UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE
Spécialité : **Mathématiques Informatique**

Arrêté ministériel : 6 Aout 2006

Présentée par

Cyril Crassin

Thèse dirigée par **Fabrice Neyret**

préparée au sein du **Laboratoire Jean Kuntzmann**
dans **l'école doctorale de Mathématiques, Sciences et Technologies de l'Information, Informatique**

GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes

Thèse soutenue publiquement le **12 Juillet 2011**,
devant le jury composé de :

Acknowledgements

Je tiens tout d'abord à remercier Fabrice pour son encadrement tout au long de ces 4 années de thèse qui furent extrêmement enrichissantes, tant d'un point de vue scientifique que personnel. J'ai énormément apprécié sa disponibilité, son regard toujours objectif et pertinent, ainsi que son implication, sa passion et son admirable vision de la recherche.

Je remercie également tous les anciens et les nouveaux membres de l'équipe ARTIS pour leur bonne humeur et la formidable ambiance que tous contribuent à créer et à perpétuer. Un merci particulier à mes co-bureaux et amis, Pierre et Olivier, pour avoir supporté mon bazar et mes mauvais jeux de mots pendant plus de 2 ans. Un grand merci également à Elmar pour toute l'aide qu'il m'a apportée durant cette thèse, ainsi qu'à Eric et à Charles pour leur appui mathématique précieux. Je n'oublie pas non plus EVASION qui fut mon équipe « d'origine » et dans laquelle j'ai également passé d'excellents moments. Puisse « l'esprit d'iMAGIS » continuer à vivre encore longtemps !

Je tiens également à remercier Mathias Paulin et Michael Wimmer d'avoir accepté de rapporter ma thèse malgré les délais assez courts que j'ai dû leur imposer. Un grand merci également à Jean-Michel Dischler d'avoir pris part à mon jury en tant qu'examinateur ainsi qu'à Francois Sillion d'avoir accepté de présider ce jury. I would like to thank also Carsten Dachsbacher for being part of my jury, as well as Miguel Sainz for his trust and the great opportunity he gave me to work for a company I have always dreamed to work for. Un grand merci également à Emmanuel Melin qui a cru en moi depuis tant d'années, et sans qui je n'en serais sans doute pas où j'en suis aujourd'hui, ainsi qu'à Sylvain Jubertie pour son amitié, son soutien et pour m'avoir donné le goût de la recherche. Je tiens également à remercier Sarah Janvier Lewis pour le travail extraordinaire de relecture et correction qu'elle effectué sur mon manuscrit.

Pour finir, je voudrais remercier tous mes amis pour le soutient et la bonne humeur qu'ils ont pu m'apporter. Un merci particulier à Adeline et à Maxime, mes camarades de galère pendant 5 ans, pour tous les moments qu'on a pu partager. Je remercie également ma famille pour son soutient tout au long de mes études. Enfin, je remercie bien sûr Marlène pour tout son soutien, sa patience (il lui en a fallu), son courage (il lui en a fallu également) et sa compréhension durant ces trois années.

Abstract

In this thesis, we present a new approach to efficiently render large scenes and detailed objects in real-time. Our approach is based on a new volumetric pre-filtered geometry representation and an associated voxel-based approximate cone tracing that allows an accurate and high performance rendering with high quality filtering of highly detailed geometry. In order to bring this voxel representation as a standard real-time rendering primitive, we propose a new GPU-based approach designed to entirely scale to the rendering of very large volumetric datasets.

Our system achieves real-time rendering performance for several billion voxels. 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.

Our solution is based on an adaptive hierarchical data representation depending on the current view and occlusion information, coupled to an efficient ray-casting rendering algorithm. We introduce a new GPU cache mechanism providing a very efficient paging of data in video memory and implemented as a very efficient data-parallel process. This cache is coupled with a data production pipeline able to dynamically load or produce voxel data directly on the GPU. One key element of our method is to guide data production and caching in video memory directly based on data requests and usage information emitted directly during rendering.

We demonstrate our approach with several applications. We also show how our pre-filtered geometry model and approximate cone tracing can be used to very efficiently achieve blurry effects and real-time indirect lighting.

Publications

The work presented in this thesis appeared in the following articles, proceedings, books, invited talks and posters.

Book chapters

[CNSE10] Cyril Crassin, Fabrice Neyret, Miguel Sainz, and Elmar Eisemann. *Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In book: GPU Pro*, chapter X.3, pages 643–676. A K Peters, 2010.

International conferences and journals

[BNM+08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *ACM Symposium on Interactive 3D Graphics and Games (I3D)*, 2008.

[CNLE09] 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, feb 2009.

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *[ONGOING SUBMISSION] Computer Graphics Forum (Pacific Graphics 2011)*, September 2011.

National conferences

[BNM+07] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Rendu interactif de nuages réalistes. In *AFIG '07 (Actes des 20èmes journées de l'AFIG)*, pages 183–195. AFIG, November 2007.

Talks ans posters

[CNL+09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, Miguel Sainz, and Elmar Eisemann. Beyond triangles : Gigavoxels effects in video games. In *SIGGRAPH 2009 : Technical talk + Poster (Best poster award finalist)*. ACM SIGGRAPH, August 2009.

[CNS+11a] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing: A preview. Poster ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D). Best poster award., feb 2011.

[CNS+11b] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing: An insight. In *SIGGRAPH 2011 : Technical talk*. ACM SIGGRAPH, August 2011.

Industry invited talks

[CNE09] Cyril Crassin, Fabrice Neyret, and Elmar Eisemann. Building with bricks: Cuda-based gigavoxel rendering. In *Intel Visual Computing Research Conference*, 2009.

[Cra09] Cyril Crassin. Gigavoxels: Voxels come into play. Crytek Conference Talk, nov 2009.

Contents

Chapter 1

Introduction and motivation

1.1 Motivation: Real-time rendering of large and complex scenes

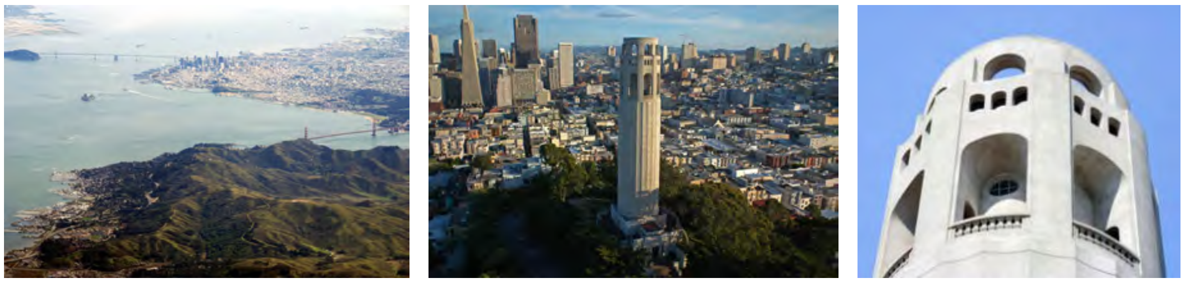


Figure 1.1. Illustration of a multiscale scene. *Sources: (Left-to-right) San-Francisco by arcortvriend , San-Francisco from www.staysf.com, Coit-tower from http://wsisf.com*

Photorealism has always been a major goal for computer graphics (CG). Today, one of the key problems is the management of details. Rendering large multi-scale scenes (such as the one presented in figure 1.1) and very detailed objects (Figs. 1.2 and 1.3) accurately and efficiently is now one of the most challenging problems in computer graphics, not only for real-time applications, but also for CG featured films. Usual rendering methods tend to be very inefficient for highly complex scenes because rendering cost is proportional to the number of primitives. In such scenes, many geometrical details have to be accounted for in each single pixel. The small size of primitives creates high-frequency signals that lead to aliasing artifacts which are costly to avoid. In order to produce high quality rendering, all lighting contributions of these details must be integrated in order to compute the final color of a given pixel. This integration poses two kinds of problems: how to do it efficiently (in terms of performance), and how to do it accurately (in terms of quality).

In this section, we will show how current rendering approaches reach their limits both in terms of quality and performance when dealing with very complex and multi-scale scenes. In this context, scalability is an absolute necessity. We will see that massive per-pixel supersampling is not affordable for real-time applications, and does not scale to an increasing number of details to be integrated per-pixel. Thus, geometrical simplification approaches have to be used, in order to deal with large amounts of geometrical data, and to prevent aliasing problems. But such simplifications inevitably remove details that would have contributed to the final image, and fails to preserve important shading effects (eg. roughness). The challenge in this context is to be able to keep shading details while allowing interactive rendering and maintaining a reasonable memory consumption. We will demonstrate how some form of geometrical pre-filtering is required to tackle this problem, and explain how we propose to rely on a volumetric enriched representation to handle it.

1.1.1 Limits of current mesh-based rendering approaches

All current rendering models used today to display geometry are based on a triangle or quad mesh representation. With this representation, objects are modeled by their surface, and are decomposed into simple primitives for rendering. Rendering is a view-based sampling operation. The idea is to sample a 3D scene in order to reconstruct a 2D image. The 3D scene represents the continuous source signal, while the 2D image is the signal we want to reconstruct. According to the Nyquist-Shannon sampling theorem [Nyq28], sampling frequency needs to be at least twice the highest signal frequency in order to reconstruct it correctly. Sampling below this Nyquist limit leads to so-called "aliasing". Conceptually, this means that we need no less than one sample for each peak and another for each valley of the original signal. That means that for each surface primitive to render, at least two samples need to be calculated in order to accurately reconstruct its contribution to the final image. Thus, increasing the geometrical complexity of a scene necessarily means increasing the sampling density used to compute the rendered image in the same proportions, in order to ensure a high quality reconstruction.

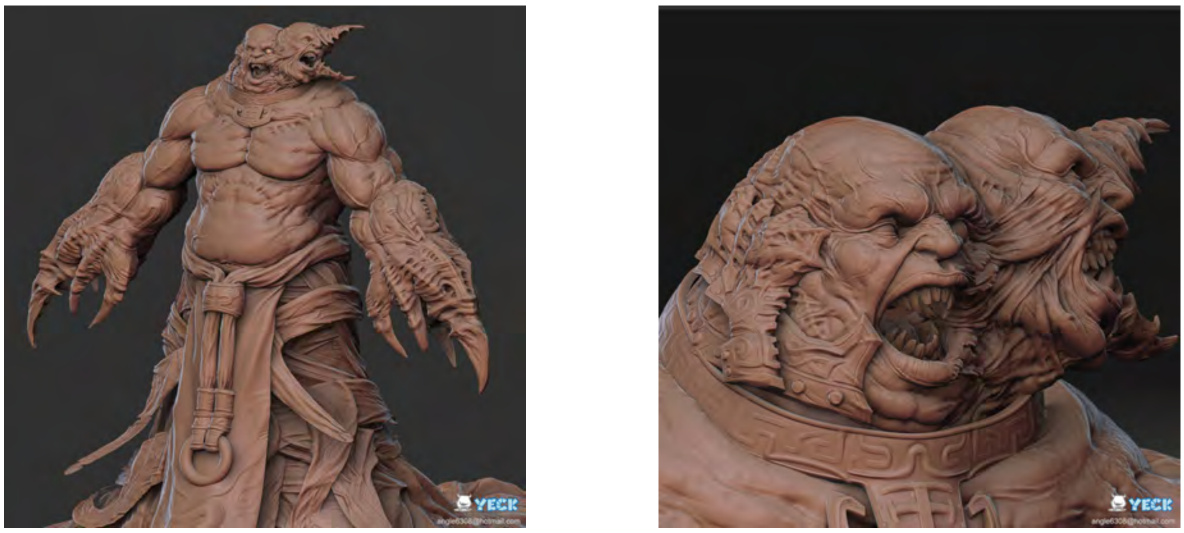


Figure 1.2. Example of an highly detailed mesh modelized using ZBrush [Spe08]. Image courtesy of Yeck.

The problem is that the amount of available processing power does not scale as fast as the need for more and more complex objects, and even if it did, increasing the computation proportionally to the complexity of a scene do not seems to be a scalable or sustainable solution. Similarly, the amount of memory available for rendering is a limited resource and storing and quickly accessing arbitrary large and complex scenes is a challenging problem.

The primary challenge in computer graphics has always been to bit these constraints in order to produce a more and more detailed rendering, without increasing the processing time and needed storage in the same proportions. Real-time rendering of highly complex geometrical scenes poses two kinds of problems. First, a quality problem: how to accurately integrate the shading contributions of all details, in order to prevent aliasing and capture expected lighting effects. Second, a performance problem: how to efficiently deal with large amounts of geometrical data, both in terms of computation and storage.

Classical mesh+texture representation

The classical way to deal with detail rendering is to split geometry representation into a coarse triangle-based surface mesh representation and fine scale surface details, reflectance and illumination properties, specified with 2D texture maps [Cat74] and used as parameters in a local illumination model. When objects can be seen from a relatively limited range of distances, this representation makes it possible to use a unique coarse mesh and to pre-filter detail maps linearly and separately. The idea is to estimate the averaged outgoing radiance from a surface to a pixel by applying the local illumination model on the averaged surface parameters from the map, instead of averaging the result of this application on fine-grained details. To do so, the classical approach is to rely on MIP-mapping [Wil83], with multiple map resolutions pre-computed before rendering. Such texture filtering does not prevent aliasing to appear at the boundary of meshes and on the silhouette of objects. This issue is usually addressed using a multi-sampling technique [Ake93], estimating per-pixel triangle visibility at higher frequency (>1 sample per pixel) than the shading itself.

This classical approach works well for moderate range of view distances, and low-detail meshes, when triangles cover at least one pixel on the screen and silhouettes do not contain thin details. But when highly detailed meshes viewed from a wide range of distances need to be rendered, many mesh triangles project to the same screen pixel and simply relying on the pre-filtering of the maps and a limited multisampling becomes insufficient to integrate all surface details, and leads to aliasing artifacts.

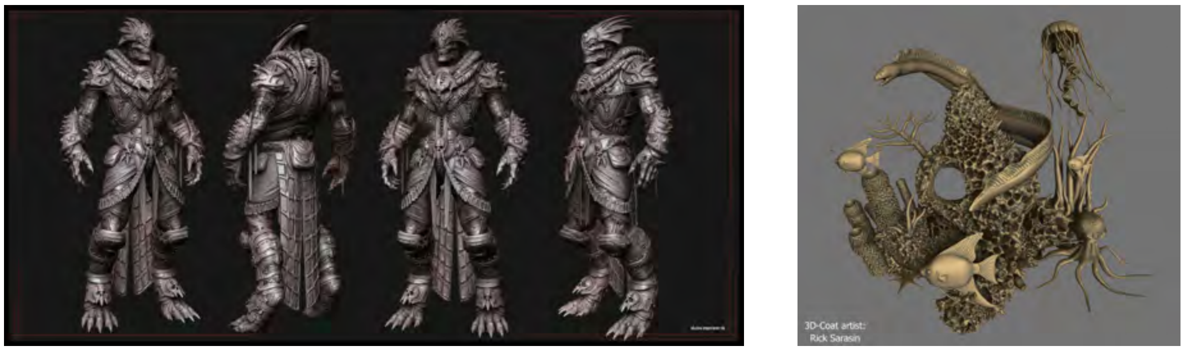


Figure 1.3. *Left:* Example of an highly detailed character mesh modelized using Z-Brush. *Right:* Very complex mesh modelized using 3D-Coat, a voxel-based 3D sculpting tool. Images courtesy of Benerdt.de (left) and Rick Sarasin (right)

Adaptive supersampling

The classical way to integrate per-pixel geometrical details is to rely on supersampling [Whi80, CPC84]. For each pixel, a weighted average of the illumination coming from all meshes elements and falling into that pixel is computed, i.e., an integral of the contributions of all these elements, which can be extremely costly. The outgoing radiance from a surface is given by a local illumination model as a function of the incident radiance and the surface properties. Computing this integral numerically during rendering (using massive sampling) can be extremely costly and many adaptive and stochastic multi-sampling methods have been proposed to speed-up its computation by optimizing the placement and distribution of samples. Adaptive supersampling allows us to push back slightly the limit of complexity that can be handled per-pixel, but still does not scale with an arbitrary increase of geometrical complexity. In addition, the cost of such supersampling of the geometry is usually not affordable for real-time application.

Mesh simplification and geometric LOD

In such situations, the approach usually employed to deal with aliasing problems and reduce rendering time is to simplify the geometry itself, in order to adapt it to the rendering resolution. This is done by relying on geometric level-of-details (LOD) approaches that progressively remove mesh details. The idea is to maintain a constant number of primitives to be rendered per screen pixel, whatever the viewing distance and complexity of the original mesh.

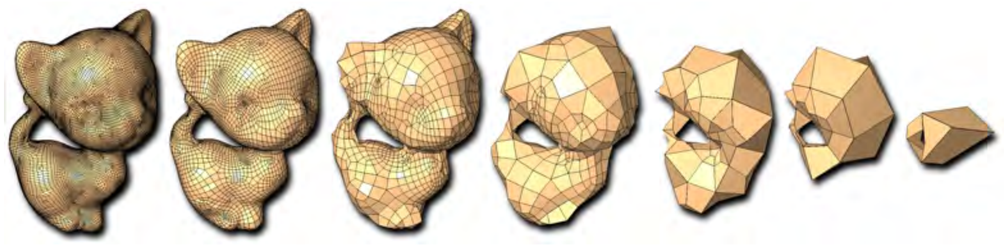


Figure 1.4. Illustration of Quadrilateral mesh simplification [DSSC08]

Mesh simplification is a tempting solution to deal with the geometry aliasing problem and is a way for pre-filtering geometry, ie. eliminating high frequency details. The main problem is then to find accurate pre-filtering methods that preserve appearance and light interaction. Many automatic methods have been proposed in order to automatically simplify triangle meshes and to provide a smooth transition between levels-of-details [Hop96, DSSC08]. However, automatic mesh simplification approaches are relatively difficult to control. Therefore in situations such as video game authoring, simplified models have to be optimized manually. Most of the time in video-games, a set of simplified versions of the meshes are pre-generated manually by artists, and the appropriate resolution is chosen at runtime. This approach prevents us from using a progressive transition and leads to popping artifacts.

