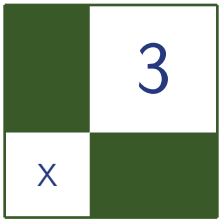


Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels

Cyril Crassin, Fabrice Neyret,
Miguel Sainz, and Elmar Eisemann

3.1 Introduction

GigaVoxels is a voxel-based rendering pipeline that makes the display of very large volumetric datasets very efficient. It is adapted to memory-bound environments and is designed for the data-parallel architecture of the GPU. It is capable of

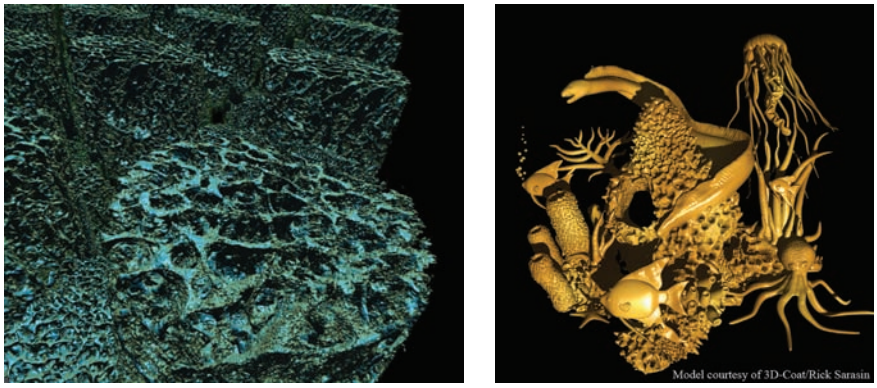


Figure 3.1. Examples of scenes composed, respectively, of 8192^3 and 2048^3 voxels rendered interactively using our engine.

rendering objects at a detail level that matches the screen resolution and interactively adapts to the current point of view. Invisible parts are never even considered for contribution to the final image (see Figure 3.1). As a result, the algorithm obtains interactive to real-time frame rates and demonstrates the use of extreme amounts of voxels in rendering, which is applicable in many different contexts. This is also confirmed by many game developers who seriously consider voxels as a potential standard primitive in video games. We will also show in this chapter that voxels are already powerful primitives that, for some rendering tasks, achieve higher performance than triangle-based representations.

3.2 Voxel Representations

The name *voxel* comes from "volumetric elements" and it represents the generalization in three-dimensional of the pixels. Voxels are axis-aligned cubes that contain some data, usually a density value, or a color value and opacity. Voxels are well-suited for complex or fuzzy data, such as clouds, smoke or foam. They handle semi-transparency gracefully due to their inherent spatial organization, whereas triangles would need costly sorting operations. Further we show that they offer a promising way to unify texture and geometry simplifying filtering simpler and making voxels a good candidate to address aliasing issues that are hard to deal with for triangulated models.

Multi-resolution representations are easily obtainable, making output-sensitive results possible. Basically, you only pay for what you see. This allows us to represent very detailed scenes with gigantic voxel volumes at high frame rates with our GPU-based implementation. Very large terrains or detailed models are particularly interesting targets for voxel representations. The most prominent feature from an implementation standpoint is that the level of detail (LOD) adjustments are automatically handled and no special object testing is needed.

Even though our main focus will be on games, voxels are often used in other contexts, such as scientific visualization or offline rendering by visual effect companies. The latter are particularly drawn to voxels due to the high rendering quality [Kapler 03, Krall and Harrington 05], which we maintain to a large extent in our approach.

Despite their many advantages, the use of voxels has drawbacks and there are reasons why they are less often employed than their triangular counterpart. Triangles have been natively supported on dedicated graphics hardware since the beginning. Real-time, high-quality voxel rendering has only become feasible since the introduction of programmable shaders. But there is a more general problem, which is that detailed representations use gigantic amounts of memory that cannot be stored on the graphics card. Hence, until recently, voxel effects in video games

were mostly limited to small volumes for gaseous materials or to the particular scenario of height-field rendering. The development of an entire rendering system, capable of displaying complex voxel data, is all but trivial and has been in the way of more involved usage of volumetric rendering in real-time applications. This chapter will propose an approach to overcome the difficulties related to large volume rendering.

3.3 The GigaVoxels Approach

To address the display of memory-intensive voxel data, we proposed *GigaVoxels* [Crassin et al. 09], a new rendering pipeline for high-performance visualization of large and detailed volumetric objects and scenes. We showed that entire volumetric environments can be displayed in real time without the need to revert to a triangle-based representation. Voxels increase the amount of displayable detail significantly beyond the limits of what can be achieved with polygons at similar frame rates and support transparency effects natively. Our system is entirely GPU-based and achieves real-time performance for scenes consisting of billions of voxels.

In this chapter, we will present a full NVIDIA CUDA implementation of the GigaVoxels engine (see Figure 3.2). CUDA is a general purpose parallel computing language that allows to program directly the data-parallel architecture of NVIDIA GPUs [NVIDIA 09]. Among other things, CUDA enabled us to implement a very efficient cache mechanism managed entirely on the GPU. The CPU workload is

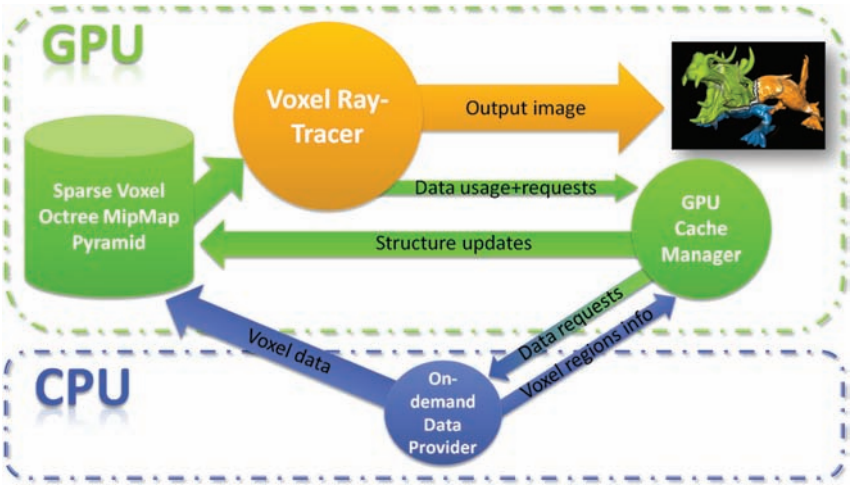


Figure 3.2. Global view of the GigaVoxels rendering engine.

very low and limited to data transfer, with no longer needs to track and manage the memory layout on the CPU.

Our data representation is a sparse voxel hierarchy that allows to efficiently encode and compress constant regions in space (Section 3.4). The rendering is based on a ray casting algorithm (Section 3.5). The rays themselves directly trigger a paging system to request missing data (Section 3.6). This paging system is based on a cache mechanism fully implemented on GPU that maintains newly used data in video memory, while recycling oldest (Section 3.6.2).

While we will first focus on a single volume, we will also show how to manage entire scenes of disjoint volumetric elements (Sections 3.7, 3.8). Finally we explain how to add game effects, such as soft shadows and depth of field approximations (Section 3.9).

3.4 Data Structure

The key to our efficient rendering technique is a hierarchical representation of the voxel data, efficient for both dynamic storage and rendering. It allows us to adapt the volume's internal resolution, to compact empty spaces, and to omit occluded parts according to the current point of view. This greatly reduces the memory consumption and avoids storing the entire information on the GPU.

In practice, the initial volumetric data set is embedded in a space subdivision structure, in form of an octree. To each node of this octree, a voxel volume of small

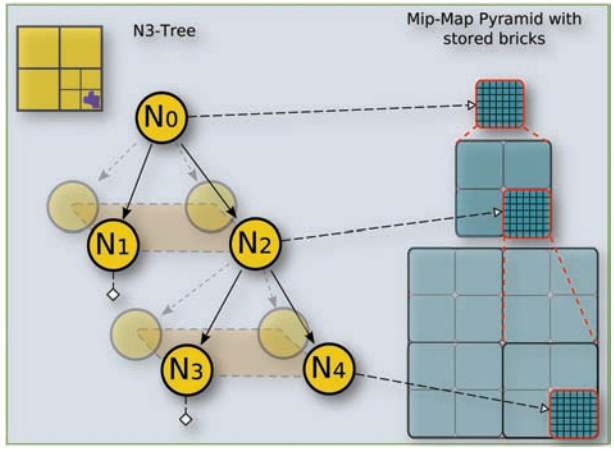


Figure 3.3. Our sparse voxel octree data structure. This structure stores a whole voxel scene or object filtered at multiple resolutions. Bricks are referenced by octree nodes providing a sparse mipmap pyramid of the voxels.

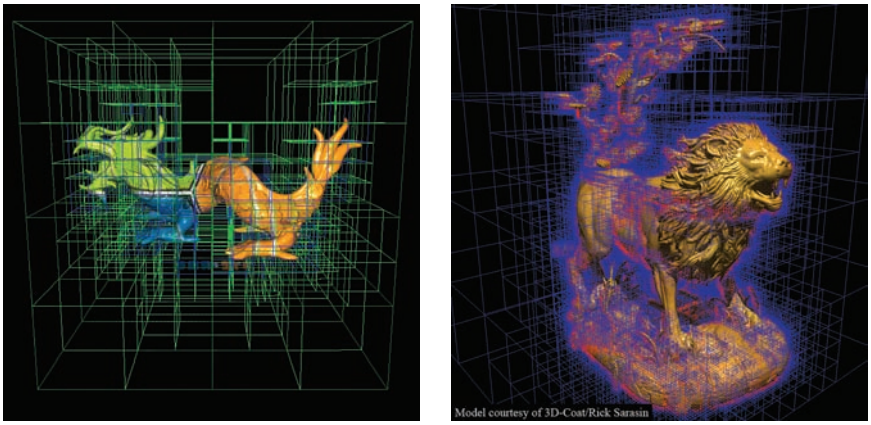


Figure 3.4. Spatial partitioning of the octree structure. A voxelized version of the Stanford XYZRGB-Dragon model rendered at around 50FPS and a 2048^3 voxels lion model modelized directly in voxels using 3D-Coat software and rendered at 20–40FPS with GigaVoxels.

resolution, e.g., 16^3 or 32^3 is associated, which we refer to as *brick*. Each node is associated to a certain spatial extent in the entire volume, and eight children always cover exactly the spatial extent of their parent.

Figure 3.3 summarizes the data structure of our approach and will be of help during the following discussion. Such a structure combines two major advantages: the octree leads to a compact storage (empty space skipping, occlusion culling, local refinement), whereas the bricks are implemented as small three-dimensional textures and, thus, benefit from hardware-based interpolation and cache coherence (see Figure 3.4). Another advantage is the constant size of the node and brick data types. Consequently, it is simple to densely store them in memory pools on the GPU. This facilitates the update mechanisms that are crucial to ensure the presence of the data needed to produce the output image.

