still needs to be communicated to the CPU and not only for the first hit node, but for the entire trajectory of the ray. In this section, we will discuss our solution to this problem.

6.1 Combining Rendering and Visibility

The basic principle is to subdivide or fuse nodes in the N^3-tree in an LRU manner. Each node and brick has a timestamp that is reset upon usage and whenever a new element is added it will replace the oldest. Let's suppose for a moment that the CPU had access to this information for all rays. Then it can conveniently update the timestamp of the pool elements (nodes and bricks) in its mirrored data structure. Furthermore, it can trigger the loading of needed data. It even knows where to store the brick in the pool because timestamps are maintained in host memory. All necessary modifications are then transferred to the GPU via texture-update calls. This leads to a unified management of the *brick pool* and the *node pool* as two LRU controlled caches.

The question that remains is how to determine which nodes were used, which ones need refinement or can be coarsened, and what

data is missing on the GPU. In [GMAG08], this information is derived by evaluating node LOD's and frustum containment through tree traversal on the CPU, that induces a significant overhead. Further, visibility is determined via occlusion queries against the rendered image of the previous frame. But even when interleaved intelligently, occlusion queries result in a significant cost.

Following our strive to establish volume data as a rendering primitive, we want to avoid much of the CPU interaction. Ultimately, the GPU should steer which data is loaded and indicate what data was helpful in the previous frames. Interestingly, this information is available on the GPU when ray tracing terminates because the current volume was just traversed. This is the key insight that motivated us to combine rendering and visibility update phase.

In our solution, besides the output color, each ray also outputs some supplementary information about data usage or update needs. Because the information is collected by the rays, frustum containment, visibility and LOD selection are naturally handled in one unified way and all workload is taken off the CPU.

There are two major problems that needs to be addressed and that we will describe in more detail in the following sections. First and especially in the presence of transparency, each ray traverses a large number of nodes, but we can only output a limited amount of information per fragment in the framebuffer (even when using MRTs). Ad-hoc solutions, e.g., storing only the first node, would not lead to acceptable results with a LRU scheme. Second, this information needs to be sent back to the CPU to trigger the updates. As bandwidth is limited, we will show how to perform a compaction before data exchange.

6.1.1 Rendered Feedback Information

During rendering we traverse the tree and stop when we reach the node corresponding to the needed level-of-detail. If the required data is present, we traverse the mipmapped brick. To make sure that the CPU keeps these elements in the pool, we collect the current node index in a *node-list* (it will be the CPU that makes sure that all the node's parents needed for the mipmapping also remain in the cache). If the appropriate level-of-detail is missing or the node is *terminal*, meaning that there is no more refined data in the original model, we simply add it to the node-list. Remember that this information is indicated via one bit in the node (Figure 5). Otherwise, we also add it to the node-list, but indicate that it needs further subdivision. In practice, it works best to limit subdivisions to one request per ray. A single subdivision might already change opacity and allow an early ray termination in the next frame.

As stated before, arbitrarily sized node-lists cannot be efficiently output on current GPUs (currently only eight RGBA32 render targets are possible). An adapted repartition of the data is necessary. We reserve the first render target for the output color. The remaining buffers will keep the node information and we will refer to them as the *node-list textures*.

Each node-list texture provides room for four encountered nodes (one per channel) and their respective subdivision tag. This indicator will make the CPU simply reset the timestamp, or induce a subdivision because the needed level-of-detail is currently missing. Interestingly, there is no need to send collapse information. If the rendering algorithm no longer descends into a node, its index will never be put into the list, thus the LRU mechanism will not reset its timestamp and thus replace the data at some point. This establishes a lazy evaluation scheme which simplifies substantially previous solutions that needed to maintain the tree actively.

Using 30 bits for the node index, 1 bit for the subdivision tag, and seven node-list textures allow for a $7 \times 4 = 28$-node output per ray. Generally, this is insufficient for complex scenes. To ensure that no nodes are missed, we exploit two important properties of our algorithm: spatial (on screen), and temporal coherence (between successive frames). This will introduce some additional computations and in practice we found that three MRTs are the best choice. Hence, each pixel can store $3 \times 4 = 12$ nodes.

Neighboring rays will visit mostly similar nodes because the brick in a node always projects onto several pixels. To exploit this spatial coherence, we group rays in packages in the image plane. E.g., for a package size of 2×2, the upper-left ray will store the first 12 traversed nodes, the upper-right the second 12 nodes, and so on (cf. Figure 7). This totals in 48 node indices.