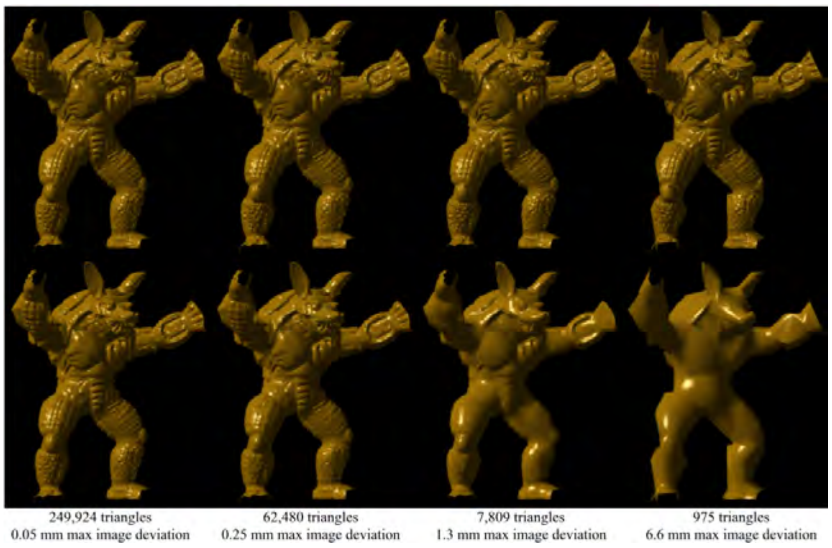

249,924 triangles
0.05 mm max image deviation
62,480 triangles
0.25 mm max image deviation
7,809 triangles
1.3 mm max image deviation
975 triangles
6.6 mm max image deviation

Figure 1.5. Geometry simplification (top) with automatic details injection into normal maps (bottom). *Source: [COM98]*

Details injection inside textures Mesh simplification inevitably removes details that would have contributed to the final image. As illustrated in figure 1.5, some details removed between two simplified versions of a mesh can be injected inside a detail map, in order to preserve their contributions inside the projected geometry. Methods have been proposed to allow automatic detail injection inside textures [COM98]. Such methods lack flexibility and in practice, details textures are usually created manually in order to be fully controlled by the artist. Moreover, details can not be preserved on the silhouette of objects.

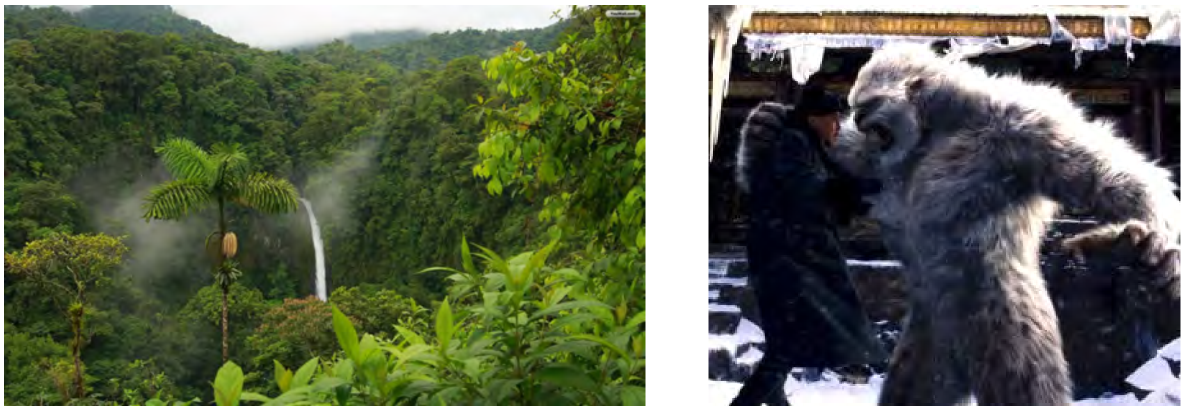


Figure 1.6. *Left:* Illustration of a large-range view of a forest. *Source: YouWall.com. Right:* Illustration of fur rendering in off-line special effects, *Source: The Mummy 3 (Digital Domain/Rhythm&Hues)*

Fundamental problem Surface-based simplification approaches perform best for finely tessellated smooth manifolds that are topologically simple, or for relatively low simplification. For complex surfaces or high levels of simplification, they fail to capture the surface details needed to reconstruct a correct appearance and shading (cf. figure 1.4). Moreover, mesh simplification approaches often require mesh connectivity. This is due to the nature of the simplification primitives (opaque triangles), that makes it difficult to treat complex and disconnected objects. Thus, the quality of simplified models becomes unacceptable for extreme simplifications and when multiple objects need to be merged.

The fundamental problem is that the light interaction occurring on a set of multiple surfaces arranged arbitrarily just can not be represented accurately with a single surface model. Thus, the filtered version of shading parameters described on such a set of surfaces can not be approximated on a single surface. This problem becomes obvious when we want to render scenes like a forest or fur on a character as presented in figure 1.6. As illustrated in Figure 1.7, the geometrical simplification of objects like trees can not lead to an accurate reconstruction of the shading of the high resolution mesh.

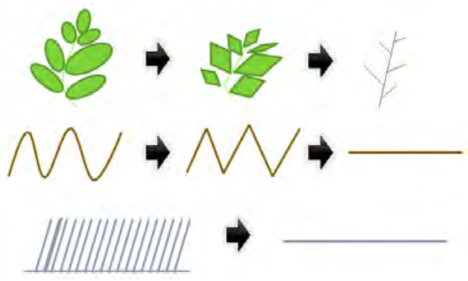


Figure 1.7. Illustration of wrong results from the simplification of meshes that can not be accurately simplified.

Multi-representation and IBR approaches

When objects need to be represented at multiple scales and geometric simplification fails, image-based rendering techniques (IBR) and multi-representation approaches can be used. Image-based representations can be very efficient for distant view as they can naturally fuse multiple objects. For instance, in order to render a forest like the one presented in figure 1.8, video-games usually rely on multiple representations [WWS01]. In such a case, close trees are generally represented using high-detail triangle meshes, while mid-distance trees are simplified versions of this mesh, generally using billboards for the leaves, and far trees are most likely entirely rendered using so-called impostors.

Although automatic simplification of complex 3D models using impostors has been proposed [DDSD03], creating such multiple representations can hardly be done automatically and usually still requires a lot of work from the artists. In addition, continuous transition between different representations is extremely difficult to achieve and such an approach leads to popping artifacts when passing from one representation to another. Impostors provide a correct representation only when displayed for a bounded viewing region called view cell, and they usually offer only limited parallax effects. The other problem with image-based representations is that they do not offer the free transformation and easy manipulation of standard geometry.

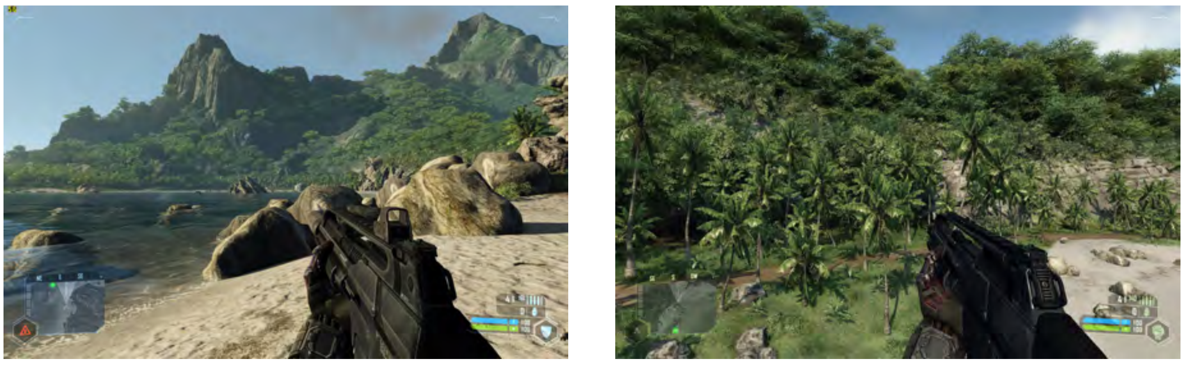


Figure 1.8. Illustration of forest rendering in video-game. Trees are rendered using multiple representations, usually high resolution geometry for closeup views, simplified geometry for medium distance, and impostors for far views. Such a scheme induces low quality geometry filtering and popping artifacts. Source: Crysis (*Crytek*)

Inefficiencies of current GPU pipelines

GPU accelerated rendering pipelines are based on a rasterization approach. Historically, these pipelines have always been optimized in order to provide the best performance for real-time applications. Current trends to improve the rendering quality in real-time applications focus on increasing the number of rendered geometric primitives and decreasing their size. Ultimately, this trend should drive real-time rendering approaches to a point where adaptive geometry approaches would be use to ensure the projection of approximately one triangle per screen pixel as illustrated in Figure 1.9. But with this trend, we are reaching the limit of efficiency of the rasterization-based rendering pipeline.

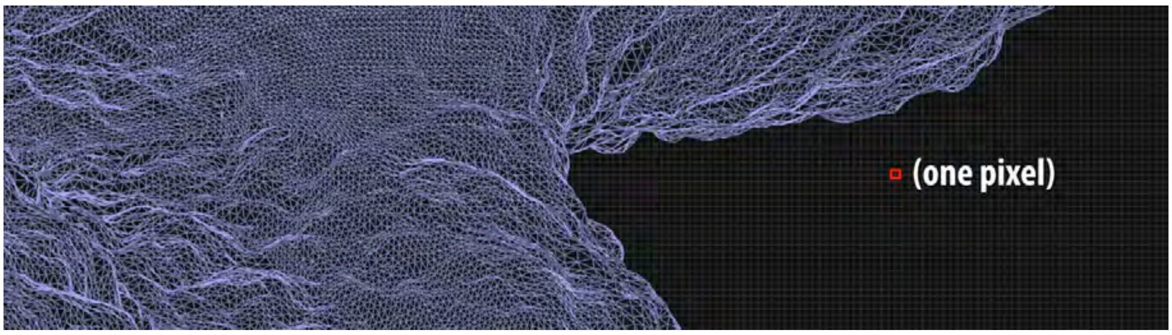


Figure 1.9. Illustration of a mesh adaptively tesselated in order to sample detailed surface accurately and provide triangles approximately 1/2 pixel in area. *Source: Kayvon Fatahalian, Stanford University, Beyond Programmable Shading Course ACM SIGGRAPH 2010*

Inefficiency for micro-geometry rendering Rendering micro-geometry (with triangles projecting on less than a few pixels in area) on the GPU is very inefficient, even on today's last generation GPUs that are designed to perform hardware tessellation of geometry [Mic11]. This is due to two main factors: the individual transformation and projection of each vertex and the inefficiency of the setup, rasterization and shading process that are optimized for big triangles.

Inefficient rasterization of micro-triangles Even the most recent hardware implementation of the rasterization pipeline [Bly06] is very badly fitted to micro-polygon rasterization [FBH+10]. Indeed, fundamentally the polygon rasterization algorithm is based on the assumption that triangles cover multiple pixels and that a filling has to be done inside them. In addition, GPUs contains many optimizations to speed-up the rasterization of relatively big triangles, covering tens or hundreds of screen pixels (e.g. hierarchical rasterization, parallel coverage tests, compressed tile-based z-culling [NBS06], etc.). In the case of pixel-sized triangles, such optimizations become inefficient and even counter-productive [FLB+09].

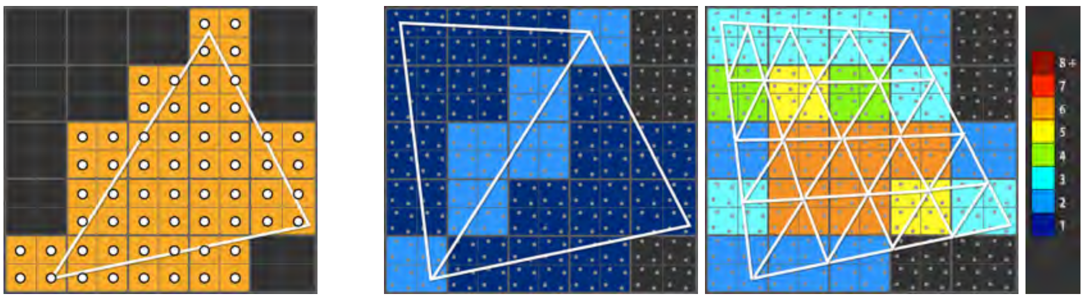


Figure 1.10. *Left:* 2×2 quad fragments generated and shaded during the rasterization of a single triangle. *Right:* Overshading created an the boundary of triangles intersection the same quads. *Source: Kayvon Fatahalian, Stanford University, Beyond Programmable Shading Course ACM SIGGRAPH 2010*

Over-shading GPU rasterization pipelines shade each triangle uniformly in screen space with one shading sample per pixel. Thus in theory, pixel-sized triangles should lead to the computation of approximatively one shading sample per pixel. However, actual GPU pipelines shade at a much higher frequency even for smooth connected surfaces. The information needed to compute the shading for a triangle at a pixel is encapsulated into a *fragment*. Each rasterized triangle generates fragments for each pixel it overlaps[1], and shading is computed once per fragment. But when the triangles are small, many pixels contain multiple fragments generated from different triangles of the same surface due to partial pixels overlap (cf. figure 1.10), leading to pixels being shaded redundantly with similar surface information (in case of connected surfaces). In addition, actual GPU behavior is even worse than that. Indeed, to support derivative estimation using finite differencing, the rasterizer generates fragments assembled in 2x2 pixel tiles called *quads* [Ake93]. This can result in more than 8× overshading for pixel sized triangles of a smooth surface as illustrated in Figure 1.11.

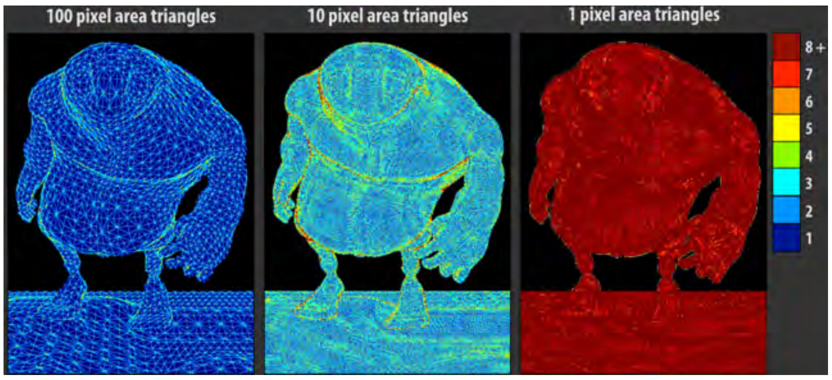


Figure 1.11. Illustration of overshading induced by microtriangle tessellation. *Source: [FBH+10]*

We see that with pixel-sized triangles we reach the limit of the rasterization model. When triangles become smaller than one pixel, shading becomes inefficient. In the context of off-line rendering for

[1]More precisely, a triangle must overlap the center of a pixel or one of its coverage samples (in case of MSAA) in order to generate a fragment.

special effects, high quality shading of pixel-sized geometry is often desired to provide optimal image quality. Thus to overcome the limit of the rasterization pipeline, another rendering approach is used: REYES.

Offline high quality rendering pipelines

REYES [CCC87] is an off-line rendering pipeline dedicated to the rendering of micro-polygons. Although it is still non-interactive, it has recently been shown that such a pipeline can be efficiently implemented on the GPU as a data parallel pipeline [ZHR+09]. REYES is based on adaptive tesselation and allows adapting the shading computation to the screen resolution. The idea of REYES is to transform large control patches instead of individual micro-polygons, and then to tessellate ("dice") only visible ones to adaptively generate pixel-sized polygons. These micro-polygons are shaded per-vertex instead of per-pixel as it is the case for rasterization-based rendering, providing a high quality object-space shading. In order to get the final image, these shaded micro-polygons are multisampled per-pixel at a high frequency. Each sample is z-tested and all samples are combined to compute the final pixel color. This approach allows us to adapt the geometry to the rendering resolution, and the per-patch transform stage of REYES allows us to greatly reduce the cost of transforming micro-geometry. These patches also allow a fast visibility test to quickly reject invisible patches.

However, while this approach works well for precisely rendering smooth surfaces described using NURBS or Bezier patches (potentially perturbed using a 2.5D displacement map), it does not scale to the rendering of complex geometry originally designed with 3D micro-details. In this case, control patches became sub-pixels and all their advantages are lost.

1.1.2 Voxel-based approach to geometry pre-filtering

As we have seen, combining a high quality alias-free rendering with interactive performance for multi-scale scenes and complex objects is a difficult challenge. Current rendering approaches do not scale to such complexity and alternative solutions need to be developed. Our conviction is that in order to maintain rendering performance, data storage and also rendering quality scaling with the increase of geometric complexity, some kind of geometrical appearance pre-filtering needs to be employed. Basically, the overall idea is to pay only for what you see, both in terms of computation and in terms of storage.

Geometry pre-filtering

The idea of pre-filtering is that, under some linearity assumptions, it is possible to factor some shading parameters out of the shading computation when integrating it over an area (in this case a pixel), and to pre-integrate (average) these parameters separately (cf. Section 2.1.2). This permits us to remove detail frequencies higher than the rendering resolution while preserving the mean value of the pixel footprint. Pre-filtering not only textures but the whole geometry definition would allow alias-free rendering of very complex geometry. Such geometry pre-filtering would provide a scalable rendering solution with an amount of rendered data depending only on the rendering resolution, and thus scaling up to very complex scenes. As we have seen in Section 1.1.1, simplifying a surface representation using a simplified surface definition does not provide accurate filtering that maintains shading details and correct appearance.