An important property of this structure is that it provides information at multiple scales, in order to match the resolution needed for a given point of view. It is built from a precomputed mipmap pyramid, obtained by downsampling the voxel object.

3.4.1 Structure Storage: The Pools

Our data is stored in two GPU memory regions; the *node pool* stores the octree as a pointer-based tree and the bricks are organized into a *brick pool*. These pools are used as caches (see Section 3.6.2), and their size is fixed once at initialization

time. In a video game context, they would be chosen depending on available video memory and bandwidth between GPU and CPU.

In order to be modifiable directly from a CUDA kernel, the node pool is implemented in global linear GPU memory. It is accessed from a CUDA kernel as a *linear texture*, in order to take advantage of the texture cache. Instead of storing each node separately, a single entity in the node cache actually regroups eight octree nodes together. The reason will become clear in the next section, but intuitively, these eight elements correspond exactly to the eight subnodes of the same parent node. We call this $2 \times 2 \times 2$ nodes organization a *node tile*.

The brick pool is implemented in a CUDA Array (that is a three-dimensional texture), in order to be able to use three-dimensional addressing, hardware texture interpolation, as well as a three-dimensional coherent cache. To ensure correct interpolation, we store bricks with an additional one voxel border. A current limitation of CUDA Arrays is that the direct modification from a CUDA kernel is not exposed in the current version CUDA 2.2.

3.4.2 Octree Implementation

Figure 3.5 shows the data elements in an octree node. Each node is composed of a data entry, and a subnode (child) pointer. The data can either be a constant value (for empty/homogeneous volume), or a pointer towards a brick (a small texture). As indicated before, each entity in the node pool corresponds to eight subnodes. It is exactly this property that allows us to only rely on a single child pointer. Because all subnodes are stored contiguously in memory, one can address any single one by adding a constant offset to the pointer. Figure 3.6 illustrates a simple example. Note that this organization also allows the structure to be flexible enough to manipulate generalized N^3-trees instead of simple octrees (which is an N^3-tree with $N = 2$), subdividing space into N on each axis. Such structure can provide interesting properties as explained in [Crassin et al. 09].

This structure produces a very compact octree encoding as each node is represented by only two 32-bit integer values. The first integer is used to define the octree structure, the second to associate volume data. With the first, one bit

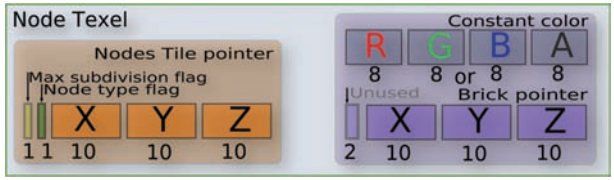


Figure 3.5. Compact octree node encoding into two 32-bit values.

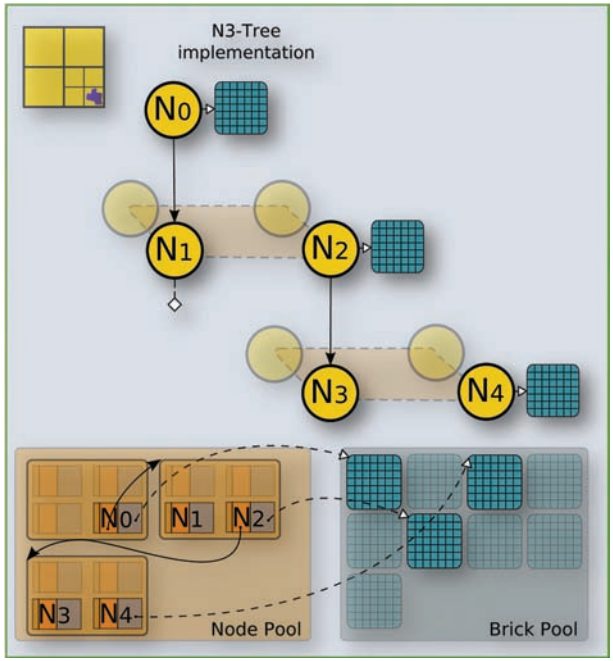
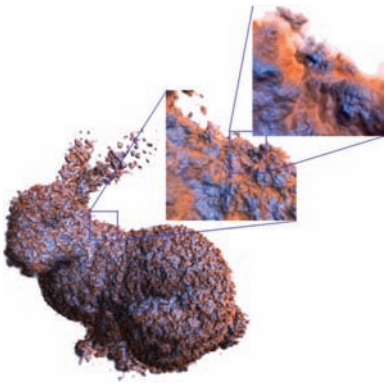


Figure 3.6. Our sparse voxel octree structure encoded in the node pools.

indicates whether a node is *terminal*, which means *refined to a maximum*, i.e., whether the original volume still contains more data and so the node could be subdivided if needed. Another bit is used to indicate the nature of the second integer value; either it represents a constant color or a pointer to a brick encoded on 30 bits. The same amount, 30 bits, is used for the child pointer to the subnodes. The pointers describe the position of the corresponding element with three 10-bit xyz-coordinates and are zero if there is no data to point to.

The potential performance penalty during traversal, due to missing links between neighboring voxels, is highly outweighed by the small memory requirements. In fact, the compactness and coherence allow for a very efficient ray casting (Section 3.5.1). Further, the structure is intended to adapt to the current viewpoint and a smaller memory footprint implies a more efficient and simpler update.

To enhance traversal efficiency during rendering, both parts of a node description (subnode pointer and data, see Figure 3.5) are not interleaved in memory, but stored in two separate arrays. Indeed, since each part of the node description is not used in the same rendering sequence, this allows us to further enhance data access coalescing and texture cache efficiency.



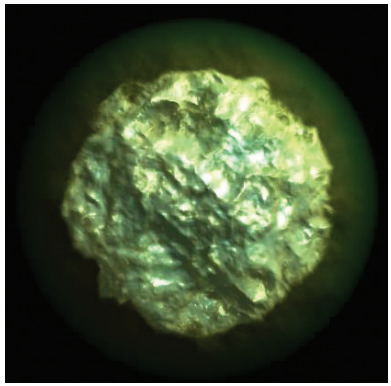


Figure 3.7. Examples of participating medias rendered with GigaVoxels.

3.5 Rendering

Since our rendering engine is designed to deal with semi-transparent voxel data, we need to evaluate the final color that reaches the eye after the compositing of various colored voxels (see Figure 3.7). Intuitively, this means that one has to integrate the voxels color and opacity along the ray. To do so, we evaluate a simple optical model describing the light transfer inside the volume. We use the Emission-Absorption optical model, neglecting scattering and indirect illumination. For a survey on GPU volume rendering, we refer the reader to the book [Hadwiger et al. 06]. In this section, we explain only the rendering component of our system. Currently, we assume that the present data structure contains all necessary data for rendering. Refinement and update mechanisms will be discussed later in Section 3.6.

3.5.1 Hierarchical Volume Ray Casting

The color of each pixel is evaluated by traversing the structure using volume ray casting [Kruger and Westermann 03, Roettger et al. 03, Scharsach 05], executed by a CUDA kernel (see Figure 3.8). We generate one thread per screen pixel, each tracking one ray through the structure in order to evaluate the volume-rendering equation corresponding to the optical model. The ray traversal is front to back. Starting from the near plane[1], we accumulate color C and opacity α, until we leave the volume, or the opacity saturates (meaning that the matter becomes opaque), so that farther elements would be hidden.

[1] Alternatively, another proxy surface can be used to initialize the rays, e.g., a bounding box or a mesh surface.

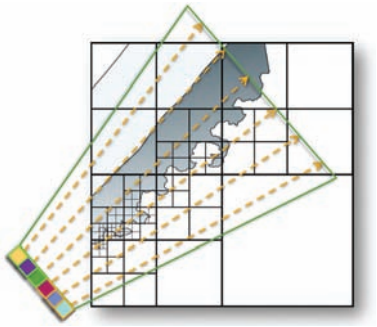


Figure 3.8. Illustration of the volume ray casting process launching one ray per pixel and traversing the hierarchical structure.

3.5.2 Octree and Node Traversal

The octree structure is traversed using a stackless algorithm similar to the kd-restart algorithm presented in [Horn et al. 07] and developed for kd-tree traversal (see Figure 3.9).

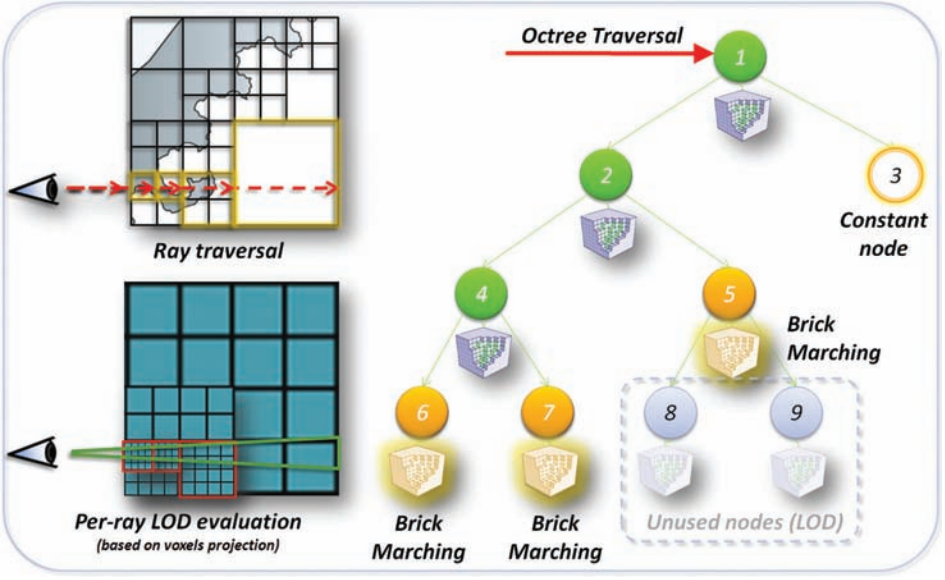


Figure 3.9. Illustration of the structure traversal with bricks marching and LOD computations based on projected voxel size.

This traversal is in fact a series of successive top-down descents in the tree. For each descent, a ray starts from the root node and then proceeds downwards (Section 3.5.4). The descent stops at a node whose resolution is sufficient enough for the current view (Section 3.5.3).