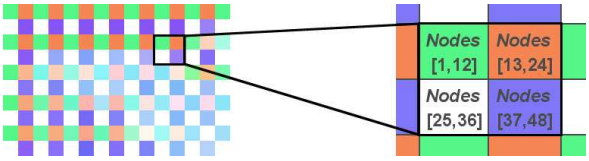

Figure 7: *We exploit spatial coherence using a pattern attributing different node sets.*

In addition, temporal coherence can be used. During traversal the rendered nodes are pushed in a FIFO. In the first frame, we stop pushing after 48, in the second frame after 96 elements. This 48-element window is shifted over a small set of frames, this increases the number of retrievable nodes further. For regions with less nodes, one should notice that the use of a FIFO ensures an information output in each frame. There is no penalization due to temporal coherence. In consequence, we gain reactivity where we can.

6.1.2 Compaction of Update Information

To simplify explanations, we will ignore the spatial-coherence pattern and assume that 12 nodes per ray suffice.

Once the node-list textures are constructed, it would be possible to read back the memory to the CPU, but the resulting bandwidth would be enormous and parsing additionally costly on CPU. Ideally, one instance of each node reference would be enough. A naive way to achieve this consists in sorting the node indices on the GPU and then use a stream reduction to remove multiple entries. Sorting is expensive and we would further sort many elements that would afterwards just be deleted. Fortunately, we just saw that coherence between neighboring rays is a powerful property and can be used to compaction the texture. It is possible to reduce the set sufficiently to make the sorting step unnecessary.

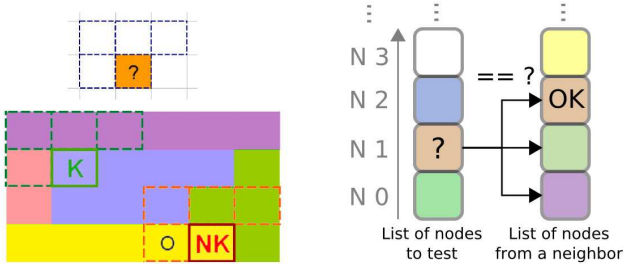

Figure 8: *Selection step: (Left:) Lists are compared following a 2D pattern - (N)K=(Not) Kept. (Right:) List entries are only compared on a local scope.*

In a first pass, we derive a *selection mask*. Here, the the i-th bit in an output pixel indicates that the i-th element in the list of the underlying node-list texel should be kept. The selection mask is based on a local comparison. We use a pattern like the one shown in Figure 8. It compares the center list (orange, Figure 8-top-right) against its surrounding lists and an entry in the list is only kept if it does not appear in the surrounding (Figure 8-left). Theoretically, this can be costly if the lists are long, but most nodes will be crossed approximatively at the same moment by two neighboring rays. This

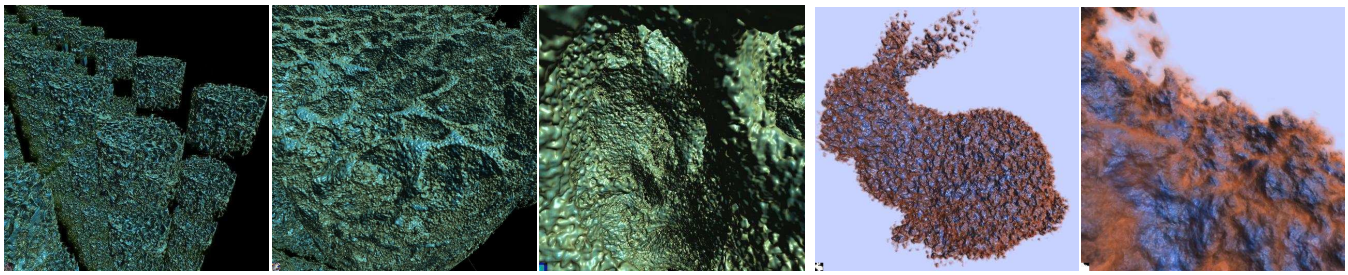

Figure 9: *Some results: Trabecular bone data (augmented with Perlin noise) and Hypertextured bunny (based on a distance field approximated on the fly on the GPU)*

allows us to perform an efficient approximate comparison: the i-th list element is only tested against the (i-1)-th, i-th, (i+1)-th elements of the neighboring lists (Figure 8-right). This can induce "false positives", but it remains conservative.