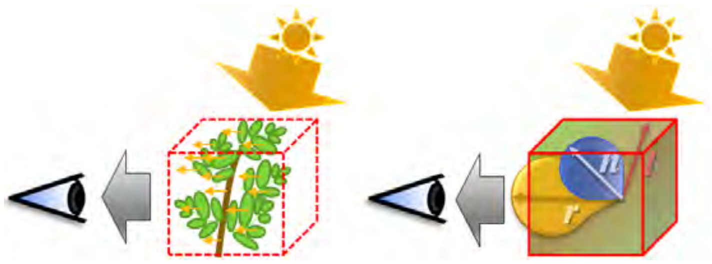


Figure 1.12. Illustration of the pre-filtering of complex geometrical details inside a single voxel element in order to allow the reconstruction of an overall shading contribution.

The key observation that allow the pre-filtering of geometry was made by Perlin [PH89] and by Kajiya and Kay [KK89]. When considering a given volume of space containing multiple surfaces more or less randomly distributed (like the branch and leaves illustrated in Figure 1.12), exact positions of surfaces inside this volume do not really matter when computing the overall light interaction within this volume [2]. Thus, only using an overall density distribution and overall reflectance function is enough to accurately compute the interaction of light with this volume (Fig. 1.12). Then, when sets of complex surface definitions are considered, the parameters used for computing illumination for such sets can more easily be described volumetrically, for a given volume containing these surfaces, than with a simplified surface definition. With such volumetric representation, the matter is represented by a density distribution associated with the parameters for the shading model describing the way light is reflected within a unit of volume, instead of a set of interfaces and parameters on them. Once the geometry is transformed into density distribution, filtering this distribution becomes a simple linear operation [Ney98].

[2] As long as positions are not deeply correlated

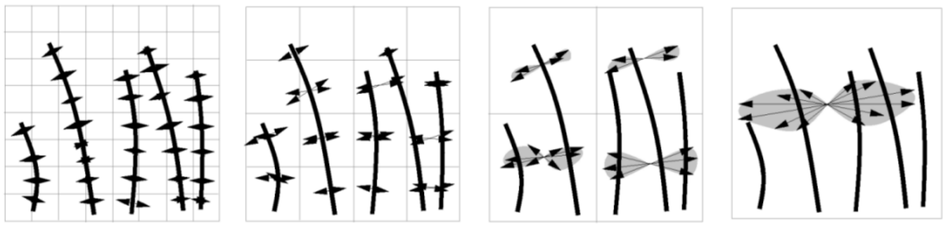
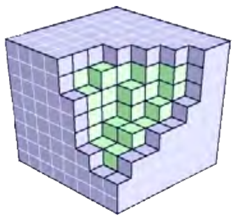


Figure 1.13. 2D illustration of voxel-based geometry pre-filtering inside a MIP-map pyramid and storing normal distributions. *Source: [Ney98]*

Voxel-based representation



The name *voxel* comes from "volumetric elements" and it represents the generalization in 3D of the pixels. Voxels represent the traditional way to store volumetric data. They are organized in axis-aligned grids subdividing and structuring space regularly. Voxel representations offer a promising way to store volumetric data in order to unify texture and geometrical representations, while simplifying filtering. The significant advantage of voxels is the richness of this representation and the very regular structure which makes it easy to manipulate. That makes voxels a good candidate to address aliasing issues that are hard to deal with in triangulated models.

Multi-resolution representations are easily obtainable based on MIP-mapping of voxel grids, making output-sensitive results possible. In Figure 1.14, voxel MIP-mapping with inter-levels interpolation allows exactly adapting the geometrical resolution to the screen resolution. In contrast to image-based approaches (IBR), voxel-based representations preserve a real 3D geometry that can be easily manipulated, transformed and eventually mixed with a mesh scene [Ney98]. As we will see in Section 4, accurate geometry pre-filtering is not simply averaging densities, but also supposes a pre-integration of the visibility in order to ensure a correct occlusion between filtered elements.

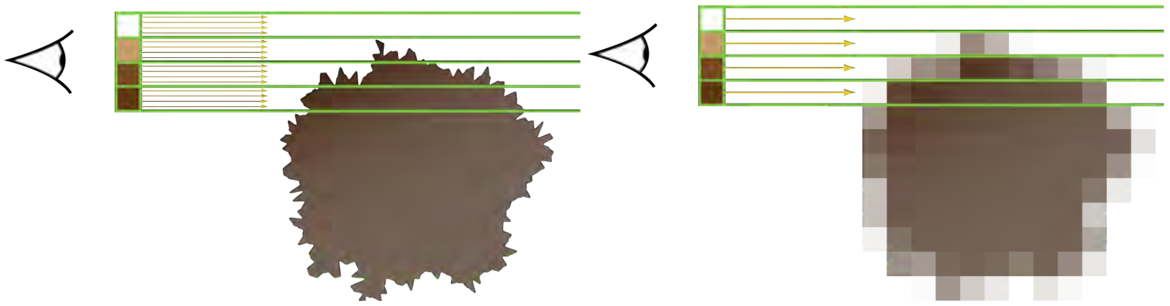


Figure 1.14. Left: Illustration of an highly detailed object rendered (through 4 pixels) with adaptive multisampling (left) and voxel-based pre-filtering (right)

What we propose is to rely on a unique voxel-based volumetric representation for both the coarse geometry and the fine-grained details (material) that can be pre-filtered accurately. This representation stores both the material and the geometry parameters of the shading model into a voxel grid as we will see in Section 4.2. Typically, each voxel contains a density-based opacity information, a material color and a normal distribution.

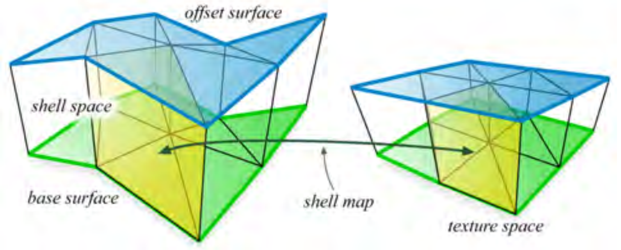
We will see in the next section that, beyond the use of voxels for geometry pre-filtering, rendering such representations can provide other interesting advantages. In particular, voxel representations handle semi-transparency gracefully due to their inherent spatial organization, whereas triangles would need costly sorting operations. In addition, such a structured representation simplifies various computations, including simulation processes.

Animation and integration into classical video-game scenes

Voxels are usually envisioned as a totally alternate scene representation, dedicated to rigid motionless data (except when the whole data are regenerated every frame such as for fluid simulation). We would like to motivate the full usability of voxel representations in video-game applications. Particularly we would like to emphasize their capacity to smoothly integrate with mesh-based representations and to handle animation and free transformations that are mandatory in such applications.

Within this thesis, we designed the foundation (in terms of geometry representation, data-structure, rendering algorithm and management of large voxel datasets on the GPU) that opens the way to a large class of new researches based on this representation. We also demonstrate smooth integration of voxel-based objects inside traditional mesh-based scenes (Chap. 8).

The animation and deformation aspects are not the primary focus of this thesis and we just briefly address real-time animation of voxel data in the last part of this work (Chap. 9). However, animation and deformation (skinning) will clearly be part of the future work required to bring voxel representations as a fully usable rendering representation in modern video-game applications. Nevertheless, the problem of animating voxel-based representations has been previously studied and realistic ways of tackling the problem have been proposed, in particular for volumetric textures [Ney95, Ney98]. We strongly believe that *shell map* based approaches [PBFJ05, JMW07] are very promising research directions to tackle real-time animation of detailed voxel representations.



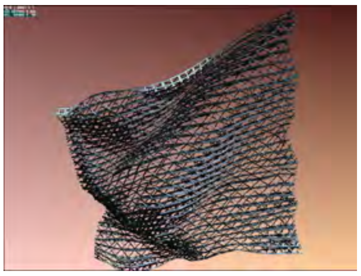


Figure 1.15. *Left:* Illustration of trilinear patches used for world to texture space mapping. *Right:* Scaffolding mapped onto an animated flag surface. *Sources: [PBFJ05, Ney98]*

World to texture space parametrization Voxel representation can be used either to represent an entire multi-scale scene that can be viewed from a very large range of distances, or as volumetric textures [KK89, Ney98] in order to add fine-grained volumetric details inside a shell at the surface of a rough triangle mesh. We see both applications as a similar problem, with the only difference being the accuracy of the parametrization providing a mapping between the texture space, containing the voxel data, and the world space of the scene. This parametrization is usually specified by trilinear patches [Ney98, PBFJ05, JMW07] with 3D texture coordinates affected to the vertices of the patch.

Since the voxel representation needs to be enclosed within this geometrical parametrization, the scale at which it is defined limits the maximum degree of filtering that can be provided by the voxel representation, and thus the maximum distance at which the object can be filtered. Indeed, if a patch projects on less than one pixel, its content can no longer get pre-filtered.

Animation Two different ways of animating voxel data can be distinguished [Ney98]: the animated deformation of the parametrization (both control points of the patches and texture coordinates) and the actual animation of the content of the voxels.

Animation of voxel content supposes a per-frame update of the data that can be extremely costly. However, this approach allows grouping multiple objects inside the same (non-animated) parametrization, and so allows a high degree of filtering (and a large range of view distances). It has the advantage of preserving spatial structuring between represented objects, facilitating visibility determination for example. We demonstrate such animation in chapter 9.

On the other hand, animation of the geometric parametrization (e.g. a shell made of trilinear patches) allows us to deform displayed volume objects in world space. This allows us to use volumetric textures exactly as a 3D generalization of traditional 2D texture mapping: Transformations can be computed on a rough geometry representation (the geometric parametrization) while fine-grained volumetric details are simply interpolated inside it. With such approach, the scale of the parametrization geometry also determines the precision of the animation that can be achieved. Precise animation requires the parametrization geometry to be detailed enough. Thus, the scale of the parametrization must be chosen as a compromise between the maximum degree of filtering, and the precision of the deformation. We believe that a continuous range of mapping scales can be of use in different situations.

1.2 Other applications and usage of voxel representations

Beyond our primary motivation in using volume representation for geometry filtering, such representation also presents other usages and interesting properties. The most obvious reason for rendering voxel representation is that voxels are a natural representation for volumetric phenomena and participating media.

1.2.1 Voxels for volumetric materials and effects

As described previously, depending on the scale on which you observe solid objects and materials, their interaction with light can be more efficiently described either by a surface or by a volumetric statistical density and lighting parameters information. There are also pure volumetric phenomena like flames, foam, clouds or smoke that represent a semi-transparent participating medium and do not actually have a well defined boundary surface at all. Such objects can only be described volumetrically and are currently not much used in video-games and real-time applications because they are too complex and too costly to render in real-time. Indeed, an accurate volumetric rendering of such effects requires an ordered traversal of the data that hardly fits inside the classical rasterization pipeline. Volumetric or semi-volumetric phenomena are usually faked in video-games using impostors [KW05] but the rendering quality is usually low as shown in Figure 1.16 (left).

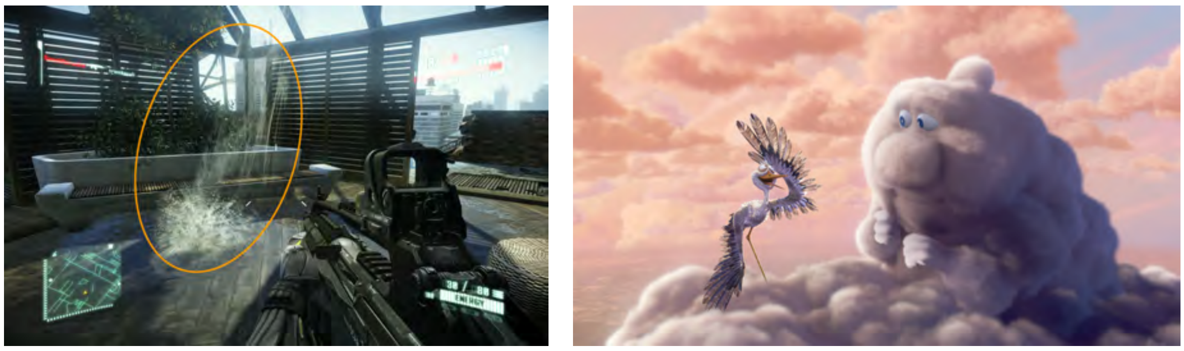


Figure 1.16. *Left:* Example of the rendering of a fall of water in a recent video game *Source: Crysis 2 (Crytek). Right:* Example of cloud rendering for special effects *Source: Partly Cloudy (Pixar).*

1.2.2 Voxels in special effects

In contrast to video games, large multi-scale volumetric effects are very often used for offline special effects by the movie industry [WBZC$^+$10] as seen in Figure 1.17. Examples can be found in many movie productions (*e.g.,XXX, Lord of the Rings, The Day After Tomorrow, Pirates of the Caribbean, The Mummy 3*). Clouds, smoke, foam, splashes, 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. For such effects, the voxel representation is usually generated from a combination of multiple source data like particles, fluid simulations or detailed meshes. Rendering such phenomena requires massive voxel grids and special effects companies such as *Digital domain, Cinesite*, or *Rhythm 'n Hues* now massively rely on so-called *voxel engines* [Kis98, BTG03, Kap03, KH05]. These off-line engines produce very realistic looking images, but at the cost of tera bytes of memory and hours of rendering computations. 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, which is a serious problem for artists designing the scene.

Figure 1.17. Example of volumetric special effects in movie productions. *Sources: The Day After Tomorow , XXX (Digital Domain)*

1.2.3 Scan and simulation data

Voxels are also the native representation of much scientific data, like 3D scans or tomographic reconstructions from radiological data (Fig. 1.18). Voxels are also used for representing simulation processes like fluid simulations that are based on an Eulerian grid. The rendering of such datasets is done through the use of scientific visualization techniques that were the first domain to deeply explore volume rendering as described in Section 2.3.

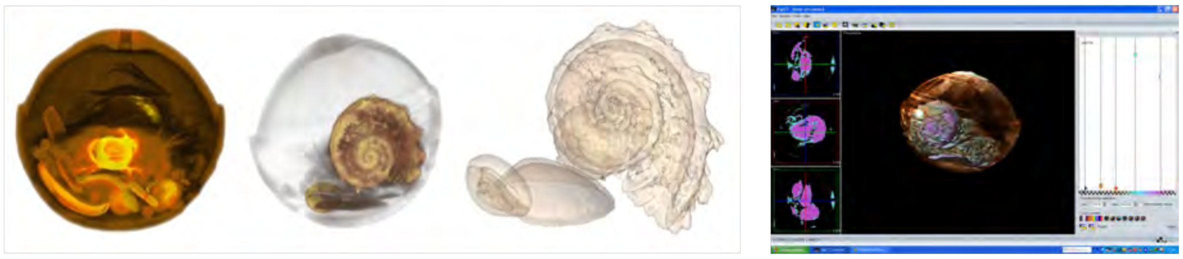


Figure 1.18. Examples of the visualization of archaeological objects scanned using X-Rays and reconstructed using tomography. *Source: Digisens 2009*

1.2.4 Voxels as an intermediary representation for authoring

Voxels are also used as an intermediate representation in the production pipeline of digital movies, special effects or video games. Indeed, another interesting property of voxel representations is that they allow very easy CSG operations. Thus, some modeling tools rely on voxels in order to store implicit surfaces and allow easy virtual sculpting (Fig. 1.19, right). The same property is also used by video game companies to easily model terrains (Fig. 1.19, left). In this context, the voxel representation is also used during the production phase to assimilate various object meshes into one unified well-conditioned textured mesh and to easily generate geometrical LODs (in the spirit of [ABA02]). In the same spirit, voxels are used by the special effects industry as an intermediate representation to generate surface meshes from point clouds acquired using a scanning device in order to digitalizes models or real-life objects that would be too complex for an artists to digitally recreate [For07].

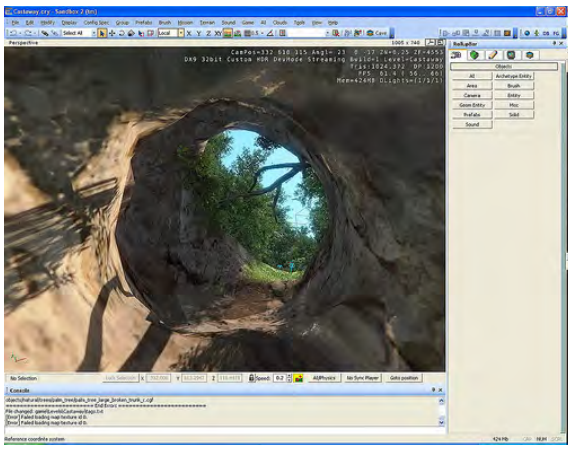
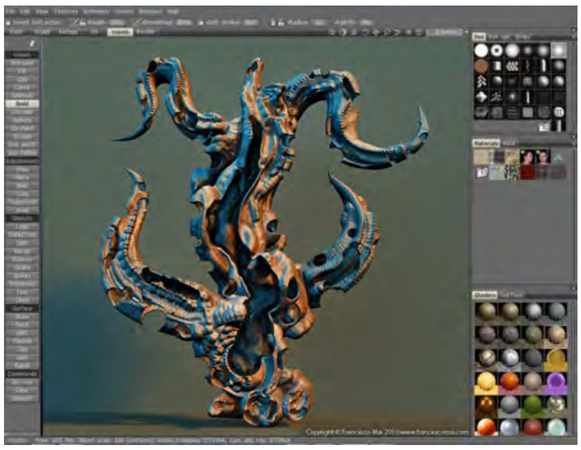



Figure 1.19. *Left:* Voxel-based terrain authoring in Crytek's CryEngine *Source: Crytek (2010). Right:* Highly detailed model designed using 3DCoat voxel sculpting software *Source: Francesco Mai (2010).*

1.3 GPU development

Graphics hardware acceleration is a rapidly evolving area. Since a large part of our work focuses on making the best use of the GPU (Graphics Processing Units) in order to propose an alternative voxel rendering pipeline, we will quickly describe its main characteristics.