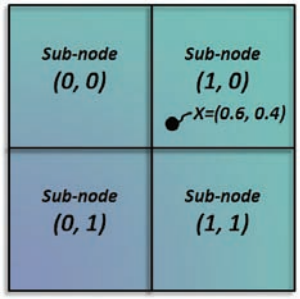
Such a node either contains a constant value or a pointer to a brick. For a constant node, the distance between entry and exit point is used to analytically integrate the constant color along the ray.

For a brick, we rely on a ray marching to accumulate samples and evaluate the volume rendering equation. More details on ray marching can be found in [Engel et al. 01, Scharsach 05, Hadwiger et al. 06]. For each sample read in a brick, we take advantage of hardware trilinear interpolation. The stepping distance between these samples is computed relative to the LOD mechanism described Section 3.5.4. It is possible to include a direct illumination component for opaque objects by storing normals per voxel and by applying a Blinn-Phong model. Once the ray leaves the brick, we use the new ray position as the origin for the next top-down descent in the tree.

3.5.3 Descending in the Octree

The descent is fast because, following [Lefebvre et al. 05], the point's coordinates can be used directly to locate it in the octree as illustrated in Figure 3.10.

Let $x \in [0,1]^3$ be the point's local coordinates in the node's bounding box and c be the pointer to its children. Since subnodes are stored contiguously in the node pool (see Section 3.4.2) a three-dimensional offset is enough to select a child. In practice, we store the children in a way such that the node containing x is at location $int(2x)$, the integer part of $2x$ on each axis: e.g., in one-dimensional for one axis with $x_{axis} \in [0,1]$, there are two possible offset values for the children,



$$\begin{aligned} \text{Index}_{2D} &= \text{int}(x * N) \\ &= \text{int}((0.6 * 2, 0.4 * 2)) \\ &= \text{int}((1.2, 0.8)) \\ &= (1, 0) \end{aligned}$$

Figure 3.10. Two-dimensional localization into a node tile to find the index of the subnode where x lies.

namely 0 ($x < 0.5$) and 1 ($x >= 0.5$). In general, to descend into the subnode containing x, we can thus use the pointer $c + \text{int}(2x)$. We then update x to $2x - \text{int}(2x)$ and continue the descent[2].

3.5.4 Tracing Cones: LOD and Volume MipMapping

We have seen in the last section how to do an efficient descent in the octree in order to find successive bricks. We will now explain how to choose when to stop such a descent and how to then traverse its associated volume data. In order to find our criterion, we will investigate how data should be integrated in the per-pixel color determination.

One screen pixel is associated to more than just a line in space. It actually corresponds to a cone because a pixel covers an area and not a single point (as illustrated in Figure 3.11). This is typically the source of aliasing that arises when a single ray is used to sample the scene.

Whereas classical rasterization or ray tracing-based rendering relies on multi-sampling to deal with aliasing, we instead launch a single ray per pixel, and deal with the aliasing by filtering the voxel representation. More precisely, while traversing the scene along the ray, we chose our lookups from a prefiltered volume representation. In practice, we rely on a similar technique as usually employed for two-dimensional textures: mipmapping. In fact, our data structure already encodes a sparse mipmap pyramid in order to provide different levels of detail. Each such level can be interpreted as a filtered version of the volume. At each traversal step, we select the volume resolution (and so the depth in the octree), in order to account for the volume hit simultaneously along a cone. The step size

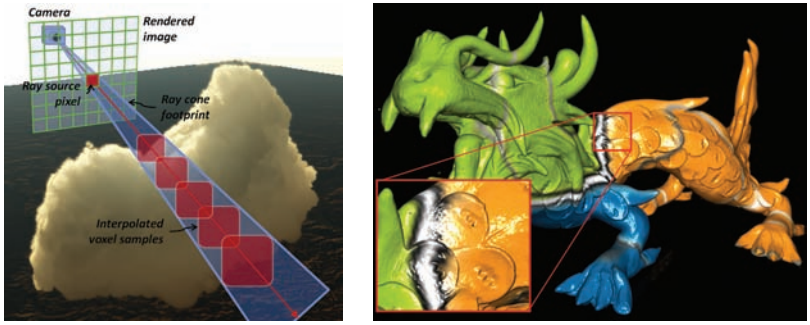


Figure 3.11. Left: illustration of the cone footprint of a ray launched from one pixel using perspective projection. Right: the XYZRGB-dragon rendered with volume mipmapping.

[2]Note that this descent can be generalized to N^3-Trees using $c + \text{int}(x * N)$, and updating x with $x * N - \text{int}(x * N)$.

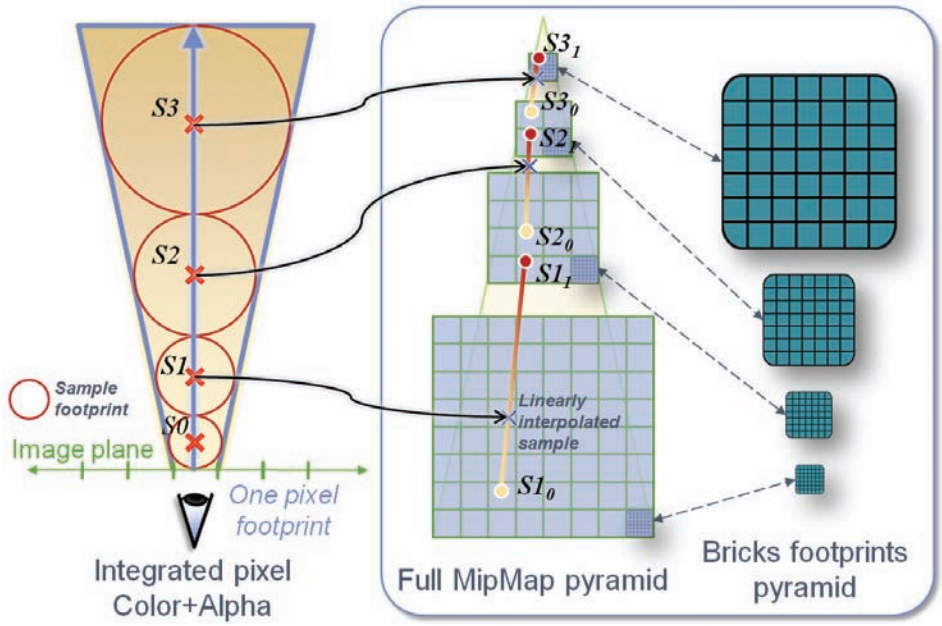


Figure 3.12. Illustration of one ray sampling using quadrilinear interpolation in the voxel mipmap pyramid.

is adapted accordingly in order to reflect the resolution of the corresponding level of detail. This principle is illustrated in Figure 3.12.

More precisely, during top-down traversal, we estimate the voxel size V via $V := \frac{N}{M}$, where N is the current node's side length and M^3 the brick resolution. We estimate its projection size from the viewpoint on the near plane by $V_{\text{proj}} := V\frac{n}{d}$, where d is the node's distance to the near plane n. Based on the field-of-view of the camera, we can then compare this size to the pixel's size P. If $V_{\text{proj}} < P$, the descent can be stopped and we traverse the current node.

This LOD selection allows us to always use the minimal voxel resolution required for a given point of view, minimizing the memory requirement and rendering cost.

Even though this approach is simple, it leads to a rough and discretized approximation of the cone footprint that produces resolution discontinuities especially visible during motion. Smooth transitions between bricks can only be achieved by making the voxels "grow" continuously along the ray. Consequently, we rely on a simultaneous interpolation between two mipmap levels, leading to a quadrilinear filtering. To have access to the lower resolution brick during the marching step,

```
//Brick value access with mipmapping. Templated with the data layer
//(color, normal, etc.) in which data is read.
template<int datalayer>
__device__
float4 getBrickValueMipMap(
        float3 posInPool0, float3 posInPool1, float3 posInPool2,
        float3 posInBrick, float nodeSize, float t,
        float2 minMaxMipMapLevel){
    //Compute mipmap level based on the distance to the view plane t
    //and the node size,
    float mipmaplevel=getMipMapLevel<mode>(t, nodeSize);
    //Clamp mimap level.
    mipmaplevel=clamp(mipmaplevel, minMaxMipMapLevel.x,
                      minMaxMipMapLevel.y);
    //Compute interpolation first level.
    float mipmaplevelI=floorf(mipmaplevel);
    float interp=(mipmaplevel-mipmaplevelI);
    float4 vox, vox0, vox1;
    float3 samplePos0, samplePos1; float3 brickPos0, brickPos1;
    //Select brick addresses.
    if(mipmaplevel<1.0f){
        brickPos0=posInPool0; brickPos1=posInPool1;
        samplePos0= (posInBrick); samplePos1= (posInBrick)*0.5f;
    }else
        brickPos0=posInPool1;    brickPos1=posInPool2;
        samplePos0= (posInBrick)*0.5f; samplePos1= (posInBrick)*0.25f;
    }
    //Get first trilinearly interpolated sample.
    vox0=getBrickValue<datalayer>(brickPos0, samplePos0);
    //Compute interpolation.
    const float interpThreshold=0.0001f;
    if(interp>interpThreshold){
        //Get second trilinearly interpolated sample.
        vox1=getBrickValue<datalayer>(brickPos1, samplePos1);
        //Compute linear interpolation.
        vox=vox0*(1.0f-interp)+vox1*(interp);
    }else
        vox=vox0;

    return vox;
}
```

Listing 3.1. Code of brick sampling function with mipmap quadrilinear interpolation.

we remember the last three visited nodes during the descent. It can be shown that this number is sufficient in practical scenarios. We will show later (in Section 3.9) that this mipmap mechanism can also be used to efficiently implement blur effects.

Listing 3.1 shows a CUDA implementation of the quadrilinear interpolated fetching of samples into a brick.

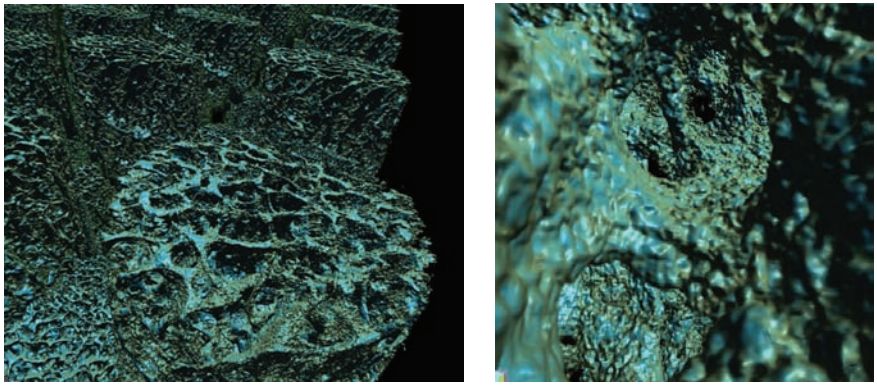


Figure 3.13. Examples of a very large 8192^3 voxel scene composed of medical data and rendered at 20–40FPS using GigaVoxels on an NVIDIA GTX 280 GPU.