After this first step, each pixel of the selection mask contains a bitvector, whose non-zero entries represent the list elements to keep. We rely on the HistoPyramids [ZTTS06] to reduce the selection mask. This algorithm is a special case of the prefix-sum scan and binary search algorithm, highly optimized to exploit 2D locality and GPU's texture processing units. The final step recovers the actual corresponding node indices, then the compacted texture is transferred to the CPU. During this rearrangement process, the original pixel position will be lost. Having 12 node entries per list, we can avoid this problem by using the remaining 20 bits (over 32) to encode the pixel position of the ray. Only with this location information we will be able to recover the actual node indices.

After the reduction, each pixel contains a ray pixel-position and a bitmask. This information allows the creation of a compact node index texture that will be read back by the CPU. In practice, the bitmask usually contains 2-3 non-zero entries. It is thus possible to make the final compacted texture one single RGBA32 texture which allows us to store indices of four nodes. If this limit is exceeded the requests are automatically postponed to the next frame.

7 Results

All tests were performed on a Core2 bi-core E6600 at 2.4 GHz, and an NVIDIA 8800 GTS 512 graphics card (G92 GPU) with 512 MB. All images are rendered at 512×512 (see Figures 1 and 9).

Example 1: *Explicit volume (trabecular bone).*
We used a 1024^3 scanned volume of a trabecular bone. Mipmaps were precomputed and analyzed for constant density. This example still fit in the CPU memory. 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 $M = 16$. The algorithm achieves 20-40 Hz with smooth mipmapping activated. Without, an average of 60fps was possible. We also added scales of Perlin noise to increase richness. This noise is added procedurally on the GPU. Evaluating noise in each frame would be expensive, at a low penalty, we create bricks on the GPU that we can then use in subsequent frames. In comparison to [GMAG08], this indicates approximately a 50% speedup (nevertheless, this is based on our implementation).

Example 2: *Procedural volume by instantiation (Sierpinski).*
We used one unique brick of size 81^3. We naturally chose $N = 3$ so that we can rely on one unique instantiation for all non-empty children. 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 $8.4M^3$. Performance often reaches 90 Hz, and usually stays around 60.

Example 3: *Hypertextures and amplification of a mesh.*
For this example, we use a volume defined by the interior of a mesh. We derive a distance field from the mesh on the fly on the GPU. and use the surface vicinity for hypertexture lookups. Here we aimed at very high quality and use 20 octaves of Perlin noise, shading, and complex materials. Due to the complex computations on the GPU to produce the volume information, the frame rate is relatively low (around 20 FPS for a 1024^3 volume).

Example 4: *Cumulus cloud.*
Our method was used to encode the cloud details in [BNM*08] (a paper dealing with multiple scattering in clouds). Even in combination with complex shading [BNM*08], our algorithm allows for an increase of detail. We used $N = 2$, $M = 32$, and 5 octaves of Perlin noise to simulate enhance the cumulus cloud, with a virtual resolution of 2048^3.

Memory usage: In most examples, the node pool was small (4 MB) corresponding to 64^3 entries. Using 16^3 bricks, 1024^3 indices can be referenced.The brick pool used 430 MB giving room for 42^3 bricks.

We usually trade-off some computation time for memory efficiency: in all our examples, normals are computed on-the-fly. The procedural noise examples further showed that bricks can also be created on the GPU. This on-the-fly noise creation proves actually to be more efficient than a CPU evaluation and transfer.

8 Conclusion and Future Work

We have presented a method for interactive rendering of large and very detailed volumes. Our work shows that real-time performance with high quality volume rendering is possible. The algorithm avoids most of the CPU interaction. Our compact data structure minimizes memory usage on the GPU side. The introduction of smooth transitions based on mipmapping allow temporal coherence and anti-aliasing. This hints at the possibility that volume data could be an important future real-time primitive.

Currently, animation is a big problem for volume data. In the future, we would like to investigate possible solutions.

Another interesting avenue would be to apply our method to other hierarchical representations: general mesh subdivisions, point rendering, or recent structures like volume-surface trees [BHGS06].