1.3.1 Hardware generations

During the last 10 years, commodity graphics hardware generations have been driven by the evolution of Direct3D [Mic11], the most frequently graphics API[3] in the video games industry. Each version of Direct3D standardize a set of features called a "Shader Model" that must be supported by the programmable shader units of the GPU [Bly06]. Thus, graphics hardware generations are traditionally designated by the version of this Shader Model they support. The first "modern" graphics hardware supporting programmable shading was SM1.0 (NVIDIA GeForce 3, Direct3D 8.0, 2001). Programmable shading evolved with more and more flexibility up to today's SM5.0 (NVIDIA GeForce GTX480, Direct3D 11.0). More details on the evolution of these shader models and the commodity graphics hardware in general can be found in the excellent reference book of Akenine-Möller et al. [AMHH08].

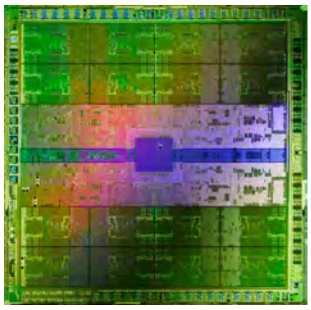


Figure 1.20. Die of the NVIDIA GF100 GPU, Fermi architecture.

1.3.2 Compute mode

Since their creation, GPUs have been dedicated to the acceleration of the rasterization rendering pipeline [AMHH08]. But nowadays (since the introduction of programmable shading units), GPUs evolve toward more generic high performance data parallel processors. In 2007, when this thesis was started, NVIDIA SM4.0 (Direct3D 10.0) generation hardware was introduced with the support of a new "compute" mode, dedicated to general purpose computation on the GPU. This mode accessible through the CUDA API [NVI11a, Kir09] made it possible to bypass the rasterization pipeline in order to use the GPU's programmable shading units as high performance general purpose parallel processors. This new way to address the GPU is radically changing the way we develop hardware accelerated graphics algorithms, gradually moving toward more generic parallelism problems. All the work present in this thesis has been prototyped and tested using OpenGL [Khr] grahics API and the CUDA API.

1.3.3 Limited video memory, limited bandwidth from the CPU

The video memory embedded on the graphics card is a high bandwidth low latency memory accessible from the GPU. This memory is used to store textures, mesh definitions, framebuffers and constants in graphics mode and can be accessed as a generic memory through pointers in CUDA. As illustrated in Figure 1.21, the bandwidth between the GPU and the video memory is hundreds of GB/s (close to 200GB/s on current high end GPUs), that is at least 4 times higher than the bandwidth between the CPU and the main memory. Despite this high bandwidth, video memory accesses from CUDA threads still represent hundreds of cycles of latency, while arithmetic operations only cost a few cycles [WPSAM10]. A cache hierarchy is used inside the GPU to speed-up accesses to the video memory by allowing data reuse between threads.

[3] Application Programming Interface

While access to the video memory is relatively fast from the GPU, the problem is that its amount is usually limited compared to the system's main memory (around 1/16 to 1/8 of the storage capacity), thus limiting the amount of data that can be kept in this high speed memory. Worse yet, access to the video memory from the CPU is tremendously slow compared to the access from the GPU (around 20 times slower). Therefore, transfers of data between the video memory and the system memory must be limited to maintain high performance in real-time applications.

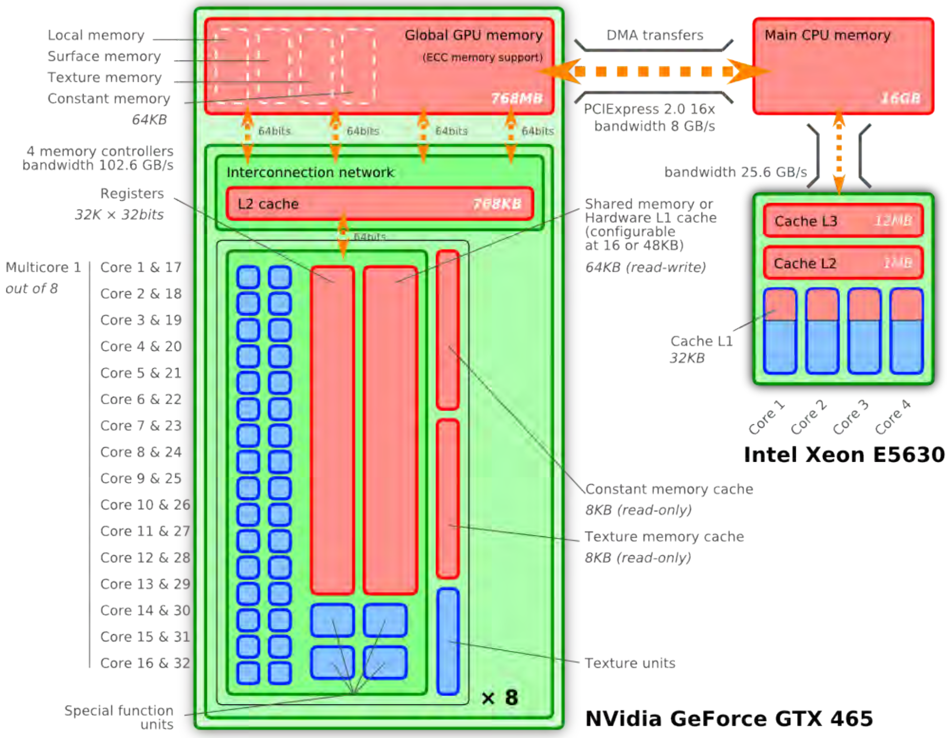


Figure 1.21. Illustration of memories and caches accessible by the GPU and connections to the rest of the system. *Source: Vincent Jordan,* `http://www.kde.cs.tsukuba.ac.jp/~vjordan/docs/master-thesis/nvidia_gpu_archi/`

When doing graphics rendering or a compute operation on today's GPUs, all data needed for this operation must have been previously loaded in video memory. GPUs do not provide a virtual memory mechanism with demand paging similar to the one provided in system memory by the operation system [SGG08]. This implies that all data necessary for a rendering operation must fit into the video memory.

1.3.4 Memory regions

The video memory embedded on the graphics card and accessed by the GPU is split into two main types of memory regions: the global linear memory and the texture memory. The global memory can be accessed from the parallel processing units of the GPU through the pointer in compute mode exactly in the same way as the system memory of the CPU. On the other hand, the texture memory is dedicated to the storage of 2D or 3D textures. It benefits from a special organization that enhances 2D and 3D spatial locality. This makes it possible to maximize the efficiency of the hardware caches when textures are sampled at spatially close locations. Texture memory is not addressed directly by the parallel processing units but is accessed through *texture samplers* that are special dedicated hardware providing very fast linear interpolations.

1.3.5 Increasing gap between memory access speed and processing power

The current trend in the design of graphics hardware (and processors in general) is leading to a widening gap between memory access speed and data processing speed. While the increasing parallelism of GPUs results in a performance improvement that tends to follow or even outperform Moore's law exponential growth prediction, memory bandwidth and data access speed grow at significantly slower rates. This relative performance gap is illustrated in Figure 1.22.

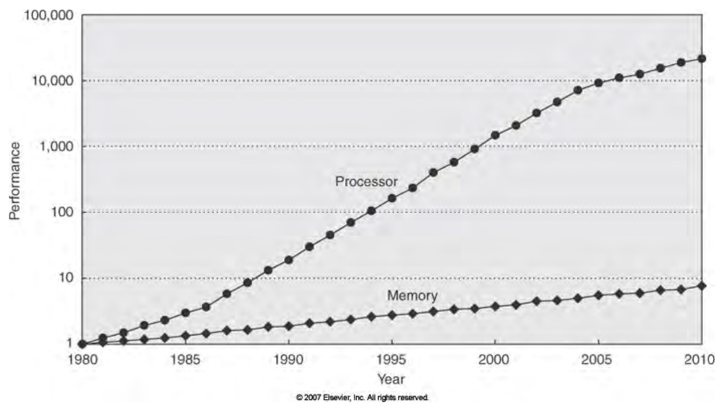


Figure 1.22. Evolution of the performance gap between memory access speed and computational power. *Source: Elsevier Inc.*

In such a context, the major computational bottleneck is usually data access rather than computation, and this is true at all levels of the memory hierarchy, from video RAM to in-GPU parallel processor clusters (Stream Multi Processors) L1 cache and shared memory. We expect this trend to continue and to become even worse in the future. In such context the major challenge in high performance rendering is to design rendering systems able to carefully manage bandwidth requirements and ensure minimal and coherent data access, while maximizing cache efficiency and data reuse.

1.4 Problems and objectives

This thesis aims at making voxels an alternative rendering primitive for real-time applications. Voxel representation has been intensively used in the past, especially in special effect productions, to render participating media and volumetric effects. While such an application is also of interest for us, we aim to introduce voxel representation as a standard primitive to render complex geometrical details and filter multi-scale scenes in real-time applications.

We show that the current graphics 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 in video-games or to previsualize special effects. Our rendering pipeline is 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.

1.4.1 Problems with voxel-based rendering

The first problem is to design a pre-filtered geometry model stored inside a voxel representation.

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 (Sec. 1.3).

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 in real-time, is anything but trivial and will make possible a more involved usage of volumetric rendering in real-time applications.

Thus, there are three major issues to overcome before making detailed rendering of massive volumes a standard rendering primitive:

- How to store a pre-filtered voxel representation ?
- How to render it efficiently ?
- How to handle large amounts of voxel data ?

The storage problem

The first problem when dealing with voxel representation is the choice of the data structure. Basically, voxel representation structures space within a regular grid of values. Such volume data can require *large amounts of memory*, thus *limiting the scene's extent and resolution of details*. As we have seen in Section 1.1.2, we want to employ voxel data as a MIP-mapped representation of the geometry. Such MIP-mapping scheme makes it possible to adapt the volume resolution used for rendering to the screen resolution and the distance to the viewer.

In order to ensure high-quality rendering, our filtered voxel representation must provide enough resolution so that each voxel project on an area not larger than one pixel of the screen. However, storing such plain grids to adapt to the screen resolution can quickly represent gigabytes of data. For instance, with a 1280×1024 screen resolution, displaying one voxel per pixel would require at least a 1280^3 voxels grid. Such a grid would already represent 8GB of data when storing only one RGBA value per voxel, with one Byte per component.

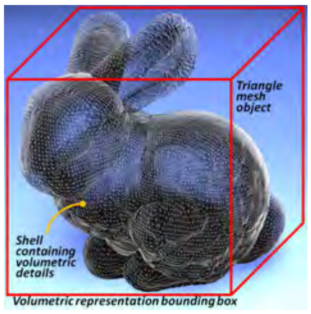
Data has to stay in the video memory embedded on the graphics card in order to be quickly accessible for rendering by the GPU. This memory is a high bandwidth, low latency memory connected to the GPU and dedicated to graphics rendering operations, but it is many times smaller than the system memory (Sec. 1.3). It is usually limited to 1GB or 1.5GB. In addition, in a video-game context, only a small subset of this total amount of video memory would be available to store such rendering data. Thus, the amount of memory required to store the voxel representation must be reduced and must be kept as low as possible.

The bandwidth problem

By keeping voxel representation as compact as possible, our goal is to handle arbitrary large scenes without any restriction on their memory occupancy. Thus, the scene can no longer be held entirely in video memory. It implies the need for *intelligent data streaming* from larger, but slower memory units. While the connection between the GPU and the video memory is high bandwidth and low latency, the connection between the video memory and the larger system memory provides a lot lower bandwidth and higher latency (Sec. 1.3). Thus, transfers between the large system memory and the limited video memory are very costly and must be kept as low as possible. In such a context, during the exploration of a scene, efficient data management techniques must be employed in order to maximize the reuse of the data already loaded in video memory, while transferring only the minimum amount of required new data.

1.4.2 Target scenes

It is true however, that a full volume is not always necessary. For computer graphics scenes, it often suffices to have detail located mostly in a layer at the interface between empty- and filled space. This is the case when adding volumetric details within a thin shell at the surface of an object as illustrated in the figure on the right, but it is also true for many scenes that are composed of sharp or fuzzy objects lying in mostly empty space (see Figure 1.23). Moreover, note that all real interfaces are fuzzy at some point since opacity is also a question of scale: Physically, very thin pieces of opaque material are transparent, and light always enters a few steps in matter. Moreover, in terms of LOD the correct filtering of infra-resolution occluders yields semi-transparency.


Triangle mesh object
Shell containing volumetric details
Volumetric representation bounding box

Our scheme is optimized for such an assumption: We expect details to be concentrated at interfaces between dense clusters and free space, *i.e.,*”sparse twice”. More generally, we deal with high-frequency interfaces between regions of low frequency within the density field: We treat constant areas (*e.g.,* core regions) just like empty areas. Our key idea is to exploit this observation in the design of our storage, rendering, streaming and visibility determination algorithms, together with the exploitation of temporal coherency of visible elements during the exploration of a scene.

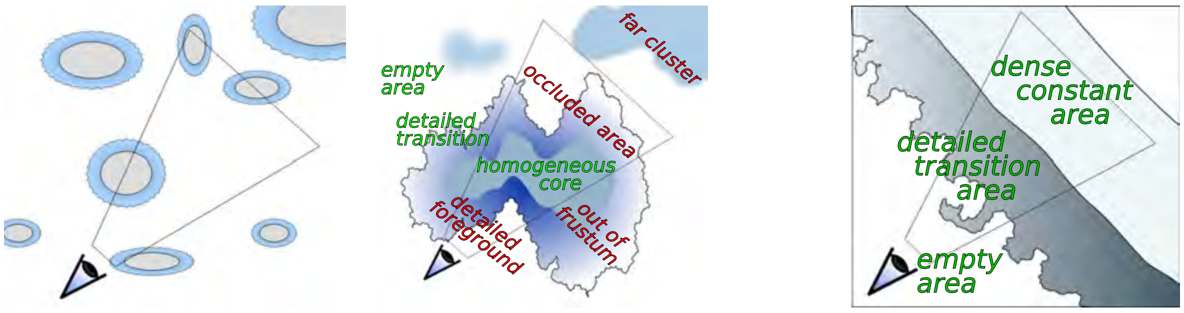


Figure 1.23. In many scenes, details tend to be concentrated at interfaces between dense clusters and empty space.

1.4.3 Objectives

Our main goal is to design a method capable of achieving the real-time rendering of very large amounts of voxel data representing large and detailed pre-filtered geometrical scenes. Our primary objective is to ensure a full scalability of the proposed solutions and algorithms, in order to allow an arbitrary increase of the size or complexity of the scenes that can be rendered in real-time.

This was achieved through three main technical objectives:

- **Rendering only visible data, at the needed resolution**. In order to scale to large scenes, rendering cost must be as much independent of the amount of data as possible, and should only be dependent on the screen resolution and distance to the viewer.
- **Loading only visible data, at the needed resolution**. The key idea is that ideally, we want to load and keep in video memory only the few voxels per pixel required for rendering.
- **Exploiting time coherency and reusing loaded data**. Once loaded into video memory, data must be kept loaded as long as possible in order to maximize its reuse among multiple frames, and minimize the cost of transferring data from system memory (or producing procedural data).

In addition, our approach must ensure a good integration inside current triangle-mesh based rendering pipelines, and must be compatible with animation that we see as a future work.

1.5 Contributions

In order to bring memory-intensive voxel representation as a standard GPU primitive, we propose a new rendering pipeline for high-performance visualization of large and detailed volumetric objects and scenes. This thesis offers contributions on four core aspects:

- A model for appearance preserving pre-filtered geometry based on a voxel representation coupled with a fast approximate cone tracing approach. Our voxel-based cone tracing relies on the pre-filtered geometry model to efficiently approximate visibility and lighting integration inside a cone footprint, providing cheap anti-aliasing (Chap. 4).
- A GPU-based sparse hierarchical data structure providing a compact storage, and an efficient access and update to the pre-filtered voxel representation (Chap. 5).
- A GPU-based rendering algorithm, based on ray-casting, and efficiently using our data structure and implementing our voxel cone-tracing (Chap. 6).
- An efficient GPU-based virtual memory scheme providing efficient caching and on-demand streaming and data production inside our data structure updated dynamically. This mechanism is entirely triggered by requests emitted per-ray directly during rendering, providing exact visibility determination and minimal data production or loading (Chap. 7).

Based on these main contributions and our new voxel-based geometry representation, we demonstrate fast rendering and exploration of very complex scenes and objects. In addition, we also demonstrate two other rendering effects that our model allows to compute very efficiently: soft shadows and depth-of-field (Chap. 8).

Finally, we introduce a new real-time approach to estimate two bounds indirect lighting as well as very fast ambient occlusion, based on our pre-filtered geometry representation and voxel-based cone tracing (Chap. 9).

Chapter 2

Related work

In this chapter, we present the previous work on all aspects addressed in this thesis as well as the different techniques we rely on. Our work builds on related work from different domains: scientific visualization, visibility detection, real-time rendering, spatial data structures, GPU algorithms, out-of-core data management, etc.

In Section 2.1 we first describe the previous work on anti-aliasing filtering for both geometry and textures. In the next sections, we discuss the primary volume representations used in computer graphics (Section 2.2) and in scientific visualization (Section 2.3). Then, we describe different techniques employed to deal with complex and memory heavy datasets in Section 2.4. In this section, we first describe available rendering techniques (Sec. 2.4.1), before presenting spatial data structures allowing fast random access as well as data compaction (Sec. 2.4.2). We then present visibility culling techniques (Sec. 2.4.3) and multi-resolution rendering approaches (Sec. 2.4.4). In the last section, we present out-of-core data management techniques (Section 2.5.3).