3.6 Out-of-Core Data Management

So far we have seen how to efficiently use our data structure for rendering. Now, we will investigate how the GigaVoxels engine deals with arbitrarily large amounts of voxel data (see Figure 3.13).

Our whole data-management scheme is organized around a cache mechanism managing both the octree structure and the bricks (see Figure 3.14). The cache

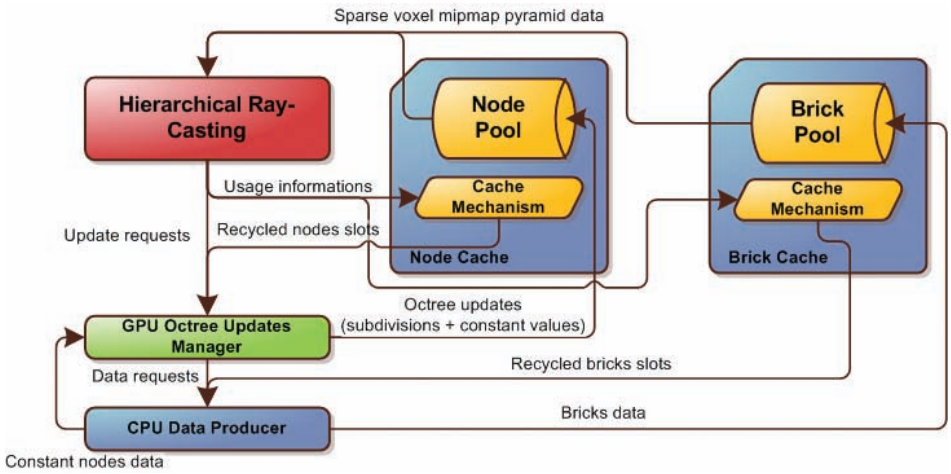


Figure 3.14. General view of the cache mechanism and data updates flows.

is in charge of maintaining the most recently used elements on the GPU, while providing room for new elements by recycling the oldest ones. This results in a fully scalable solution that can deal with extremely large volumes.

When the point of view changes, some missing data that was previously occluded, or out-of-frustum, might becomes visible. Updates to the structure are requested directly by the voxel ray tracer, on a per-ray basis. In this way, it is possible to only trigger data management based on what is actually needed for the current point of view. Each ray informs accurately about the needed data instead of having to heuristically estimate the need for data. This also leads to minimal CPU intervention, since both LOD and visibility computations are performed per-ray on the GPU.

3.6.1 Global Scheme

During the traversal of the hierarchical volume described in Section 3.5.1, rays visit several nodes in the structure. For some rays, the data needed to get the right volume resolution (according to the LOD mechanism described Section 3.5.4) might be missing. Nodes might need to be subdivided, implying the need for new data, or nodes could be present at the right depth, but might miss the according volume data. In both cases, a ray will *ask* for missing data by issuing an *update request*. This request will not be processed immediately, but instead be added to a batch of requests. Structure updates are then executed incrementally, from top to bottom, subdividing nodes one level at a time. This scheme ensures that unnecessary nodes (not accessed by rays because of occlusions for instance) will never be created nor filled with data. With this strategy, we avoid excessive data refinement and we reduce data requests, resulting in a higher frame rate.

The rendering process is organized into passes (see Figure 3.15). Each pass interleaves a ray casting phase, producing an image, a batch of requests, and an update phase. The update phase fulfills update requests by uploading new data to the GPU and updating the structure providing the rays with the necessary data for the next ray casting pass.

To provide a control over image quality during updates, multiple passes can be issued per frame. In this case, rays will stop whenever data is missing and traversal is continued from this point in the following pass in a stop-and-go manner. The number of passes authorized per frame depends on the application context. For games, it is acceptable that some frames show less details, to favor high frame rates which can be achieved by producing an image even when data is missing. For this rays can rely on some coarser data that is actually present in upper levels in the tree. For instance, for offline rendering, physical simulations, or movie productions, it is important that each frame is accurately computed. Hence the stop-and-go solution is most appropriate.

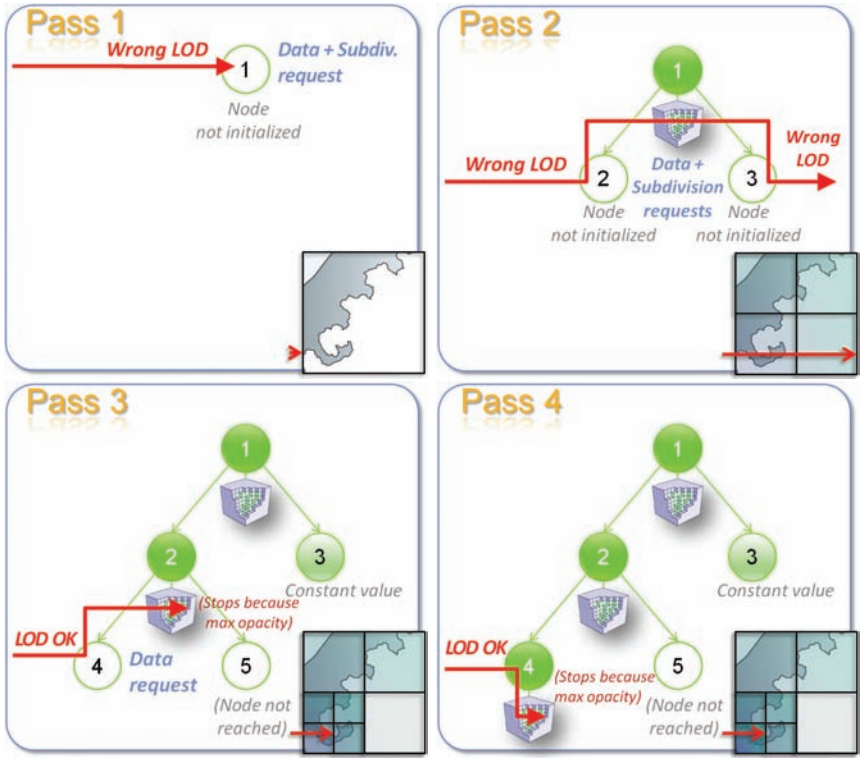


Figure 3.15. Illustration of ray-guided multi-pass update sequence for one ray. In pass (1), the ray hits a node not initialized and requests data and subdivision (since needed LOD is not reached). At the end of the pass, a brick is loaded for the node and it gets subdivided. In pass (2), the ray goes down, traverses newly created nodes and uses the higher level brick in order to produce an image. Data is requested for both nodes and LOD is still not reached. The first node gets a brick and the second, a constant value. In pass (3), the ray traverses the first new node and requests data for it. Subdivision is not requested since correct LOD is reached. The second new node is not touched because, due to opacity, the ray stops in the middle of the upper brick. Starting from pass (4) the ray got all data it needs and no more update is necessary.

The progressive top-down refinement scheme we use (first low, then high details) is very useful to ensure real-time responsiveness even in the case of fast movements and small time budgets per frames. The disadvantage of this strategy is that for fast movements, this can potentially lead to a data upload at the wrong resolution because even more precision might be needed. In practice, this is not objectionable. The lower resolution data is always eight times cheaper to load

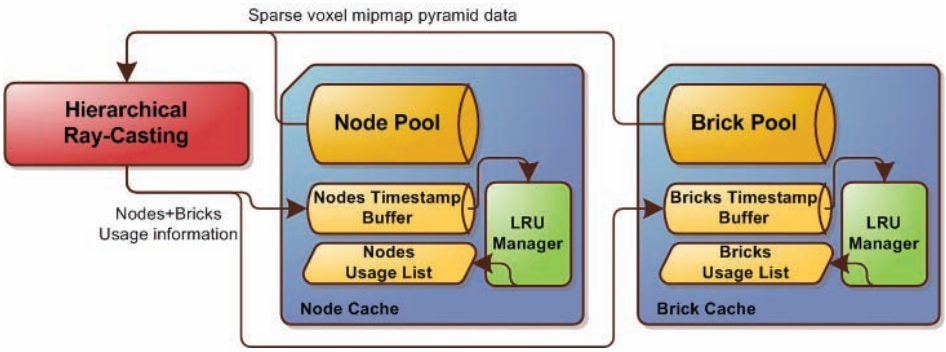


Figure 3.16. Global view of node pool and brick pool cache mechanisms with their GPU LRU management.

than the full resolution (the eight child nodes cover the same spatial extent as their parent) and more than a level is rarely jumped for reasonable motions.

In the following, we will first explain how our cache mechanism is implemented and then describe the ray based update mechanism.

3.6.2 GPU Cache Mechanism

The cache mechanism basically relies on a *least recently used* (LRU) scheme in order to allocate slots for new nodes or bricks, by recycling slots occupied by those elements that have not been used for a long time (see Figure 3.16).

In practice, both memory pools, node pool, and brick pool are managed as caches. In order to maintain these caches, rays provide information about the nodes and bricks that were traversed during the ray casting process. Both caches are implemented in the same way, except that for the brick pool the cache elements are bricks and for the node pool the smallest managed entity is the node tile, grouping $2 \times 2 \times 2$ nodes (Section 3.4.1).

It is important to note that in this context, because of the cache mechanism, the structure updates (described Section 3.6.3) are performed lazily and only when necessary. At a given point in time, the octree leafs don't necessarily correspond to the nodes used during the rendering. For instance, it might be that the current tree encodes a very fine representation, but due to a distant point of view, the used data might solely be taken from higher node levels. The unused nodes, though currently unimportant, might need to be reactivated, hence, it makes sense to try to keep them in memory as long as no other data is needed. But if new data is needed, these are the first elements to be overwritten. In the following, we will explain how to establish such a mechanism.

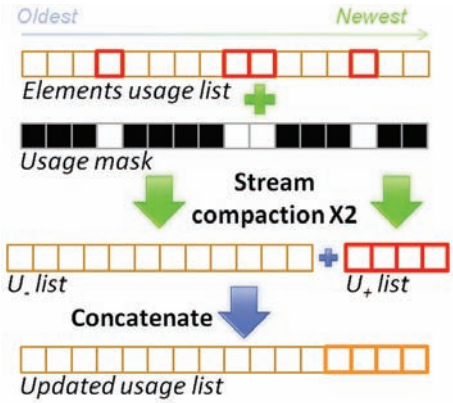


Figure 3.17. Illustration of the sorting procedure for the *element usage lists* based on stream compaction. The *usage list* is separated into a list of nodes used in the current pass and a list of nodes not used. The list of used nodes is then concatenated at the end of the list of unused nodes.