We would like to thank: Digisens for their support; Antoine Bouthors, Eric Bruneton, and Hedlena Bezerra for proof reading, and the anonymous reviewers for their helpful comments. This work has been partly supported by the Excellence Cluster on Multimodal Computing and Interaction (MMCI - www.m2ci.org).

References

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

BOUBEKEUR T., HEIDRICH W., GRANIER X., SCHLICK C.: Volume-surface trees. *Computer Graphics Forum 25*, 3 (2006), 399–409. Proceedings of EUROGRAPHICS 2006.

BOUTHORS A., NEYRET F., MAX N., BRUNETON E., CRASSIN C.: Interactive multiple anisotropic scattering in clouds. In *ACM Symposium on Interactive 3D Graphics and Games (I3D)* (2008).

BOADA I., NAVAZO I., SCOPIGNO R.: Multiresolution volume visualization with a texture-based octree. *Vis. Comput. 13*, 3 (2001).

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

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

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

DECAUDIN P., NEYRET F.: Volumetric billboards. *Computer Graphics Forum* (2009).

ENGEL K., HADWIGER M., KNISS J., REZK-SALAMA C., WEISKOPF D.: *Real-time Volume Graphics*. AK-Peters, 2006.

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.

GOBBETTI E., MARTON F.: Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH)* (2005), ACM.

GOBBETTI E., MARTON F., ANTONIO J., GUITIAN I.: A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *Vis. Comput. 24*, 7 (2008), 797–806.

GUTHE S., STRASSER W.: Advanced techniques for high quality multiresolution volume rendering. In *Computers & Graphics* (2004), Elsevier Science, pp. 51–58.

HONG W., QIU F., KAUFMAN A.: GPU-based object-order ray-casting for large datasets. In *Volume Graphics, Fourth International Workshop on* (2005), pp. 177–240.

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).

KAPLER A.: Avalanche! snowy FX for XXX. In *SIGGRAPH Sketch* (2003). http://portal.acm.org/ft_gateway.cfm?id=965492 .

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

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

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 Sketch* (1998). http://portal.acm.org/ft_gateway.cfm?id=282285 .

KAJIYA J. T., KAY T. L.: Rendering fur with three dimensional textures. In *SIGGRAPH* (1989), pp. 271–280.

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

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

LAMAR E., HAMANN B., JOY K. I.: Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of Visualization (VIS)* (1999), pp. 355–361.

LEFEBVRE S., HORNUS S., NEYRET F.: *GPU Gems 2*. 2005, ch. Octree Textures on the GPU, pp. 595–613.

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

LEFOHN A., KNISS J. M., STRZODKA R., SENGUPTA S., OWENS J. D.: Glift: Generic, Efficient, Random-Access GPU Data Structures. *ACM Transactions on Graphics 25*, 1 (2006).

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

LI W., MUELLER K., KAUFMAN A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of IEEE Visualization (VIS)* (2003), p. 42.

MÖLLER T., MACHIRAJU R., MUELLER K., YAGEL R.: A comparison of normal estimation schemes. In *Proceedings of VIS (IEEE Conference on Visualization)* (1997), pp. 19–.

MEYER A., NEYRET F.: Multiscale shaders for the efficient realistic rendering of pine-trees. In *Proceedings of GI (Graphics Interface)* (2000).

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.

OLIVEIRA M. M., BISHOP G., MCALLISTER D.: Relief texture mapping. In *SIGGRAPH* (2000), pp. 359–368.

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

PERLIN K., HOFFERT E. M.: Hypertexture. In *SIGGRAPH* (1989), pp. 253–262.

POLICARPO F., OLIVEIRA M.: *GPU Gems 3*. Addison-Wesley, 2007, ch. 18: Relaxed Cone Stepping For Relief Mapping.

RECHE-MARTINEZ A., MARTIN I., DRETTAKIS G.: Volumetric reconstruction and interactive rendering of trees from photographs. In *SIGGRAPH proceedings* (2004), pp. 720–727.

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

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).

VOLLRATH J. E., SCHAFHITZEL T., ERTL T.: Employing Complex GPU Data Structures for the Interactive Visualization of Adaptive Mesh Refinement Data. In *of the International Workshop on Volume Graphics* (2006).

ZIEGLER G., TEVS A., THEOBALT C., SEIDEL H.-P.: GPU point list generation through histogram pyramids. In *Proc. of VMV* (2006), pp. 137–141.