2.1 Anti-aliasing filtering

Since first discussed by Crow [Cro77] in the middle of 1970s, anti-aliasing has been an highly studied problem in computer graphics. There are two main antialiasing strategies: *Supersampling*, computed in image space, and *pre-filtering* computed in object space. Prefiltering removes the high frequency components of the geometry it samples, while supersampling averages together many samples over a pixel area.

2.1.1 Geometry anti-aliasing filtering

Supersampling

The idea of supersampling is to trace an image at higher resolution and to average the results per pixels. This operation is computationally intensive and very costly. In order to produce high quality antialiased images at more reasonable sample rates, Whitted suggested an adaptive supersampling scheme in which pixels were recursively subdivided for further sampling only when necessary [Whi80]. The problem is that such uniform and regular sampling, even when hierarchical, creates visible patterns like moire patterns leading to a kind of aliasing.

Distribution (or distributed) ray-tracing (DRT) was introduced by Cook et al. [CPC84] in order to deal with both spatial and temporal aliasing using multisampling. Cook et al. proposed to use stochastic sampling of pixels using multiple rays that are spatially and temporally distributed to compute antialiased images and to render non-singular effects such as depth of field, glossy reflection, motion blur, and soft shadows. Since then, many stochastic sampling patterns have been proposed by various authors, based on jittering, stratified, Poisson disks, Monte Carlo or quasi-Monte Carlo sampling. The main problem with supersampling schemes is their cost in terms of rendering time, that stay very high and keep their usage out of the range of interactive applications.

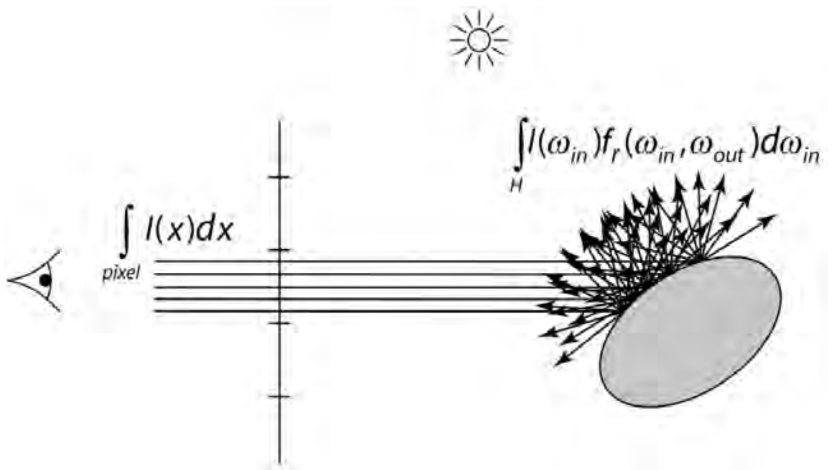

$\int_H l(\omega_{in}) f_r(\omega_{in}, \omega_{out}) d\omega_{in}$
$\int_{pixel} l(x)dx$

Figure 2.1. Illustration of a 5x per-pixel supersampling used for rendering a shaded sphere. Each direct view-sample integrate indirect lighting following the rendering equation. *Source: Don Fussell, University of Texas at Austin, CS384G Computer Graphics course*

Cone or beam tracing

In the context of ray-tracing, some techniques have been developed to solve the anti-aliasing problem using so-called "analytic prefiltering" instead of explicit supersampling. These approaches allow the computation of a single sample to estimate the color of a pixel, but lead to a more expensive primitive intersection test and are not able to precisely estimate filtering of complex geometry.

Amanatides [Ama84] proposed to use cone tracing with an analytical cone-primitive intersection functions. It uses a heuristic approximation for modelling the occlusion inside a cone. The major limitation of the cone tracing algorithm is that this calculation is not easily performed except for simple objects, and it is not suitable for complex aggregated surfaces. It can also generate artifacts caused by the various integration approximations. This approach was extended by Kirk [Kir86] to handle reflections and refractions on bump mapped surfaces. Similarly, Heckbert and Hanrahan [HH84] describe beam tracing, that uses polygonal beams and a hybrid combination of polygon scan conversion and ray tracing. Occlusion is correctly taken into account by clipping the beam with the occluding geometry. Similar ideas were introduced by different researchers, like pencil tracing [STN87], and ray bounds [OM90].

In order to reduce the cost resulting from complex intersection tests when tracing cones or beams, Thomas et al. [TNF89] proposed to rely on a single ray per pixel but to artificially "grow" objects to make sure object silhouettes were not missed during ray-tracing. This ensures a conservative visibility detection and allows the detection of cases where the ray passes close to a surface in order to integrate its filtered contribution. Like cone tracing, ray tracing with covers requires only one sample per pixel to generate an antialiased image but is also limited to models made up of very simple primitives.

All the analytic-prefiltering approaches we just presented are very specialized and limited. They worked well at the time they were developed, but they do not allow us to precisely filter complex materials and reflectance models. In the current state of the art, antialiasing must deal with many different mechanisms that contribute to unwanted high frequencies, that would require special dedicated strategies with analytic methods.

In order to provide antialiased rendering, we take inspiration from the cone and beam tracing approaches, but instead of relying on an analytical intersection function, we take advantage of our voxel-based pre-filtered geometry representation as detailed in Section 4.2.

2.1.2 Texture filtering

Macroscopic geometry is traditionally represented using a triangle-based surface representation, while the finer (microscopic level) scale is described by the local reflection model and traditionally encoded into the BRDF [AMHH08]. This micro-scale description is modulated at mesoscopic and macroscopic levels by the geometrical description of objects, based on surface orientation. The mesoscopic level interaction is generally described using a Normal Map (or Bump Map) representation. Pre-filtering methods have been developed in order to ensure correct integration of these interactions into the shading model, along with a fast real-time evaluation [BN11]. We build upon this previous work on normal map filtering to integrate correct meso-scale geometry orientation information into our voxel-based geometry model, and allow high quality illumination (cf. Section 4.5).

Pre-filtering theory

The idea of pre-filtering consists in factoring the illumination equation. The color to be seen in a screen pixel results from the integration of whatever is visible in the cone starting from the eye and passing through this pixel. This cone covers an area on the shape to be rendered, so that the correct shading is obtained by integrating the local illumination equation on this surface. Pre-filtering techniques rely on the fact that, under some linearity assumptions, it is possible to factor-out from the illumination integration some shading parameters when the result of the shading is averaged over an area, and to filter them separately. This factorization is possible when the shading parameters that are factored-out have a linear relationship to the final color, meaning they are involved only linearly in the shading function.

Assuming that the effect of incoming radiance to a surface element is linear to the reflectance model, the local illumination equation can be represented as:

$$I_R = I_I\, f(N, E, L, C), \tag{2.1}$$

for a given surface element dA for which all terms can be considered constant. I_R is the radiance and the reflected light, I_i the radiance of the incident light, $f()$ the function implementing the reflection model based on the BRDF of the material, N the normal to the surface, E the view direction, L the direction of the incident light, C the color of the surface. To get the total amount of reflected light coming from a finite surface area A, this equation is integrated over A:

$$\int_A I_R(P)\, dA \;=\; \int_A I_I(P)\, f(N(P), E(P), L(P), C(P))\, dA, \tag{2.2}$$

Some terms can be considered constant over A, usually E and L. Also, any quantity that is combined linearly into f can be brought out of the integral and integrated separately. This is usually the case for the color C, in this case, if everything else is constant over A, the equation becomes:

$$\int_A I_R(P)\, dA \;=\; I_I(P)\, f(N, E, L) \int_A C(P)\, dA, \tag{2.3}$$

The MIP-mapping scheme

MIP-mapping has been introduced by Williams [Wil83] for filtering 2D color textures and can be generalized to other surface attributes. The idea is to precompute the integrals corresponding to different scales of pre-filtered surface shading attributes and to store them into a set of texture maps of decreasing resolutions (power-of-two reductions).

This allows us to use the appropriate resolution directly at rendering time thus providing a quasi-correct antialiased image using only a single sample per pixel. Trilinear interpolation is usually used in order to provide a continuous resolution, the area of integration being approximated by its bounding square. More precise anisotropic filtering can also be used in order to approximate very closely the area of integration on the surface.

Normal maps filtering

Normal maps decouple the (low resolution) geometry sampling density from the (high resolution) normal sampling density but it is still necessary to filter it properly in order to avoid aliasing. MIP-mapping geometrical data, such as normals or depths, is not as simple as MIP-mapping material color information. Indeed, such parameters are not linear in the shading equation. Filtering normal map representations has been a long studied problem [BN11]. All normal map filtering approaches rely upon the idea of representing the normal information using a statistical distribution of normal directions over a surface, that can be linearly combined. This representation is defined by a *normal distribution function* (NDF) giving the density of normals as a function of direction for each point of a surface.

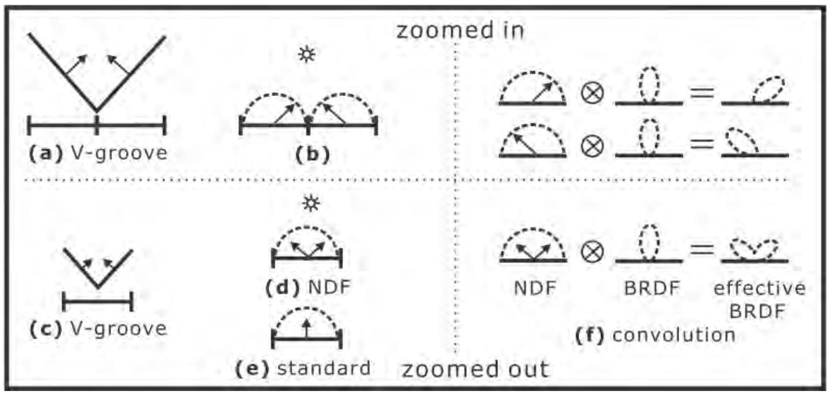


Figure 2.2. Illustration of the use of a normal distribution function (NDF) to represent a V-shaped surface element (a). In closeup view, each face normal is stored in a single pixel of the normal map (b). When this map is filtered these pixels must be combined into a single pixel (c). Standard MIP-mapping averages the normals to an effectively flat surface (e), while an NDF definition preserves this distribution of normals (d). This NDF can be linearly convolved with the BRDF (f) to obtain an effective BRDF used for shading. Source: [HSRG07]

Some methods approximate the normal distribution function (NDF) using a single Gaussian lobe. [Sch97] proposed to describe this lobe using a mean direction obtained by perturbing a polygon normal using a bump map, and a roughness represented using covariance matrices and stored into a separated roughness map. He also showed how a single standard deviation can be used in case of isotropic roughness. [ON97] introduced a single 3D Gaussian lobe representation describing probability distributions on the entire sphere, instead of on an hemisphere as in previous methods. This allows correct multi-scale combinations when at a coarser level, the probability of having normals pointing in opposite directions increases. The 3D Gaussian density function is characterized by using nine numbers, three for the mean direction and six for the matrix of second moments, that combine linearly. Following these works, [Tok05] proposed a very compact representation for isotropic roughness allowing a fast GPU implementation. This representation relies on only one single non-normalized mean direction vector generally computed by MIP-mapping a simple normal map. The length of this mean vector is used as a simple consistency measure of surface normal directions and is used to estimate the standard deviation of the Gaussian distribution. All these representations have the advantage of being linearly interpolable and to allow linear MIP-maping

A single Gaussian lobe is often insufficient to model complex normal distributions at coarse resolutions and a Gaussian mixture model (GMM) may be employed to model such distributions with more precision. Early work on GMMs of NDFs representation has been done by [Fou92a, Fou92b] which rely on a set of Phong lobes very similar to Gaussian lobes, and explains how to filter this representation. It relies on a basis with a large number of lobes to ensure a precise reconstruction but uses a costly non-linear least-squares optimization to fit lobes. In addition, efficient MIP-mapping of these highly-sampled distribution-based parameters is a challenging problem. [TLQ^{+}] enhance this model by proposing a prefiltering technique that allows real-time MIP-map construction as well as a representation that allows linear interpolation and the use of hardware filtering. Han et al. [HSRG07] have shown that, for a large class of common BRDFs, normal map filtering can be formalized as a spherical convolution of the *normal distribution function* (NDF) and the BRDF. This formalism explains many previous filtering techniques as special cases. In this method, the NDF is encoded either using *spherical harmonics* (SH) for low frequency materials or *von Mises-Fisher* (vMF) distributions when high frequencies are present. Unlike previous techniques, the SH representation makes all operations (interpolation and MIP-mapping) linear, and no nonlinear fitting is required. In this representation the BRDF is well decoupled from the NDF, enabling simultaneous changes of BRDF, lighting and viewpoint. Spherical harmonics are good for low frequency distributions, but are impractical for higher-frequency distributions due to the large number of coefficients required. The vMF representation is a better fit for high frequency materials, but requires an offline optimization similarly to [Fou92b]. It can create coherent lobes allowing hardware interpolation but require a complex reconstruction of the distribution from this vMF representation.

2.2 Volume representations in Computer Graphics

2.2.1 Adding details and realistic appearance

Full voxel grids have been used in many applications to benefit from the visual complexity permited by volumes for, e.g., fur [KK89], vegetation [DN04, MN00] (cf. Figure 2.3), or pseudo-surfaces [Ney98]. It is true, however, 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, JMW07], relief textures [OBM00], bidirectional textures [TZL+02], and hypertextures [PH89]. Volumetric textures and shell maps [KK89, Ney98, PBFJ05] rely on a tiling of a volume pattern within the layer. Even if our model allows instancing, our primary focus is to allow very large voxel representations containing no self-similar patterns, but instead a lot of different user-generated or automatically-generated volume details. In addition, most of these methods do not explicitly store volume data but rely on implicit procedural representation, while we want to display arbitrary volume details possibly authored.

The family of relief map approaches [KTI+01, BT04, OBM00, Wil05, BD06b, BD06a] fake or simulate the disparity below a surface using a 2.5D representation. Some recent extensions of relief maps are indeed ray-marchers [BD06b] and rely on optimization structures [CS94] to accelerate empty space traversal. However, we are interested in fully volumetric details and not only displaced ones.

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 and the filtering problem is not addressed. Instead, we will deal with general volumes and we will see how to filter them.

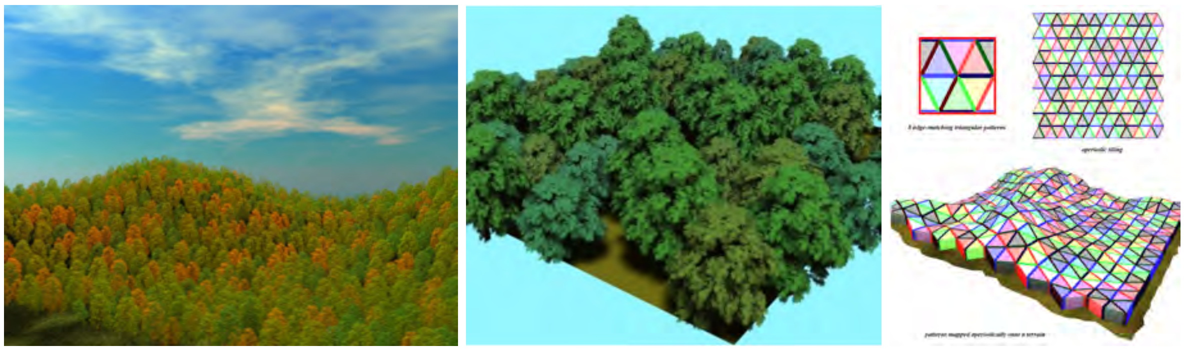


Figure 2.3. Example of forest rendering using volumetric tiles. *Source: [DN04]*

General volumetric data sets also often consist of dense clusters in free space 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.

Because of the very detailed and realistic appearance that arises from the use of voxels, much effort was spent on accelerating volume rendering. Many efficient methods assume that the entire volume is present in the graphics card 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.

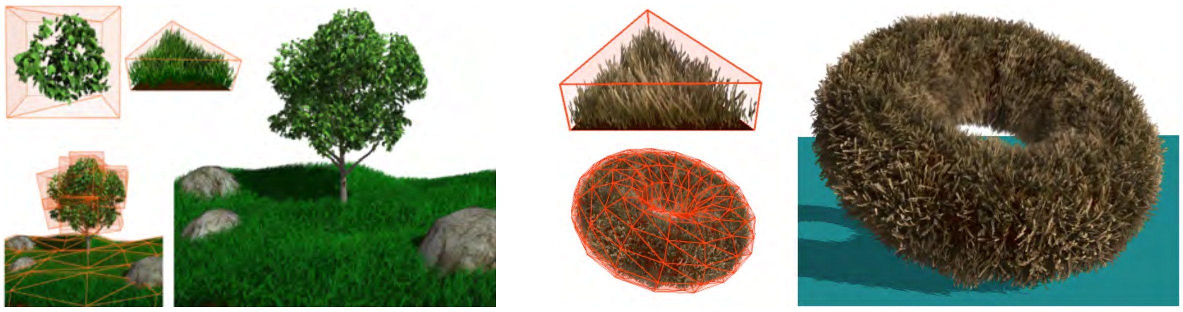


Figure 2.4. Example of deformable volumetric representations used to render natural elements (left, trees and grass) as well as hyper-texture like surface volumetric enhancement (fur, right).*Source: [DN09]*

2.2.2 Octree textures : sparse solid-texturing

Parameterizing a model in texture space for 2D texture mapping can be a very difficult problem on surfaces with no natural parametrization. Distortions and seams are often introduced by this difficult process, particularly with representations such as subdivision surfaces, implicit surfaces or dense polygonal meshes. Plenty of methods have been developed to provide automatic mapping but their description is beyond the scope of this thesis [AMHH08].

To overcome this problem, solid texture mapping was introduced in 1985 independently by Perlin [Pea85] and Peachey [Per85a]. The idea of solid texture mapping is to allow parametrization-free texturing by directly using the surface's position in 3D space as texture coordinates in a 3D (volume) texture (cf. Figure 2.5). This approach was mainly designed for procedural texturing, with texels generated in a volume enclosing the surfaces being rendered.