Per-ray data usage information. To keep track of the elements used by each ray, we associate a timestamp buffer to each of the managed pools (node pool and brick pool). This timestamp buffer "attaches" a 32-bit integer timestamp to each element (node tile or brick) of the pool.

During the ray traversal of the structure, this timestamp is renewed. Each time a ray uses a node, its associated timestamp set to the time of the current pass. In the same way for bricks, each time a brick is used during the traversal, its associated timestamp is updated.

Due to the parallel computation model of the GPU, many threads may write to the same timestamp buffer element at the same time. But since all rays will write the same value (the time of the current rendering pass), no atomic operations are needed and the approach remains efficient.

LRU management. The previously described timestamps are then used by the LRU mechanism to keep track of the oldest elements which are the candidates that might be overwritten with new data. In order to classify the elements, we maintain for both caches (node and brick) a list of element addresses sorted by usage called the *usage list*, a one-dimensional buffer of 32-bit values that contains as many entries as there are elements in the corresponding pool. Each entry contains the 30-bit address of its corresponding pool element.

In a usage list, the last used elements are kept at the end of the list, whereas the beginning contains those that were not used for a long time. When n new elements need to be inserted, the n elements corresponding to the first n entries of the list are recycled (Figure 3.17).

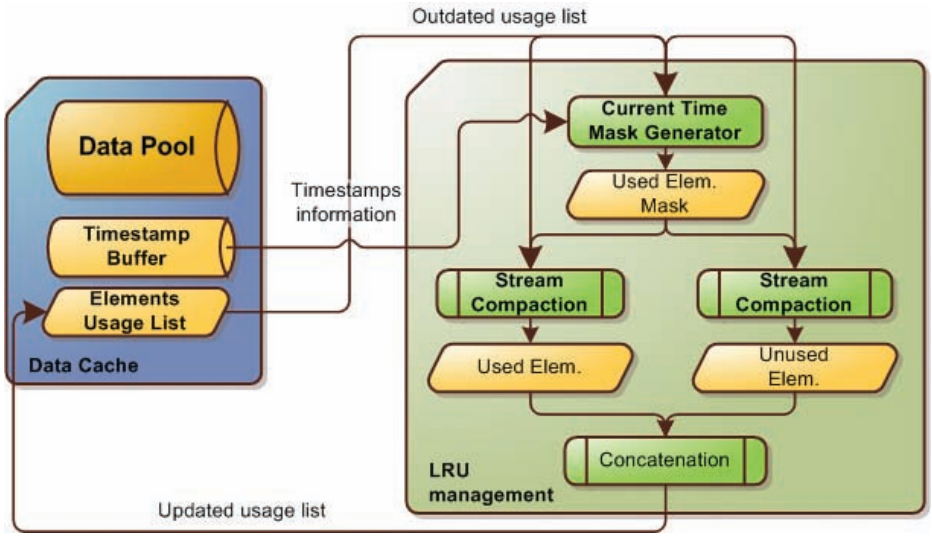


Figure 3.18. Update procedure of the caches usage list.

Maintaining usage list ordering. To maintain a usage list efficiently, we do not need to rely on sorting operations. Instead, we make use of two order-maintaining stream-compaction steps. This procedure is illustrated in Figures 3.18 and 3.17.

We first generate a *usage mask*. In practice, we use one of the remaining bits in the usage-list entry as a flag. This mask indicates, for each element of the usage list, if it was used in the current rendering pass. For this, we launch a kernel with one thread per usage-list element, and verify the corresponding timestamp. If the element was used in the current frame, its timestamp will match the current time step. The usage list then undergoes two stream compaction[3] steps to separate the used $\mathbf{U}_+$ from the unused elements $\mathbf{U}_-$ based on the usage mask. Because the stream compaction algorithm preserves the initial element order, the list of the unused elements will still have the oldest elements at the beginning and the most recently used in the end. Therefore, when concatenating $\mathbf{U}_+$ to the end of the $\mathbf{U}_-$, we obtain an updated and accurately sorted usage list.

LRU invalidation procedure. Update procedures will retrieve entries from the front of the usage list and, hence, retrieve information about where to overwrite older data. Unfortunately, this overwriting can introduce problems because elements in the data structure with pointers to this location will still assume that the old content is located at that address.

[3]We rely on the CUDA *Thrust* library to implement all stream compaction operations involved in the data management system.

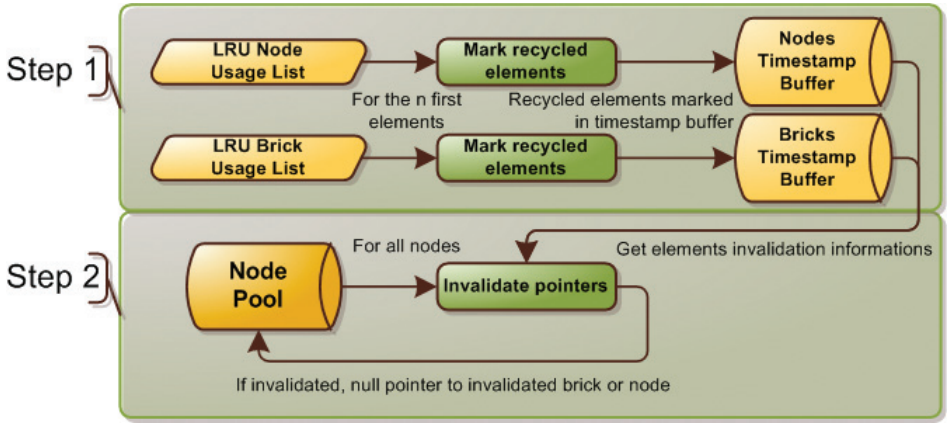


Figure 3.19. Two step pointers invalidation procedure.

In other words, each time cache elements are deleted in order to make room for new data, an *invalidation procedure* has to be executed to remove all the references to the data being recycled. More specifically, this means that when a *node tile* is deleted, all pointers in the octree that still reference this element, have to be invalidated (meaning *set to NULL*). The same care has to be applied when bricks are removed and all nodes pointing to such a brick should have their pointer invalidated.

This cleanup procedure is not straightforward because nodes and bricks could be referenced by more than just a single parent, as is needed in the case of recursive definitions, as well as instancing (Section 3.7).

Our solution to invalidate pointers reliably is to use a two step procedure illustrated in Figure 3.19 before overwriting elements. First, for both the node cache and the brick cache, we perform a kernel launch with one thread for each of the n recycled elements in the usage list (the n first elements). These threads set the timestamp buffer value of the according elements to a special value (in practice, we reserve zero for this purpose) using a memory scattering operation.

The second step then consists in testing every node in the *nodepool*. If it points to an element (a node or a brick) that was flagged with a zero timestamp its corresponding pointer is nulled. These steps are done in parallel for all nodes in the node pool using a kernel launch.

Once all pointers are invalidated, the elements to be recycled can be overwritten with new content, uploaded by the CPU. The usage list will automatically update itself in the next frame when the new elements are used.

3.6.3 Structure Update Requests Management

We have just seen how to decide about the importance of data elements by sorting them according to usage using an LRU scheme. Further, we discussed how to invalidate elements in order to enable overwriting. Now, we will discuss how the actual update mechanism is triggered and controlled (see Figure 3.20). First, we will talk about how we make it possible to use per-ray requests to determine which information should be uploaded to the GPU. We then discuss how to analyze these requests and trigger the CPU-side update.

3.6.4 Per-ray Requests

To enable per-ray node subdivisions and data load requests, we rely on a special *request buffer*. Its size is chosen to match exactly the number of nodes in the node pool, and it is arranged in the same way, so that there is a one-to-one correspondence between each node and the request buffer entry. Both can be accessed with the same address. Hence, we can store one request slot per node. In practice, this request slot is a 32-bit value. Two bits encode the type of the request. The first bit can indicate the need for a subdivision, the second for a data fetch. The other 30 bits store the node address. This latter information may seem redundant, as it could easily be derived from the address of the request slot, but it will facilitate the subsequent steps. We make the distinction between subdivision and data requests in order to avoid having to transfer all data after a subdivision. This could be wasteful because some child nodes might be invisible.

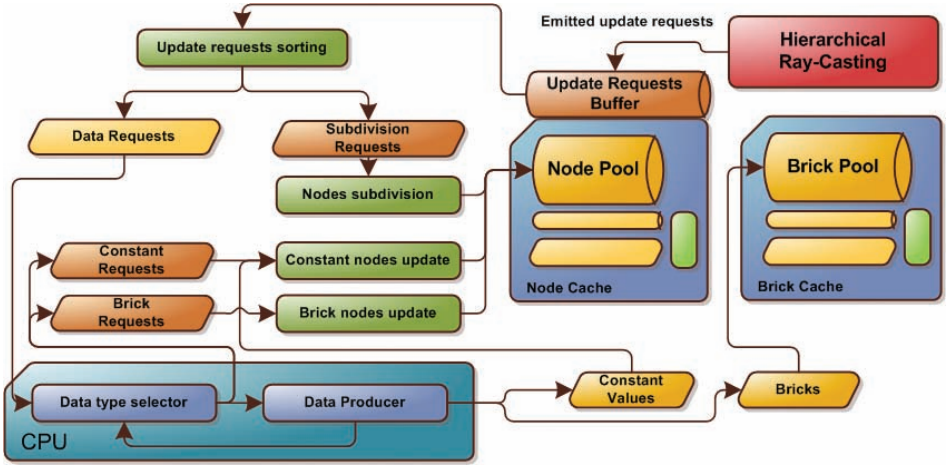


Figure 3.20. General structure updates flow with per-ray requests collection, request type sorting, data type retrieval through CPU producer and structure updates.

Request buffer elements are zeroed at the beginning of each render pass. During ray casting, each time a ray requests an update for a node, it fills the corresponding request buffer entry. While many rays with different needs (subdivision/data request/both) might ask for an update in parallel, and so access the same request slot concurrently, we want to avoid expensive atomic operations in global memory.

The first observation is that the 30 bits containing the node's address will always be the same, since it is always the slot address itself. So concurrent write will not be a problem for this part. A potential issue could only arise from the first two bits that encode the updates.

The different conflicts that can potentially appear are the following:

1. If the node is terminal, meaning that it cannot be refined, only data requests can be made. Whether a node is terminal is indicated by a flag (see Section 3.4.2).

2. If the node has data, obviously no ray would ask for data. Any update request is thus a subdivision request.

3. If the node has no data but children, it indicates that an update implies a data request.

4. If the node does not have data and no children, both kind of requests could be made.

Only the last case could lead to potential conflicts between rays. Two types of nodes fall into this category: uninitialized nodes (right after a subdivision), and nodes whose brick data was deleted (recycled in the cache). For both cases we force to always demand data. This allows us to detect constant and terminal nodes for which a refinement operation is impossible.