When using explicitly stored solid textures, one of the main problems is the huge amount of memory required for storing high resolutions. Indeed, traditional volume textures are stored in a regular grid of voxels, and their memory usage grows with the cube of the resolution. This has made them impractical for general use in computer graphics and confined their use to medical and scientific visualization (cf. section 2.3).

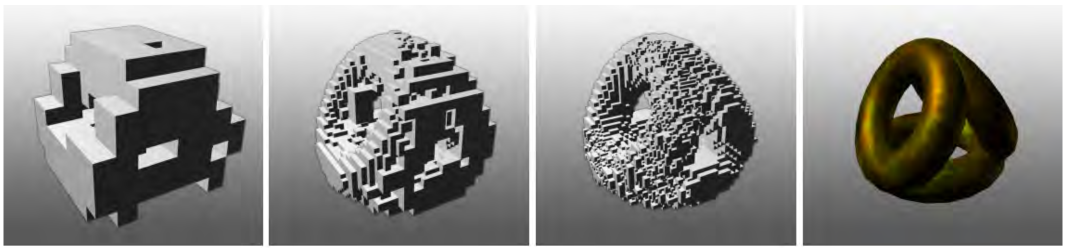


Figure 2.5. Example of an octree texture used for solid texturing of objects without parametrization. *Source: [DGPR02]*

In this context, Benson and Davis [BD02] and DeBry et al. [DGPR02] independently developed "octree textures", an octree-based representation to encode solid textures in a compressed form, storing only the subset of the volume that actually intersects the surface of a model. Octree textures are based on the idea of extending traditional 2D texture MIP-mapping [Wil83] to 3D MIP-mapping. Higher resolution texel values intersecting the surface of the model are stored in the leaves of the octree, while averaged versions of them are stored in the inner nodes, corresponding to higher MIP-map levels. When texturing an object rendered at a distance, a minification filter is used on texture data to prevent aliasing. In the simple case of isotropic filtering, this filter is simply an average made over a rectangle in a 2D texture's parameter space. The size of this filter depends on the distance to the object and determines which MIP-map level is used to get the required average. The same idea is used when sampling inside an octree texture, with the size of the filter used to select the octree level. Continuous interpolation is ensured between regions at different levels of detail using a subdivision strategy that provides enough texels to apply the usual interpolation scheme at all levels.

Lefebvre et al. [LHN05a] proposed a GPU implementation of octree textures. The octree is directly stored in texture memory with a pointer-based representation to link the tree nodes together. At this time, the GPUs did not allow us to index arbitrary memory from a shader and all the structure had to be implemented as color texels inside a 3D texture. This texture-based structure is illustrated in Figure 2.6. In this representation, each node is encoded as a $2 \times 2 \times 2$ array of RGBA texels called indirection grids, inside a 3D texture called the indirection pool. Each texel represents either a pointer to an indirection grid of sub-nodes, or the color content of the leaves directly stored as an RGB value. Pointers are encoded as RGB values representing 3D coordinates inside the indirection pool texture. The Alpha component of each texel is used to distinguish between a pointer to a child and the content of a leaf. Lefebvre et al. also describes an efficient point localisation scheme inside this octree structure based on a top-down traversal of the tree that allow logarithmic complexity lookups. Trilinear interpolation of texel data is handled similarly to [BD02] by over-subdividing the tree to ensure that all of the samples involved in the trilinear interpolation are included in the tree. 3D MIP maps data inside the inner nodes of the structure are encoded using a separate 3D texture containing a color information for each $2 \times 2 \times 2$ array of texels.

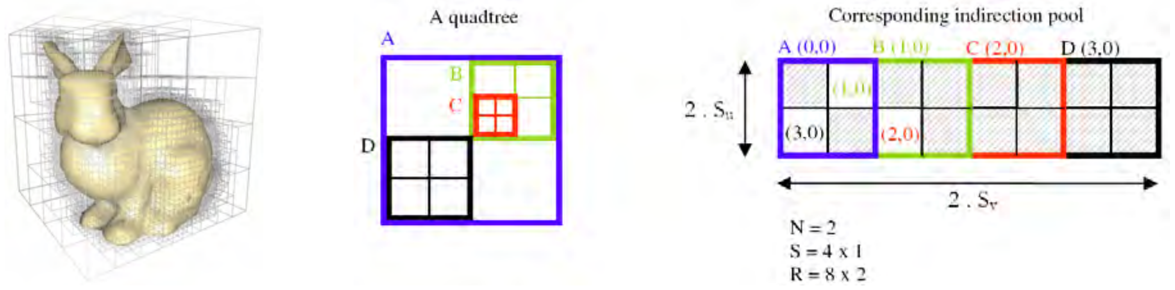


Figure 2.6. Illustration of the storage in texture memory of the GPU octree texture structure from [LHN05a]. The indirection pool encodes the tree, indirection grids are drawn with different colors and the grey cells contain data. *Source: [LHN05a]*

We will take inspiration from this representation for the design of our GPU data structure described in Section 5. However in our context, one of the main weakness of the octree texture representation described in the previous section is their poor performance when sampling trilinearly interpolated values. Indeed, trilinear interpolation of the samples taken inside the volume representation is fundamental to provide a high quality rendering (cf. section 6.1.5). Octree texture do not provides a good memory coherency and do not allow the use of hardware accelerated filtered texture sampling (cf. section 1.3).

2.2.3 Brick maps

In the context of offline global illumination computation on the CPU, Christensen and Batali [CB04a] introduced the "brick map", a tiled 3D MIP-map representation for volume and surface data using a hierarchical bricking scheme (cf. Section 2.4.4). Brick maps combine the idea of octree textures (cf. Section 2.2.2), with a tiling scheme that stores voxels inside constant size bricks (small regular grids, typically 16^3 voxels). Bricks are associated to each octree node, instead of storing them individually directly inside the octree. Such tiling enables an efficient caching of the voxel data that are stored coherently in memory, while providing empty space compaction.

Christensen and Batali [CB04a] described a caching scheme for their brick map structure used between the hard disk and the system memory. This scheme is only dedicated to the caching of the bricks, while the octree structure itself is read once from disk and kept in memory. For the bricks, their cache use a least-recently-used (LRU) replacement strategy managed entirely on the CPU, since all rendering operations are also performed on the CPU.

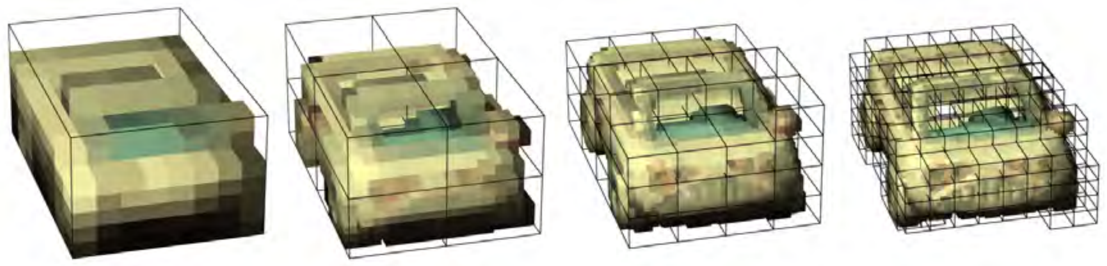


Figure 2.7. Illustration of the brick map structure at multiple resolution used to stored irradiance values for a car model. *Source: [CB04a]*

In our context, using a mixed structure like brick maps, combining an octree-based acceleration structure and coherent tiles of voxels, presents several advantages. First, similarly to the brick maps in the context of system memory caching, the bricks represent interesting candidates for our GPU-based caching scheme as coherent units of data that are easy to manipulate and manage for caching in video memory. In addition, as we will see later, such regular grids of voxels allows us to rely on the hardware texturing features of the GPU to perform fast trilinear interpolation and to benefit from a the 3D caching scheme during sampling. Our structure shares similarities with brick maps, but is dynamic, and, as a consequence, its content is view-dependent (through LOD (level-of-detail) and visibility). In addition, our structure is adapted to a compact storage in video memory (Sec. 1.3), and is designed to be efficiently accessed and traversed on the GPU. In contrast to brick maps, our typical use case in video game requires a much more aggressive caching scheme not restricted to the bricks but also applied on the octree structure itself.

2.3 Volume rendering for scientific visualization

For years, volume visualization has been mainly used to visualize simulated or acquired 3D datasets in the domain of scientific visualization. For a complete survey of GPU volume rendering approaches in this context, we refer the reader to the excellent book of Engel et al [EHK+06]. Here, we will limit our discussion to the structure and algorithms most related to our work. A detailed overview of volume rendering methods used in the special effects industry can be found in [WBZC+10].

In the context of scientific visualization, volume data usually represent scalar or vector values transformed into opacity and shading parameters during rendering. This operation is usually done using a transfer function that maps scalar density to optical coefficients. Volume rendering approaches are divided into two categories: indirect and direct volume rendering. Indirect volume rendering explicitly extracts geometric structures from volume data and renders such surface structures (meshes). On the other hand, direct volume rendering (DVR) directly renders volumes without extracting surfaces and that is the approach we are interested in.

In this section, we will first quickly explain the light transfer theory behind direct volume rendering, and then present the most relevant rendering methods.

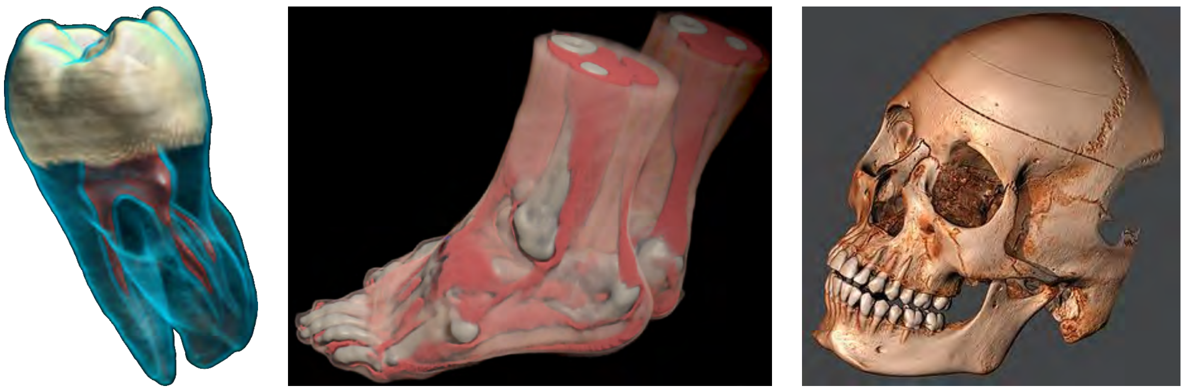


Figure 2.8. Examples of volume rendering results for scientific visualization. *Sources: [Had02], [Sch05],* *http://www.thefullwiki.org/Volume_rendering*

2.3.1 Physics of volume light transport

In order to render our volumetric geometry representation, we rely on the previous work on direct volume rendering [EHK+06, HHS93]. The theory of direct volume rendering models light transport into participating medium. It is based on *geometrical optics* that assumes that light propagates along straight lines when there is no interaction with matter, in contrast to *physical optics* that considers the wave character of light and its two possible states of polarization. With the approximations of *geometrical optics*, light does not interact with itself and the interaction of light with surfaces and volume elements can be described within the framework of linear light transport theory [CZP68]. We will discuss the basis of this theory in the next section and see how to solve the *equation of light transfer* that is the central equation of light transport theory.

Basis of light transfer

The light energy is described by its *radiance* $I(\mathbf{x}, \omega)$, also called *specific intensity*. It describes the radiation field at any point $\mathbf{x}$, giving the light direction ω. It is expressed in $W \cdot sr^{-1} \cdot m^{-2}$ and is defined as:

$$I(\mathbf{x}, \omega) = \frac{dQ}{dA\, cos\theta\, d\Omega\, dt}, \tag{2.4}$$

with Q the radiant energy (in Joules), A the unit area (in m^2), θ the angle between light direction ω and the normal vector of the surface A, and Ω the solid angle (in steradians).

The radiance along a light ray is affected when it passes through a participating medium. This interaction between light and matter is modelled as three different types of interactions: *Emission*, *Absorption* and *Scattering* effects.

Emission describes the amount of radiative energy of light that is directly emitted by the participating media. *Absorption* is the amount of energy that is absorbed by the material. Finally *Scattering* describes the amount of energy that is scattered by the material, changing the direction of the propagation of the light. Scattering can both increase (in-scattering) and reduce (out-scattering) radiative energy along a light ray.

The equation for the transfer of light is obtained by combining these three effects:

$$\omega \cdot \nabla_x I(\mathbf{x}, \omega) = -\chi I(\mathbf{x}, \omega) + \eta, \tag{2.5}$$

where $\nabla_x I$ is the directional derivative (the gradient) expressed using the "nabla" operator $\nabla = (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$. The dot product between the light direction ω and the gradient of radiance $\nabla_x I$ describes the directional derivative taken along the light direction. The term $\chi(\mathbf{x}, \omega)$ is the total absorption coefficient. It is defined as the sum of $\kappa(\mathbf{x}, \omega)$, the true absorption coefficient, and $\sigma(\mathbf{x}, \omega)$ the *out-scattering* coefficient:

$$\chi = \kappa + \sigma, \tag{2.6}$$

The ratio σ/χ of scattering coefficient over the total absorption coefficient is the *albedo*. An albedo of one means that no absorption appears at all, this is the case of perfect scattering.

The term $\eta(\mathbf{x}, \omega)$ of the equation of light transfer is the total emission coefficient that is the sum of the true emission coefficient $q(\mathbf{x}, \omega)$ and the *in-scattering* coefficient $j(\mathbf{x}, \omega)$:

$$\eta = q + j \tag{2.7}$$

While κ, σ and q are optical material properties, the *in-scattering* coefficient j needs to be computed by integrating the contributions of all incoming light directions over the sphere:

$$j(\mathbf{x}, \omega) = \frac{1}{4\pi} \int_{\Omega} \sigma(\mathbf{x}, \omega') p(\mathbf{x}, \omega', \omega) I(\mathbf{x}, \omega') d\omega' \tag{2.8}$$

The contributions from incident lights $I(\mathbf{x}, \omega')$ are weighted by both the scattering coefficient q and a *phase function* $p(\mathbf{x}, \omega', \omega)$ that describes the probability of scattering from the incoming direction ω' to the accounted direction ω.

Since we usually consider light transfer along a single ray, we can rewrite the equation of light transfer when considering a parameter s along a line segment expressed by $\mathbf{x} = \mathbf{p} + s\omega$, with $\mathbf{p}$ being some arbitrary reference point:

$$\frac{dI(s)}{ds} = -\chi(s)I(s) + \eta(s) \tag{2.9}$$

Emission-Absorption optical model

For the following work, we will concentrate on the *emission-absorption* optical model [EHK⁺06, HHS93]. This model neglects scattering and indirect illumination effects and represents only local light emission and absorption. This is the most widely used model for volume rendering in scientific visualization applications. Within this model, the equation of the light transfer reduces to:

$$\frac{dI(s)}{ds} = -\kappa(s)I(s) + q(s) \tag{2.10}$$

This integro-differential equation can be solved transformed into a pure integral equation by solving it along the direction of the ray between a starting point $s = s_0$ and an end point $s = D$, relative to back-to-front traversal. Boundary conditions also need to be fixed and we define I_0 the initial radiance along the ray at $s = s_0$. This leads to the *volume rendering integral* [EHK+06] :

$$I(D) = I_0\, e^{-\tau(s_0,D)} + \int_{s_0}^{D} q(s)\, e^{-\tau(s,D)}\, ds \tag{2.11}$$

The term $\tau(s_1, s_2)$ is called the *optical depth* or *optical thickness* between position s_1 and s_2 and is defined as a measure of the proportion of radiation absorbed or scattered along a path through a partially transparent medium. It is defined as:

$$\tau(s_1, s_2) = \int_{s_1}^{s_2} \kappa(t)\, dt \tag{2.12}$$

The corresponding transparency $T(s_1, s_2)$ (in [0, 1]) is defined as :

$$T(s_1, s_2) = e^{-\tau(s_1,s_2)} \tag{2.13}$$

This leads to the following simplified version of the volume rendering integral:

$$I(D) = I_0\, T(s_0, D) + \int_{s_0}^{D} q(s)\, T(s, D)\, ds \tag{2.14}$$

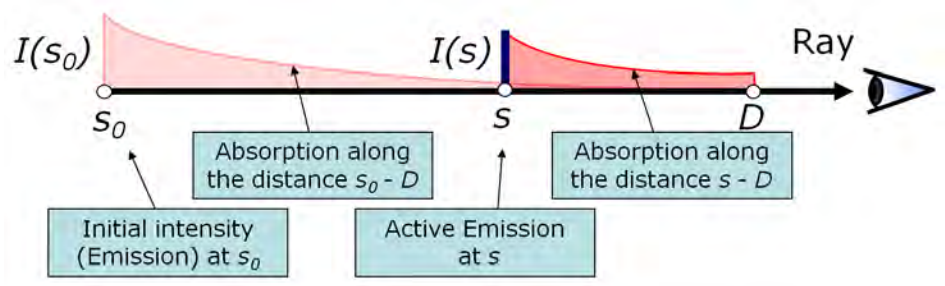

$I(s_0)$
$I(s)$
Ray
s_0
s
D
Absorption along the distance s_0 - D
Absorption along the distance s - D
Initial intensity (Emission) at s_0
Active Emission at s

Figure 2.9. Illustration of the volume rendering integration between $_0$ and D. Source [Had02]

Evaluating the volume rendering integral

In the general case, the volume rendering integral 2.14 can not be evaluated analytically. This integration is usually approximated by a Riemann sum, splitting the integration domain into n subsequent intervals described by locations $s_0 < s_1 < ... < s_n$, and not necessarily with equal lengths. Thus, based on equation 2.14, the radiance at a given location s_i can be evaluated as:

$$I(s_i) = I(s_{i-1})\,T(s_{i-1}, s_i) + \int\limits_{s_{i-1}}^{s_i} q(s)\,T(s, s_i)\,ds \tag{2.15}$$

This function is simplified by introducing the term $T_i = T(s_{i-1}, s_i)$ the transparency of the i^{th} interval and $c_i = \int_{s_{i-1}}^{s_i} q(s)\,T(s, s_i)\,ds$ the radiance contribution of this interval. Then, by recursively evaluating this function over all points of the interval, the discrete version of the volume rendering integral is obtained:

$$I(D) = \sum_{i=0}^{n}\left(c_i \prod_{j=i+1}^{n} T_j\right),\ with\ c_0 = I(s_0) \tag{2.16}$$

Please note that the radiance contribution c_i is wavelength dependent and is usually assimilated to a color contribution and so is the resulting integrated radiance $I(D)$. Finally, the transparency T_i and color contribution c_i need to be evaluated for each interval. This is generally done using a *piecewise-constant* or *piecewise-linear* approximation of the functions between two sampling points. With a piecewise constant approximation, the transparency T_i over a ith segment is approximated as:

$$T_i \;\approx\; e^{-\kappa(s_i)\Delta_i}\,, \tag{2.17}$$

with Δ_i the length of the ith interval. The radiance/color c_i is approximated as:

$$c_i \;\approx\; q(s_i)\Delta_i \tag{2.18}$$

Incremental evaluation

The discrete version of the volume rendering integral presented in equation 2.16 is evaluated incrementally, with either a *front-to-back* (starting from the eye) or a *back-to-front* compositing scheme. The *front-to-back* compositing scheme is the most usually used due to the fact that it allows to stop the evaluation prematurely when the accumulated transparency reaches zero and then no more radiance can be accumulated [KW03a].