However, in theory, when rays ask for a *subdivision and data* or *data only*, only one of the two might prevail. Therefore a conflict is possible but we do not consider this situation as a problem. If only a data request flag is kept, instead of both, then the data is loaded, but no subdivision is performed. Data-only rays are satisfied, but not the subdivision rays. Fortunately, the latter will be in the next pass because, now that data is present, subsequent requests for this node can only be subdivision requests. This scheme does introduce a one-pass delay for the subdivision in the case a conflict actually occurs, but the treatment is much simpler and more efficient than other solutions.

3.6.5 Requests Type Sorting

After the ray traversal, the request buffer is filled. Now, we want to collect the needed update operations. For this, the request buffer is reduced on the GPU using a stream compaction algorithm keeping only non-zero elements (as illustrated Figure 3.21).

The resulting *compacted request list* is then split into a *data request list* and a *subdivision request list*, using two more stream reduction steps. Both lists contain the addresses in the node pool of those nodes that require to load data and to subdivide nodes, respectively.

Data requests are applied before dealing with subdivisions in order to ensure that, in case of both data and subdivision request (see previous section), a constant or terminal node will not be subdivided.

The length of these lists is an indicator for the memory that is needed to update the structure according to the requests. Whereas the subdivision requests can be treated on the GPU, data requests support from the CPU. Only the CPU is able to determine the constantness of a region, and to load new data from a mass storages (usually the hard disk).

In the following, we will discuss how both data and subdivision requests are handled.

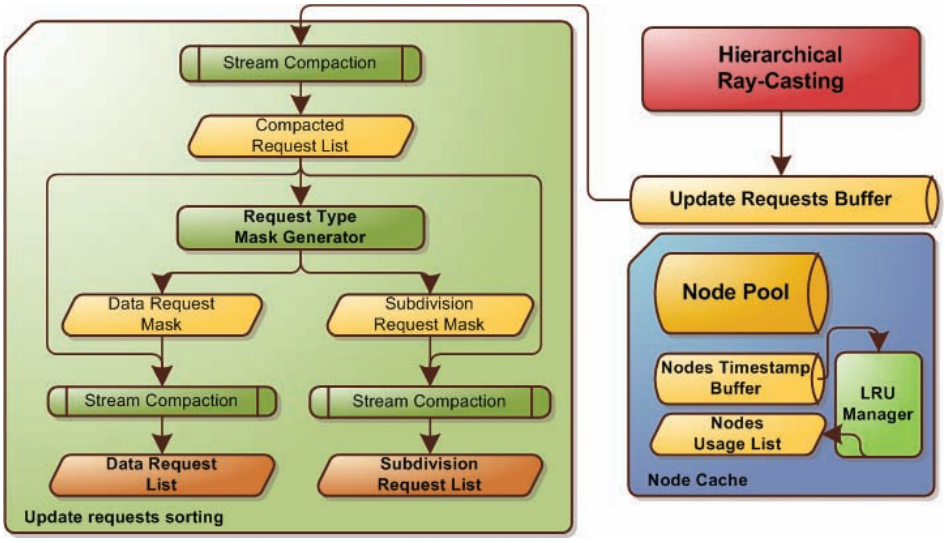


Figure 3.21. Request collection and sorting flow from the request buffer to the data request list and subdivision request list.

3.6.6 Dealing with Data Requests

Data requests correspond to missing data. This information has to be loaded or produced on the CPU side. Thus, the data request list needs to be sent to the CPU.

One important observation is that the CPU needs two kinds of information in order to update the data: *what* to load and *where* to store it. For the moment, let's concentrate on the question of what to load. What we need is information about the spatial extent of the node addressed in the data request. Only this localization information allows us to fetch the corresponding data from the disk. Unfortunately there is no relationship between the address of a node, which is contained in the data requests and the spatial extent it represents. In fact, due to the cache mechanism, the organization of the nodes in the node pool can be arbitrary and it has nothing to do with the actual scene.

To be able to provide the information about the spatial organization, we add another two arrays to our GPU memory:

First, to each node, we associate a code, which we call *localization code* that encodes the node's position in the octree and is stored in three times 10 bits, grouped in a single 32-bit integer. Each 10-bits represent one axis. Bit by bit, this series encodes a sequence of space subdivisions, so basically a descent in the octree. More precisely, the n^{th}bit of the first 10-bit value represents the child taken on the X axis at level n. Each bit represents the choice (left or right child) along this axis. This is similar to the descent in the octree from top-to-bottom, described in Section 3.5.3.

Second, each node will also store a *localization depth value* in form of an 8-bit integer. It encodes how deep in the tree the node is located. A localization depth of n means that only the first n bits of the localization code are needed to reach the node in the tree. Consequently, these two values describe exactly one node in an octree subdivision structure. This allows us to derive exactly what information needs to be loaded or produced on the CPU side.

In practice it is actually possible to reduce both array sizes by a factor of eight by storing these values per node tile (group of eight nodes) instead of per single node. This is possible because the CPU will also have access to the node's address. Due to the fact that nodes in a tile are grouped together in memory, the node address allows us to complete its localization code. We can derive the final bit that misses from the node's localization code with respect to the tile's localization code from the last bit of the node's address.

Of course, the values in these two arrays need to be updated or created when nodes are spawned. This is the case when a subdivision is applied and we will detail this simple process in Section 3.6.8, when explaining how to treat subdivision requests.

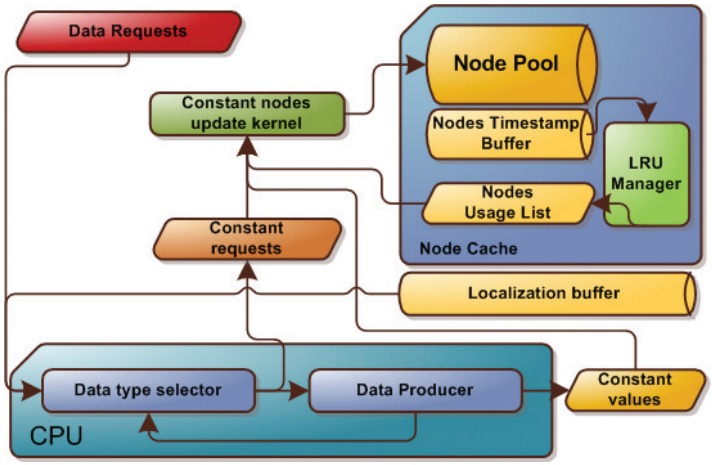


Figure 3.22. Constant nodes update execution flow. Constant information is provided by the CPU data producer.

3.6.7 Separating Constant from Brick Nodes

Now, that we have seen how the CPU determines what data needs to be loaded, let us look at the actual update process. On the CPU side, each request is passed to a data provider managing an on-disk data representation of the volume. The data provider determines, for each element of the data request list, whether the corresponding voxel region is constant or requires a brick. Accordingly, the data-request list is split into a list of nodes requiring a constant value $\mathbf{L}_c$ and a list of nodes requiring bricks $\mathbf{L}_b$. Both types are handled independently.

Updating constant nodes. Constant nodes are simple to treat because the necessary data field to store the value is already present in the node pool (see Figure 3.5). To make the update efficient, nodes in the octree structure are updated in parallel using a single kernel launch on all entries of $\mathbf{L}_c$ and getting data from a list of all corresponding constant values (see Figure 3.22).

Updating brick nodes. More work is needed for the brick nodes in $\mathbf{L}_b$. Bricks are provided one by one by the CPU data provider (see Figure 3.23). Each new brick emplacement in the brick pool is provided via the cache mechanism, recycling the oldest bricks slots (Section 3.6.2).

We have seen how the GPU-side cache maintains a usage list of all bricks. Its first entries contain the addresses of those elements that should be overwritten first. To determine where to upload the n new bricks, the CPU simply reads back

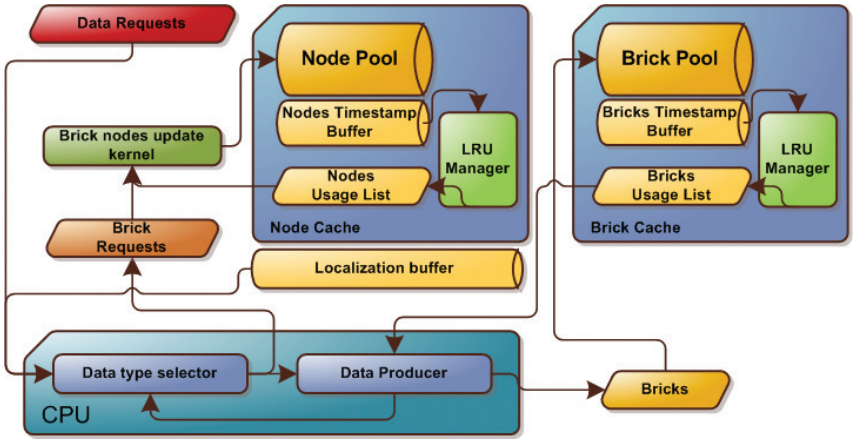


Figure 3.23. Brick update execution flow. Bricks data is provided by the CPU data producer.

the n first entries of the usage list. Bricks are then uploaded to these addresses in the brick pool.

The next step is to invalidate all pointers that refer to the overwritten elements. We have seen this mechanism in Section 3.6.2. The final step is to update the nodes themselves. Their pointers must be corrected to reference the new bricks. This is done in parallel by launching an update kernel with a thread for each element in $\mathbf{L}_b$. The new brick pointers are read from the n first elements of the bricks' usage list.

To gain some performance, we do not perform these last steps right away because other invalid pointers might arise from the subdivision requests. So we only use a single correction step to invalidate brick pointers, as well as child pointers, once the subdivisions were applied. We will discuss these next.

3.6.8 Dealing with Subdivision Requests

Contrary to data requests, subdivision requests do not require any CPU intervention and can be addressed directly on the GPU. In principle, the update mechanism is somewhat similar to the solution for bricks. If n subdivision requests occurred, we start a kernel with n threads. Thread i fetches the ith subdivision request and the ith entry of the node usage list. The subdivision request contains the address of the node to subdivide. Consequently, we update its pointer to the address fetched from the node usage list, at which location we will store our new node.