The *front-to-back* evaluation can be expressed recursively based on equation 2.16:

$$\begin{aligned} I(s_i) &= I(s_{i+1}) + T(s_{i+1})\,c_i \\ T(s_i) &= T(s_{i+1})\,T_i \end{aligned} \tag{2.19}$$

width $I(s_n) = c_n$ and $T(s_n) = T_n$. This leads to the following *front-to-back* compositing scheme, with opacity α defined as $\alpha = 1 - T$:

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst})\,C_{src} \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst})\,\alpha_{src} \end{aligned} \tag{2.20}$$

2.3.2 Main volume rendering approaches

In order to apply the volume rendering integral we described in the previous section, three main classes of direct volume rendering approaches can be found in the literature.

Splatting The first and oldest family encompasses approaches based on the splatting of individual voxels using a reconstruction kernel (usually Gaussian) in the spirit of [Wes90]. The splatting approach only provides a poor approximation of the volume rendering integral presented in Section 2.3.1 and is not compatible with high quality rendering.

Texture slicing Texture slicing approaches are projective (object-order) approaches that were mainly designed to support graphics hardware acceleration. It has been the most widely used and most efficient class of approaches for years. They rely on the projection on screen of textured geometric slices of the volume as illustrated in Figure 2.10.

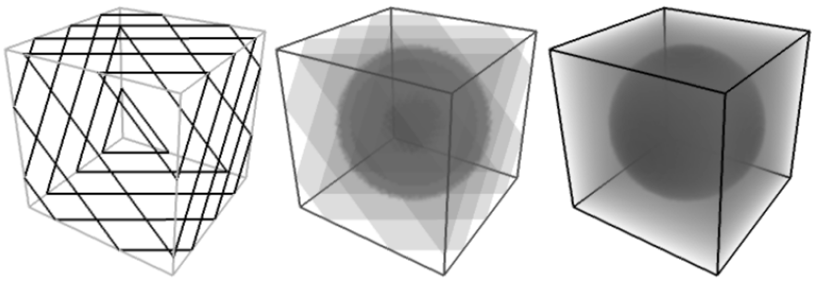


Figure 2.10. Illustration of the texture slicing approach for direct volume rendering. *Source: [WE98]*

The basic idea is to store the volume either in a set of 2D textures or into a 3D texture, and to rasterize multiple slices located into the 3D volume domain and used to sample the texture. The slices can be either aligned on the screen plane [WE98] or on volume axis [RSEB$^+$00], and sorted either front-to-back of back-to-front depending on the blending function used to accumulate fragments on the screen. The use of slicing with texture mapping for volume rendering was introduced by [CN94] and [CCF94]. This approach has since been revisited, fine-tuned and extended many times in the litterature, especially to follow graphics hardware evolution [GK96, RSEB$^+$00]. A lot of work has also been done on efficient transfer function [EKE01], or more complex illumination models [KPHE02]. But these approaches have many drawbacks. One of the main drawbacks is that they are restricted to uniform grids and are difficult to adapt for adaptive multi-resolution rendering. In addition, such approaches lack flexibility in the evaluation of the volume rendering integral (restricted to a frustum projection) and are highly fillrate (and therefore bandwidth) consuming. Indeed, texture fetch operations, shading operations and per-pixel blending operations are performed for a significant number of fragments that do not contribute to the final image because the approach does not allow an early termination of the evaluation in a per-pixel basis.

Volume ray-casting Volume ray-casting is a forward (image-order) method. The idea is to trace rays inside the volume data from the camera in order to evaluate directly the volume rendering integral as illustrated in Figure 2.13. With the evolution of the graphics hardware and since the advance of programmable shading, GPU based volume ray-casting is now the most popular method for volume rendering in interactive applications.

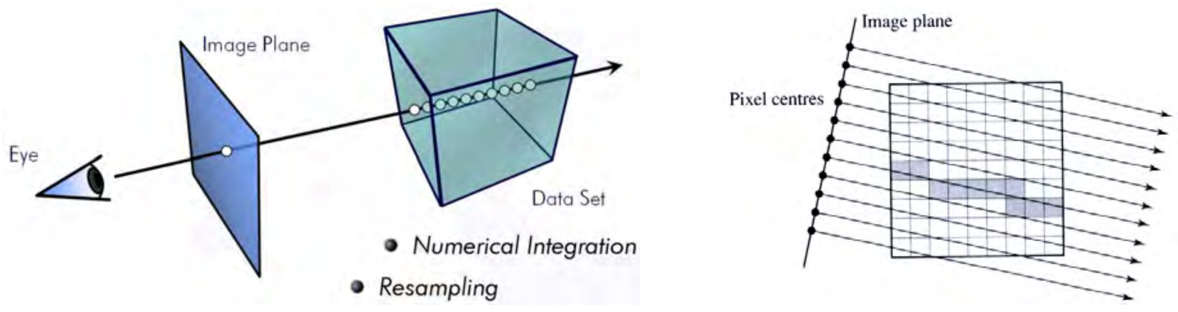


Figure 2.11. Illustration of volume raycasting for direct volume rendering. *Source: [EHK+06]*

Volume ray-casting, also called volume ray-tracing (if secondary rays are used), was first introduced by Levoy [Lev88] to display surfaces from binary volume data and improved by him [Lev90] for performance with CPU based-implementations. Levoy was followed by many others who enhanced the approach further for CPU implementations [DH92, YS93, FS97].

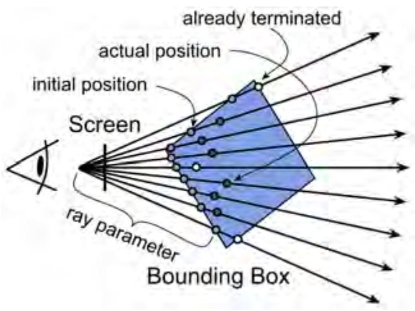


Figure 2.12. Illustration the ray-casting scheme where all viewing rays are processed simultaneously. *Source: [RGW+03]*

Krüger and Westermann [KW03a] were the first to bring volume ray-casting to real-time implementation on the GPU using the newly introduced pixel shaders 2.0 that provide flexible per-fragment texture fetches and arithmetic. At this time, the transition from texture slicing approaches (that were the standard for real-time volume rendering thanks to the acceleration by fixed pipeline graphics hardware) to volume ray-casting was driven by the evolution of commodity graphics hardware from fixed function pipelines to programmable ones that spawned completely new classes of graphics algorithms [PBMH02].

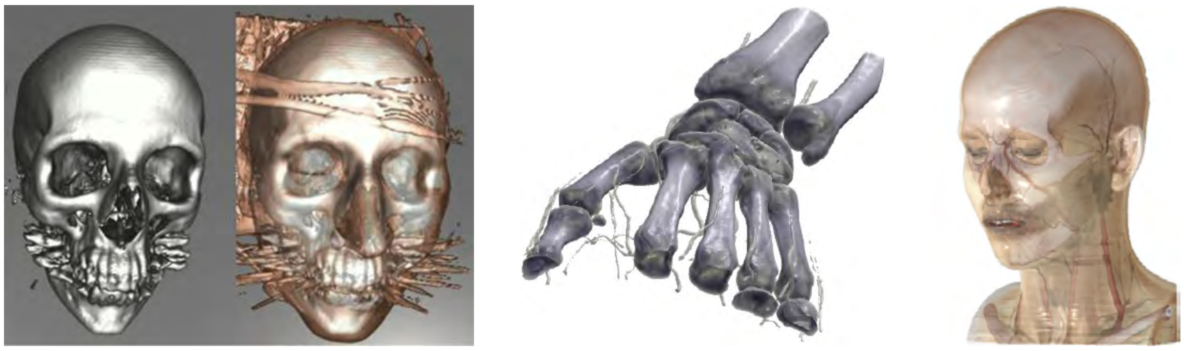


Figure 2.13. Examples of images rendered using volume ray-casting. *Sources: [KW03a, Sch05]*

In order to optimize bandwidth usage and computations, Krüger performs an early-ray termination to stop the evaluation of pixels that have already reached a saturated opacity. Due to the lack of conditional branching in pixel shaders model 2.0, a multi-pass algorithm is used in order to split the ray evaluation and allows using the early z-test mechanism of the GPU to prevent continuation of terminated rays. Krüger's approach was enhanced by Scharsach [Sch05] and Stegmaier et al. [SSKE05] who relied on a single-pass algorithm and performed early-ray termination, thanks to the conditional branching feature and additional flexibility of the pixel shaders model 3.0. Scharsach proposed a way to enable fly-through applications, making it possible to place the viewpoint into the dataset, and to correctly intersect the rendered dataset with normal OpenGL geometry. He also proposed a simple scheme for empty-space skipping based on a 2-way blocking scheme, subdividing the volume into constant size blocks stored into a 3D texture and referenced from a second indirection texture.

Volume ray-casting is now the fastest approach to render volume data, it offers a great flexibility in the way the volume rendering integral is evaluated, and is not restricted to a view frustum. It allows arbitrary evaluation direction and per-ray control as well as secondary rays evaluation. That is why we chose to rely on a ray-casting approach for our rendering algorithm, combined with a spatial data structure as detailed in Chapter 6.

2.4 Managing the complexity: Representations and algorithms

Interactive visualization of massive models has been a challenging problem and a very active area for years, especially in the domain of scientific visualization where massive datasets need to be visualized [GKY08]. In the context of massive datasets visualization, efficient data access and data management are the keys for providing interactive visualization. In addition, as explained in Section 1.3.5, these key elements become especially critical in the current context of a limited improvement of memory accesses performance of high performance GPU architectures compared to processing power.

The choice of the rendering algorithm (Sec. 2.4.1) as well as the data structure (Sec. 2.4.2) plays an important role in the scalability of the visualization approach. In addition, visibility culling (Sec. 2.4.3) and view-dependent multi-resolution (Sec. 2.4.4) are key techniques to optimize data access.

2.4.1 Rendering

Any rendering operation is relative to a sorting problem for each pixel of the final image, which consists of determining what part of the rendered model is closer to the viewer. The various approaches of rendering vary in the order in which this sorting operation is performed, this can be image-order, or object-order. There are basically two classes of rendering algorithms that can be used for real-time applications:

- Rasterization with z-buffer that performs object-order visibility determination by traversing objects and scene primitives (usually polygons), projecting and rasterizing them on the screen.
- Ray-tracing that performs image-order visibility determination by computing ray-primitive intersections for each pixel.

Although the ray tracing and rasterization fields have independently developed their own approaches in the past, the underlying issues faced when dealing with massive models are somewhat similar, and state-of-the-art systems are converging towards applying similar solutions.

Rasterization In terms of memory access efficiency, rasterization theoretically offers better data access locality and cache coherency. Per-batch rendering attributes can be shared between primitives and per-primitive attributes can be read sequentially. In addition, transform and shading computations are very coherent inside batches of primitives, ensuring an efficient parallelization of computations. This explains the success of rasterization for real-time rendering applications, and its native implementation inside graphics hardware. However, rasterization-based approaches are limited to visibility determination inside a view frustum. In addition, object-order approach with no special data structure provides a linear time complexity in the number of scene primitives. All data have to be read in order to be visibility tested, which stays reasonable for relatively small scenes, but becomes prohibitive for very large datasets.

Interactive ray-tracing On the other hand, ray-tracing algorithms [Whi80, CPC84] support arbitrary point-to-point visibility determination that allows easier computation of global lighting effects. In their most basic form, ray-tracing techniques are also limited to linear time complexity relative to the number of primitives. Usually, the term ray-casting is used to refer to once-bound ray-tracing, in the context of of volume rendering (Sec. 2.3.2) or when it is used only for visibility computations. Ray-tracing usually designate Whitted's [Whi80] algorithm, with multiple bounces used to compute advanced global lighting effects.

2.4.2 Spatial data structures

In order to enable rendering in sub-linear time complexity, spatial index structures must be employed. Such scene structuring is based on a tree structure that provides a fast random access to the scene geometry and allows visibility queries to be answered in logarithmic time. Thanks to such structures, only the parts of the scene that are visible need to be accessed. Spatial structure can be employed to accelerate both rasterization and ray-tracing rendering algortithms. The main difference is the granularity at which they can act. With ray-tracing, such structure allows per-ray visibility detection and ordered traversal of objects data. With rasterisation, a much larger granularity has to be used (usually screen granularity) in order to preserve object-order efficient GPU rendering. Reducing this granularity is difficult with today's rasterization hardware due to the fact that rasterization commands have to be emitted from the CPU, preventing an efficient parallel traversal of the structure on the GPU.

Many ray-tracing systems based on various spatial data structures have been designed to display massive datasets and such approaches start to be efficiently implemented on today's GPUs (thanks to their generic computing capabilities, cf. section 1.3.2). A comprehensive survey, analysis and comparison of acceleration structures for CPU ray-tracing applications can be found in Havran's PhD thesis [Hav00]. For the state of the art in interactive ray tracing, we refer the reader to Wald and Slusallek [WMG+07]. A large variety of special data structures has been used to accelerate the intersection tests of ray tracing, like regular grids [CN94, PBMH02, Pur04], BVH, KD-Trees [EVG04, FS05, HSHH07]. A comparison of acceleration structures for GPU ray-tracing that covers uniform grid, Kd-tree and BVH (Bounding Volume Hierarchy) can be found in Niels Thrane and Lars Ole Simonsen Master's thesis [TS05]. Each structure has its own advantages depending on the context in which they are used.

Complex data structures on the GPU

The main difficulty when working on the GPU is to efficiently implement such a data structure so that it can be quickly accessed and provides a good usage of the GPU resources, in particular of the texture hardware units. Lefebvre et al. [LHN05a, LD07] proposed a GPU implementation for octrees directly stored in texture memory with a pointer-based representation. Lefebvre and Hoppe [LH06] proposed to rely on a multidimensional hash function evaluated on the GPU to encode sparse 2D and 3D images. Lefohn at al. [LKS+06] proposed a common framework to implement various hierarchical data structures on the GPU through a virtual memory abstraction (Fig. 2.14). These work were interesting and defined the foundation for efficient GPU implementation of complex data structures. However, their implementation now tends to become obsolete with the introduction of the *compute* (CUDA) paradigm to address the GPU that provides more flexible and more standard memory access (Sec. 1.3.2).

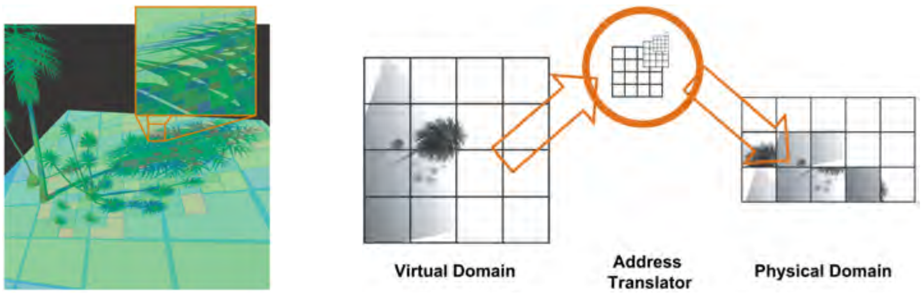


Figure 2.14. Illustration of the three components of the multiresolution "adaptive page table" structure from [LKS+06] used to represent a quadtree for an adaptive shadow mapping example. *Source: [LKS+06]*

Volume compression and empty space skipping

In the context of volume rendering, in addition to the fast random access, spatial data structures provide data compression through empty space skipping. Many decomposition schemes have been proposed in order to reduce the percentage of empty voxels represented in the texture memory. Early volume rendering papers like [SFH97] and [TWTT99] proposed to use an octree to deal with large datasets by skipping empty regions of the volume. Empty regions are detected during preprocessing and only non-void regions are loaded into texture memory as leaves of the octree structure. An octree is a hierarchical binary decomposition of 3D-space along its component axes [Kno08]. Octrees provide a regular subdivision of space with non-overlapping node spacing, thus are well suited to store rectilinear voxel data. As detailed in section 2.4.4, octrees also allow data to be stored with adaptive levels of resolution. As we will see in chapter 5, we chose to base our GPU data structure on an octree in order to take advantage of these structuring and compression properties.