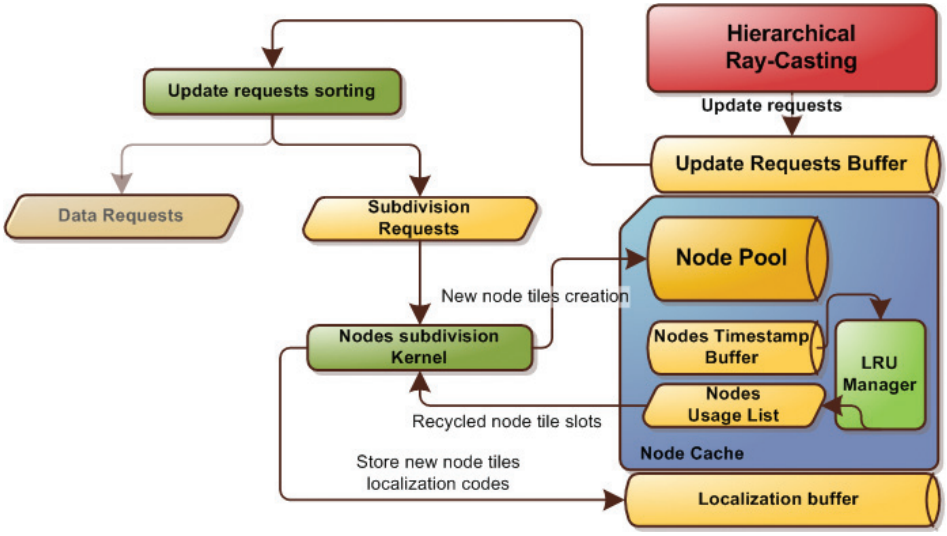


Figure 3.24. Subdivision requests execution on the GPU.

Finally, all pointers to the overwritten elements need to be invalidated. We do this in parallel with the brick pointer invalidation of overwritten elements, as described in Section 3.6.2. The final element to talk about is the localization code and localization depth of node tiles. These values were needed by the CPU to be able to associate to each node a corresponding region in space and, hence, the corresponding data to be uploaded. In order to initialize these values for the child node tile, we proceed as follows. We use the address of the node N that needs a subdivision. N's address can be truncated in order to find the address of the node tile $\mathbf{N}$ that contains N. This address allows us to retrieve N's localization code and localization depth. The new tile depth is obtained by incrementing the old depth by one. The new localization code can be found, based on the same principle we used to reduce the memory requirements of the localization code by a factor of eight in the last section. In other words, we make use of N's address itself to discover which child of $\mathbf{N}$ N corresponds to. Accordingly, this position indicates the missing bit in the localization code for the new node tile (see Figure 3.24).

3.6.9 Implementation Summary

Figure 3.25 presents a summary of all permanent GPU memory regions used by our data management mechanism. Presented memory occupancies correspond to the ones used in most of our example scenes and gives an idea of the relative sizes

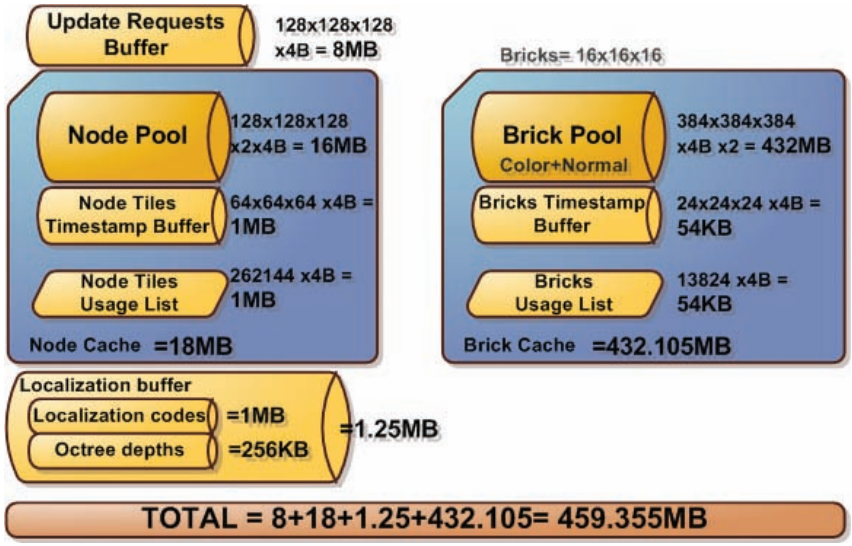


Figure 3.25. Summary of the memory requirement of every GPU memory regions used for a typical usage case.

of the memory regions. Unsurprisingly, the brick pool appears to be the largest buffer of our implementation. The node pool may seem oversized compared to the brick pool, but it allows us to better cache the subdivision structure which leads to a performance increase. Finally, all regions used by the data management algorithm are relatively small.

3.7 Octree-Based Synthesis

Our pointer-based octree structure allows us to produce many interesting scenarios.

A first feature is the ability to implement instancing of interior branches, as well as recursions. We can reuse subtrees by making nodes share common subnodes. This can be very advantageous if a model has repetitive structures and can significantly reduce the necessary memory consumption. This kind of octree instancing is illustrated Figure 3.26.

The node pointers further allow us to create recursions in the graph. This is particularly interesting in the context of fractal-like representations. The self-similarity is naturally handled and the resulting volumes are virtually of an infinite resolution. Figure 3.27 shows an example of a Sierpinski sponge fractal. It is implemented using the generalization of our octree, an N^3-tree node with $N = 3$ (see Section 3.4.2).

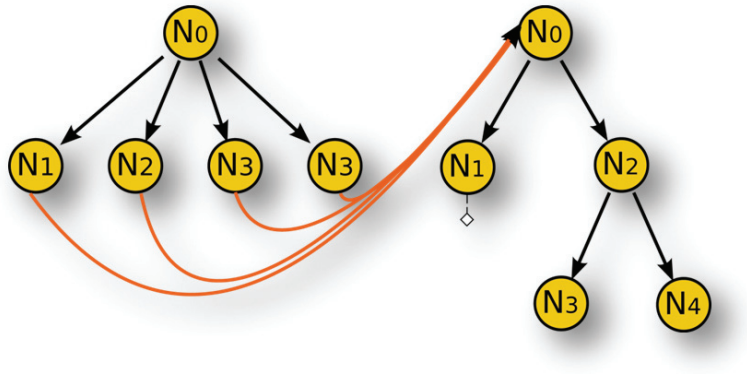


Figure 3.26. Example graph of an octree using instancing.

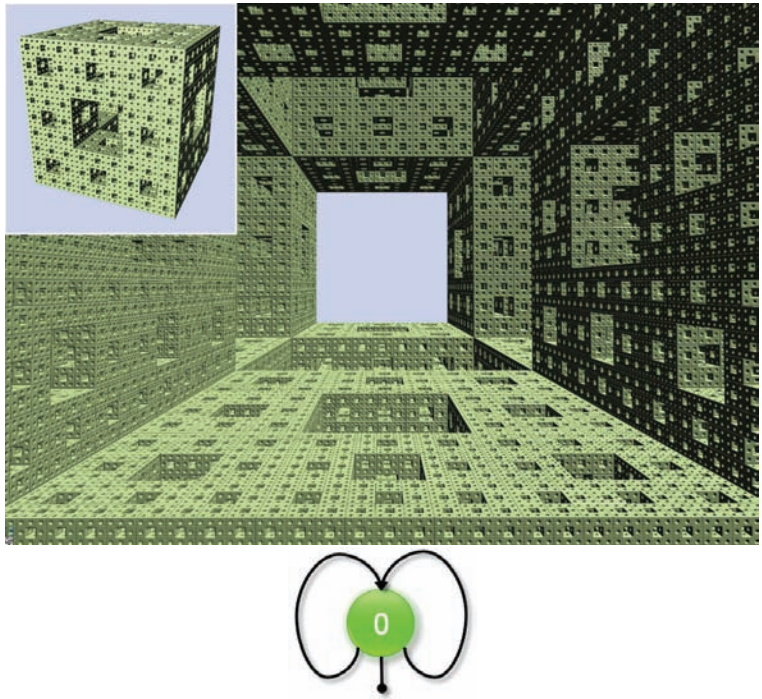


Figure 3.27. Example of a Sierpinski sponge fractal fully implemented with recursivity in the octree. This example is running at around 70FPS. The bottom graph shows the only recursively linked node used in this case.

3.8 Voxel Object Instancing

Our ray-tracing framework is compatible with several voxel entities present in the scene. In such scenario, when different objects are mixed, multiple GigaVoxels octrees are handled in the node pool and managed by the cache mechanism.

For each object, a bounding volume is drawn. A fragment shader outputs ray origins on the bounding volume, as well as the direction of the corresponding view rays. These rays are used to initialize the ray casting process. Scaling, rotation and LODs are automatically handled and allow us to represent scenes with many complex objects at high frame rates. Figure 3.28 shows an example of such a scene.

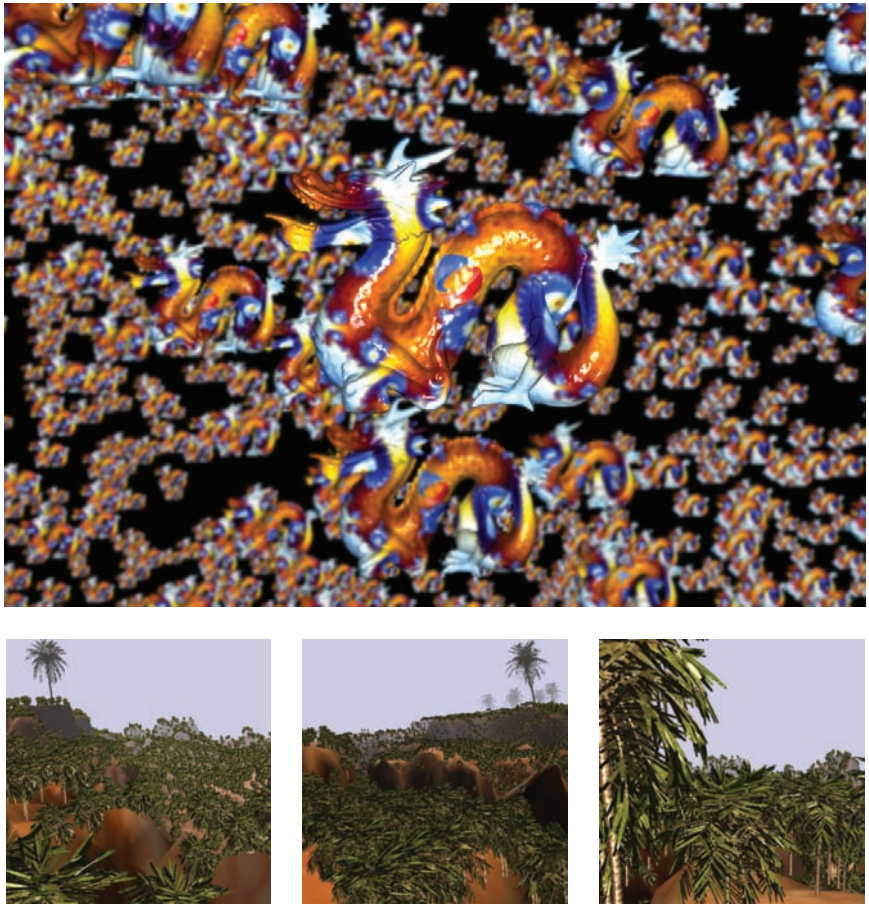


Figure 3.28. Examples of free instancing of GigaVoxels objects in space.

3.9 MipMap-Based Blur Effects

Throughout this chapter we have shown that our hierarchical structure is the key element to enable the treatment of the large data volumes. We further pointed out in Section 3.5.4 that the way we traverse this structure can be related to an approximative cone tracing. Cone tracing has many interesting applications and even though our solution is approximate, it enables us to benefit from these possibilities.

First we will explain how to integrate shadow computations in our application. Then we will illustrate how the same algorithms can be used to produce approximate soft shadows. Finally we will show how a similar scheme can be used in the context of depth of field rendering. Interestingly, even though this process is very challenging for triangular models, volume rendering benefits from the out-of-focus effect; the stronger the lens blur, the more performance is gained.

3.9.1 Soft Shadows

Shadows are an important cue that help us to evaluate scene configurations and spatial relations. Further, it is a key element to make images look realistic. So far, this point has not been addressed in our current pipeline. Here, we will explain how our rendering engine can be used to obtain convincing shadow effects for gigantic volumetric models (see Figure 3.29).

Before tackling soft shadows, let us take a look at the case of a point light source. Given a surface point P in our volumetric scene, or volumetric model, we want to determine how much light reaches P. This basically remounts to shooting a ray from P towards the light source. If the opacity value of the ray saturates on

Figure 3.29. Example of soft shadows rendered by launching secondary rays and using the volume mipmapping mechanism to approximate integration over the light source surface. Interestingly, the blurrier the shadows, the cheaper it is to compute.

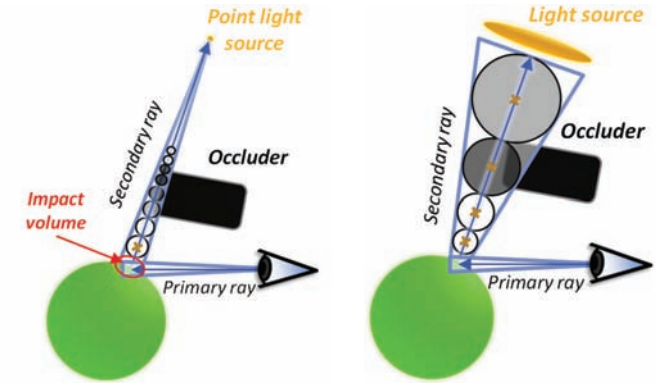


Figure 3.30. Illustration of shadow computation for a point light source, taking impact volume into account (left). Soft shadows computation for a surface light source (right).

the way to the light, the P lies in shadow. If the ray traversed semi-transparent materials without saturating, the accumulated opacity value gives us the intensity of the soft shadow. It should be pointed out that this traversal can be used to also accumulate colors to naturally handle colored shadows.

In practice, we do not really have an impact *point*. Rather, due to our traversal inspired by cone tracing, we obtain an *impact volume* at the intersection between the cone and the object. To take this into account, we should not shoot a simple ray, but again an entire cone. This *light cone*'s apex will lie on the light source itself and its radius is defined by the size of the impact volume (see Figure 3.30).

To sample the light cone, we perform a similar traversal as for the view. Only this time, the resolution is defined by the light cone instead of the pixel cone radius. During this traversal we accumulate the opacity values. Once the value saturates, the ray traversal can be stopped.

To approximate soft shadows, we compute a shadow value at the impact volume V. If V were a point, a cone instead of a simple shadow ray would need to be tested for intersection. This cone would be defined by the light source and the impact point. Consequently, a possible approximation for an impact volume V is to define a cone that contains not only the light, but also V (see Figure 3.30). Again, we accumulate the volume values. The resulting value reflects how much the light is occluded.

This very coarse approximation is extremely efficient, delivers acceptable shadows and is fully compatible with the cache mechanism presented in Section 3.6.

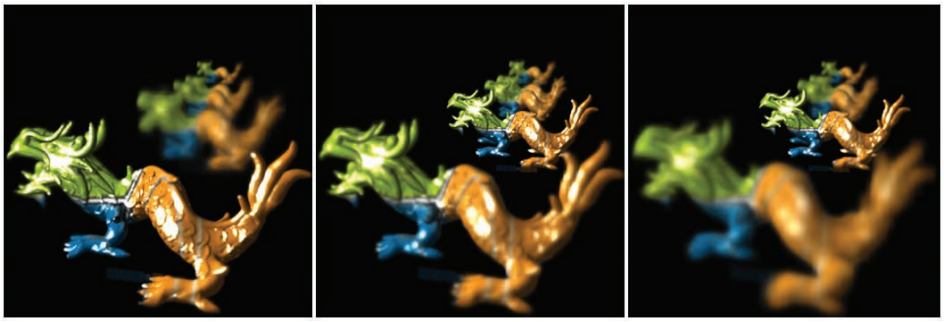


Figure 3.31. Example of depth of field rendering with GigaVoxels thanks to the volume mipmapping. Once again, the blurrier the objects, the cheaper it is to render.

3.9.2 Depth of Field

Another very important element for realistic images is the depth-of-field lens blur, present in any camera, as well as our own optical system. It results from the fact that the aperture of a real pinhole camera is actually finite. Consequently, unlike standard openGL/DirectX rendering, each image point reflects a set of rays, passing through the aperture and lens. The lens can only focus this set of rays on a single point for elements situated on the focal plane. As illustrated in Figure 3.31 this set of rays can again be grouped in some form of double cone, the *lens cone*. This lens cone defines the resolution that should be used during the traversal in order to approximate this integral over the camera lens.

Paradoxically, the more blur is introduced, the faster the rendering becomes, and the less memory is necessary to represent the scene. This is very different for triangle-based solutions, where depth of field, and even approximations, are extremely costly processes. In games, depth of field is usually performed as a

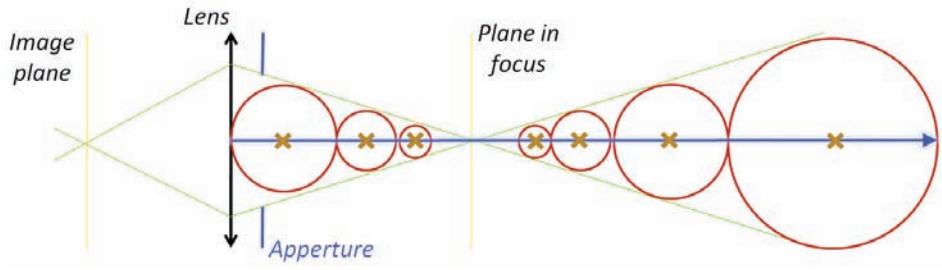


Figure 3.32. Illustration of the cone tracing used to approximate depth of field effect with a single ray.

post-process by filtering the resulting image with spatially varying kernels. One problem of such a filtering process is the lack of hidden geometry. In our volumetric representation, *hidden* geometry is integrated as much as it is necessary to produce the final image (see Figure 3.32). The algorithm does not need to be adapted to consider the different reasons for why volume information is needed. Disocclusion due to depth of field, transparency and shadows are all handled in the same manner. In fact, any kind of secondary ray is supported, showing the versatility of our framework.

3.10 Conclusion

In this chapter, we have seen how to efficiently deal with the main problem of voxel representations: huge memory consumption. The data sets we work with exceed the memory of current GPUs by large amounts. Our efficient strategies to adapt the volume resolution according to the point of view enable us to completely overcome the memory limitation by transferring data only when necessary and applying updates to the structure solely where needed.

The presented approach underlines that voxels have many interesting properties. We showed how to reduce aliasing, represent transparent materials, and allow recursive or instanced scene definitions. Furthermore, our algorithm inherently implements several acceleration methods that usually have to be addressed with particular routines and strategies: Frustum culling, visibility testing, LOD selection, temporal coherence, and refinement strategies. They are all integrated in the same framework: our per-ray queries. Consequently, the code to treat complex geometry becomes much simpler and easier to maintain, which is important for practical implementations.

Our GigaVoxels approach can, in some scenarios, be more efficient than their triangular mesh counterpart. We showed how to produce approximate shadow and depth of field effects that appear more convincing than many standard approaches applied for computer games based on image filtering techniques. Interestingly, in contrast to these triangle-based solutions, effects like depth of field or approximate soft shadows decrease the computational workload when compared to a pinhole image or hard shadows. Thus resulting in more realism for less computational power.

All these advantages hint at a more extensive future use of volumetric representations in real-time applications, such as games. Our GigaVoxel engine is an efficient solution in this context and we believe it will open up the road to many interesting special effects, of which we have currently only scratched the surface.

Bibliography

[Crassin et al. 09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. "GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering." In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*. ACM, 2009. Available online (http://artis.imag.fr/Publications/2009/CNLE09).

[Engel et al. 01] Klaus Engel, Martin Kraus, and Thomas Ertl. "High-Quality Pre-integrated Volume Rendering Using Hardware-Accelerated Pixel Shading." In *ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware (HWWS)*, pp. 9–16, 2001.

[Hadwiger et al. 06] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-Time Volume Graphics*, 2006.

[Horn et al. 07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. "Interactive k-d Tree GPU Raytracing." In *ACM Siggraph Symposium on Interactive 3D Graphics and Games (I3D)*, 2007.

[Kapler 03] Alan Kapler. "Avalanche! Snowy FX for XXX." *SIGGRAPH Sketch* (2003): 1.

[Krall and Harrington 05] Joshua Krall and Cody Harrington. "Modeling and Rendering of Clouds on 'Stealth.' " *SIGGRAPH Sketch* (2005).

[Kruger and Westermann 03] J. Kruger and R. Westermann. "Acceleration Techniques for GPU-based Volume Rendering." In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*. Washington, DC: IEEE Computer Society, 2003.

[Lefebvre et al. 05] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Chapter Octree Textures on the GPU, pp. 595–613. Boston: Addison Wesley, 2005. Available online (http://www-evasion.imag.fr/Publications/2005/LHN05a).

[NVIDIA 09] NVIDIA. *CUDA Programming Guide 2.2*, 2009.

[Roettger et al. 03] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. "Smart Hardware-Accelerated Volume Rendering." In *VISSYM '03: Proceedings of the Symposium on Data Visualisation 2003*. Eurographics Association, 2003.

[Scharsach 05] Henning Scharsach. "Advanced GPU Raycasting." In *Central European Seminar on Computer Graphics*, pp. 69–76, 2005.