2.4.3 Visibility culling

The first obvious way to deal with the complexity of the scene is to be able to detect and discard all the parts that are not visible from a given point of view. Visibility determination has been a fundamental problem in computer graphics since the very beginning. Determining visible elements is important for many different rendering paradigms, ranging from ray tracing [Wal04, WMG+07], to point clouds [WBB+07], or volume rendering (cf. Section 2.3). The first usage for visibility determination is *visibility culling* that aims at quickly rejecting invisible geometry before actual rendering is performed. This allows the reduction of the total amount of computation needed to render a scene in the context of object-order rendering. In the context of out-of-core rendering of massive scenes or objects, visibility determination is also a critical component used to prevent loading in memory all data that will not contribute to the final image.

Traditionally, visibility culling is split into three tasks illustrated in Figure 2.15:

- **Backface culling** Back-face culling applies only to surface-based rendering and avoids rendering primitives that do not face the camera.
- **View-frustum culling** Viewing-frustum culling avoid rendering primitives located outside from the view frustum. Many hierarchical techniques based on hierarchical space partitioning structures (Sec. 2.4.1) have been developed to speed-up the frustum culling process [BEW+98, AM00, Cla76].
- **Occlusion culling** The most complex and most difficult task is in fact occlusion culling. Occlusion culling aims to avoid rendering primitives that are occluded by some other part of the scene. This is also the most studied visibility problem and that is usually the type of culling designated by the term "visibility culling".

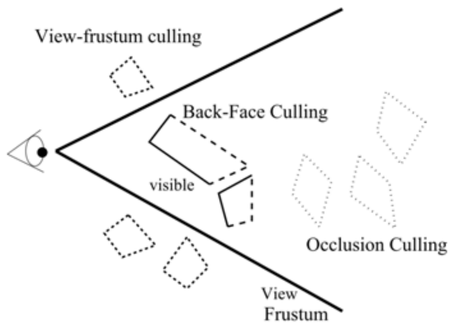


Figure 2.15. Illustration of the three types of visibility culling tasks: view frustum culling, back-face culling and occlusion culling. *Source: [COCSD03b]*

Occlusion culling

Conservative prediction of visibility in scenes is an important and difficult topic in Computer Graphics. A survey of this domain is beyond the scope of this thesis. A comprehensive survey of occlusion culling approaches was published by Cohen-Or et al. [COCSD03b] and another survey from Bittner and Wonka [BW03] discusses visibility culling in the context of more general visibility problems.

Visibility determination approaches are generally classified into from-point and from-region visibility algorithms. From-region algorithms rely on the precomputation of a potentially visible set (PVS) for points in cells of a fixed subdivision of the scene and are processed in an offline preprocessing step. These preprocessing algorithms have no runtime overhead but are hard to compute accurately for general environments and only work for static scenes. For aggressive visibility sampling [NB04], ray-tracing can be a good option to interactively compute from-region visibility [BMW$^+$09]. On the other hand, from-point algorithms are computed online for each particular viewpoint and allow for fully dynamic scenes. Most online occlusion culling algorithms work in image space, usually using rasterization. Recent image-based occlusion culling algorithms exploit graphics hardware built-in occlusion query mechanism to perform online visibility culling [KS01]. Before the integration of such dedicated hardware support, software online occlusion culling was mostly considered too costly to be used. There are some worthy exceptions such as Hierarchical Occlusion Maps from Zhang et al. [ZMHH97] or the dPVS portable framework from Aila et al. [AM04].

Hardware occlusion queries

Hardware occlusion queries are relatively lightweight operations that return the number of pixels that passed the z-buffer test during the rasterization of a proxy geometry, without having to read back the entire frame buffer. Their main advantage is their generality and speed. However, hardware occlusion queries require read-back of information from the graphics card, and due to the long graphics pipeline this introduce a high latency when the application waits for the query to return. In order to reduce the overhead and latency of hardware occlusion queries and to optimize their scheduling, spatio-temporal coherence is exploited with coherent hierarchical culling (CHC) [BWPP04, GBK06, CBWR07, MBW08]. Most of these techniques rely on a hierarchical scene structuring traversed in front-to-back order with a clever interleaving in order to reduce the number query issued and to mask latency overhead as much as possible.

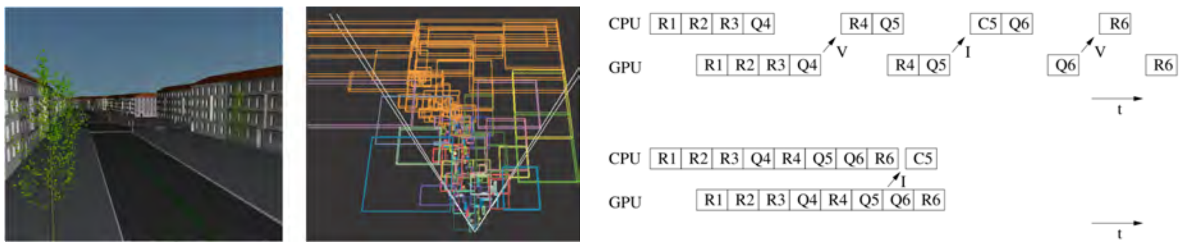


Figure 2.16. *Left:* A sample view point in a city scene with scene subdivision and all state changes required by a CHC algorithm [BWPP04] illustrated with different colors. *Right:* Illustration of CPU stalls and GPU starvation when interleaving occlusion queries with rendering (top). Qn : querying, Rn : rendering, Cn : culling. More efficient query scheduling from [BWPP04] (Bottom). *Sources: [MBW08, BWPP04]*

Gobbetti et al. exploited such hardware occlusion queries in the context of visualization of large scenes with a mixed volume-surface representation [GM05], and more recently full volume datasets rendering [GMAG08]. They rely on optimized occlusion queries [BWPP04] based on a screen partitioning to detect visible parts of a tree-based space subdivision structure maintained off-core. Query results are used to trigger loading of data for required parts of the scene.

But even when interleaved intelligently, hardware occlusion queries are complex to efficiently put into place and still result in significant overheads. Additional costs come mainly from the GPU-CPU synchronization induced by this approach and the serial processing of query results that must be done on the CPU. They also come from rendering steps of proxy geometry that have to be interleaved with normal scene rendering. In addition to being costly, occlusion query based visibility can only provide a rough granularity resulting in an highly over-conservative visibility approximation.

We experimented such an approach in early work on our voxel rendering model [CN07], but we quickly moved it further apart due to its poor performance in our context and the low granularity in visibility tests. Instead, as we will see in Chapter 7, we turned toward a solution based on direct tracking of visibility during rendering, that allows us to directly exploit the visibility determination happening during ray-tracing.

2.4.4 Multiresolution rendering approaches for volume rendering

The issue of large volumes has been an important issue since the early usage of volume rendering, even before GPU implementations. Compression inside a sparse data structure allows a significant reduction of the amount of volumetric data to manipulate. However, even compressed inside a sparse data structure, the volume datasets we are interested in still represent gigabytes of memory and do not fit inside the video memory of the graphics card for rendering (cf. Section 1.3).

The idea of multiresolution approaches for volume rendering is to adapt the rendering resolution to the distance to the viewer and data homogeneity. To do so, they break down a single large volume into several smaller ones called bricks and rely on a spatial hierarchy with recursive subdivision and increasing resolution (usually an octree) to index these bricks. The hierarchy is built for the data set in a pre-processing step and each node of the spatial hierarchy contains a certain part of the volume within its bounding box at a specific resolution.

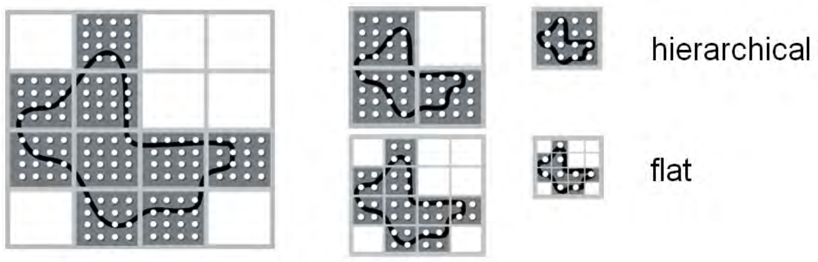


Figure 2.17. Illustration of the difference between flat and hierarchical bricking schemes. Source [BHMF08]

As shown in Figure 2.17, either the resolution of the spatial structure or the resolution of the bricks can be adapted [BHMF08]. Adapting the resolution of the octree leads to *hierarchical bricking schemes*, while adapting the resolution of the bricks leads to *flat bricking schemes* with a constant spatial subdivision. Thus, flat bricking schemes do not allow refining only non-empty regions and are limited to relatively low resolution datasets.

As for any rendering approach, multiresolution rendering approaches can be either object-order or image-order. Object-order approaches are projective approaches that traverse and render sub-parts of the volume independently. Image-order approaches are ray-casting approaches that start from the view and traverse the structure per pixel in order to compute the volume rendering integral.

Early texture slicing based approaches

LaMar et al. [LHJ99] proposed the first multiresolution scheme based on an octree structure segmenting the dataset into a set of block tiles (also called bricks) at different resolution levels. These different resolution levels are kept in memory and blocks are chosen according to a view-dependent criteria for rendering. Rendering is done adaptively by traversing the structure on the CPU and selecting block tiles to be transferred in video memory and rendered one after the other. High resolution tiles are selected close to the viewer, and low resolution away from it. Block tiles are rendered independently using a slicing approach(cf. Section 2.3.2) and results are combined in the framebuffer. Boada et al. [BNS01] presented a similar approach with the same data structure but introduced the idea of using a frequency-based level-of-detail selection based on a data homogeneity measure when building the set of octree nodes to be stored for rendering (the *cut*). In these approaches, both the octree depth and the resolution of the bricks can be adapted and chosen for rendering based on various importance criteria. Unfortunately, they do not provide totally artifact-free rendering and spatial continuous transitions and interpolation between resolutions levels is not ensured.

Weiler et al. [WWH+00] improve on LaMar et al. [LHJ99] by explicitly addressing the avoidance of interpolation errors and discontinuity artifacts between bricks of the same or different levels-of-detail inside a given cut. Continuous interpolation between adjacent bricks at the same level is ensured by duplicating and sharing boundary voxels of adjacent bricks. Boundaries between bricks of different resolution are managed by copying and duplicating coarser values of an adjacent brick in the boundary voxels of the finer brick. This scheme restricts brick transitions to differ by at most one level in order to maintain the continuity between levels (Fig. 2.18). This problem of inter-bricks interpolation was tackled by the following research that proposed improved interpolation schemes [LLY06, BHMF08].

Even if they provide spatial inter-blocks interpolation inside a given set of bricks, these techniques do not ensure a continuous temporal transition between different levels-of-details. No interpolation is done between LODs (quadrilinear filtering). Thus, transitions from one resolution to another in a given region is not continuous, leading to popping artifacts. Our scheme provides a smooth transition between resolutions (cf. chapter 6).

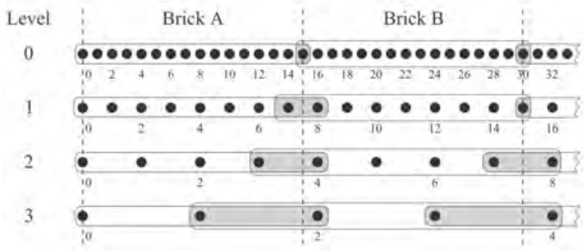


Figure 2.18. Illustration of the block tile storage with border sharing used by [WWH+00]

In order to reduce even more the amount of data stored in system memory, [GWGS02] proposed a compressed hierarchical wavelet representation for the volume data generated in a preprocessing step and stored on disk. This wavelet representation is decompressed on-the-fly on the CPU during rendering in order to produce regular voxel grids transferred in texture memory and rendered using slicing (cf. Figure 2.19). It was one of the first papers to demonstrate interactive exploration of a large dataset on a conventional PC with a graphics card.

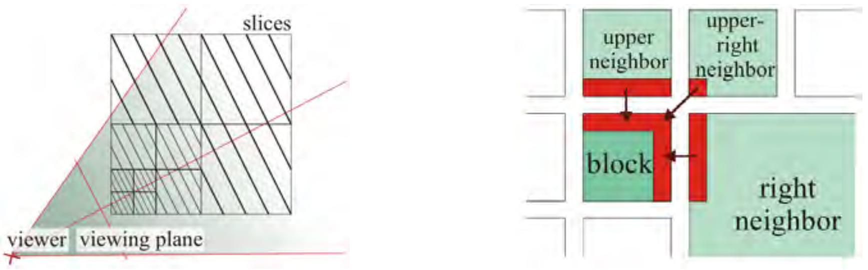


Figure 2.19. Left: Multi-resolution rendering with view-plane aligned slices. Right: Copy of adjacent voxels at different resolution into 3d-texture blocks for correct interpolation. *Source: [GWGS02]*

This approach was improved in [GS04] that introduces a method for coarse early termination of the rendering evaluation. It is based on occlusion tests that prevent occluded data to be loaded and rendered. An approximated occlusion calculation is done prior to the rendering. It is computed using an approximated occlusion map at a fixed resolution built using a software raycaster based on cell projection and a uniform conservative (minimum) opacity for each block. 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.

Object-order GPU volume-raycasting based approaches

All multiresolution approaches presented previously relied on a texture slicing approach (cf. Section 2.3.2) to render sub-blocks of the original volume. Slice-based hierarchical volume rendering approaches make it possible to skip empty blocks from rendering and, to a limited extent, to exploit an opacity map for occlusion culling. However, these approaches are rasterization limited, hard to optimize algorithmically and suffers from precision problems. When the graphics hardware became flexible enough to allow it, multiresolution techniques based on volume ray-casting (cf. section 2.3.2) started to appear. Object-order volume raycasting approaches are very similar to hierarchical texture slicing based approaches but render each voxel tiles separately using volume raycasting on the GPU.

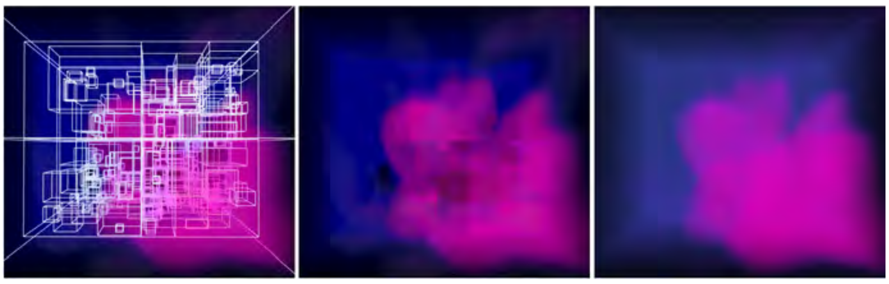


Figure 2.20. Comparison of [KWAH06] of the image quality of slice-based (middle) and GPU-assisted raycasting (right) approaches for rendering a cosmological AMR (Adaptive Mesh Refinement) dataset (left). Artifacts in the slice-based approach are due to insufficient framebuffer precision to correctly blend highly-transparent slices. *.Source: [KWAH06]*

Such approach was first proposed by Hong et al. [HQK05] using an octree structure and was extended to AMR[1] (Adaptive Mesh Refinement) datasets by Möller et al. [KWAH06]. Möller et al. compared both the rendering performance and the image quality between texture slicing and volume raycasing approaches. Image quality comparison is illustrated in Figure 2.20 on their test dataset.

The result for the texture slicing approach shows severe rendering artifacts in the refined regions due to insufficient framebuffer precision during blending on the hardware used at this time[2]. The ray-casting approach does not suffer from these artifacts since full 32bit floating point registers are used during the accumulation in a fragment shader. At this time, the performance of the GPU volume raycasting approach was only about 30% of the slice-based method. This performance penalty was mainly due to the low performance of dynamic loops and branches in the fragment shaders of the GPU at this time (NVIDIA NV40) and improved widely in later generations of hardware.

[1]Adaptive Mesh Refinement (AMR) is a popular technique in large scale CFD (Computational Fluid Dynamic) simulations. AMR data is organized in hierarchical grids with resolution refined adaptively in regions of interest or where the simulation requires it

[2]These tests were made on an NVIDIA NV40 GPU supporting floating point blending only on 16bits

Image-order GPU volume-raycasting based approaches

The transition to volume ray-casting approaches together with the improved flexibility of the graphics hardware (GPU) made possible the storage and the traversal of the hierarchical structure itself directly on the GPU. This approach makes it possible to traverse the structure in parallel, removing the bottlenecks that were previously created by the serial traversal on the CPU. Researches on such GPU based acceleration structures are described in Section 2.4.2. This transition to a GPU based parallel traversal of the acceleration structure constitutes an important step for GPU based volume rendering and that is the approach we decided to follow.

As far as we know, the only methods for volume ray-casting preceeding our work and relying on an acceleration structure stored and traversed on the GPU were proposed in [VSE06] for AMR data. In this paper, two techniques relying on two different GPU based data structures are presented. The first technique is based on the GPU implementation of octree textures [BD02] proposed by Lefebvre et al. [LHN05a]. The second relies on the flat "dynamic adaptive multi-resolution GPU data structure" proposed by Lefohn et al. [LKS$^+$06] and referred to as the "adaptive page table" structure. The octree-based method uses a sparse octree structure and stores individual voxels in the leaves of the octree texture, leading to a more difficult filtering scheme. The adaptive page table implementation relies on a full (not sparse) mipmap hierarchy of page tables to index non-empty tiles of voxels stored in texture memory, allowing hardware interpolation, but that tends to overrefine some parts of the scene. The comparison between these two structures concluded that the octree approach is notably more memory-efficient, but that the adaptive page table dramatically outperforms it in rendering speed, due to its better access complexity and the support of hardware filtering. In their test, the adaptive page table requires a three to four times larger memory for a 40 times increase in rendering performance.

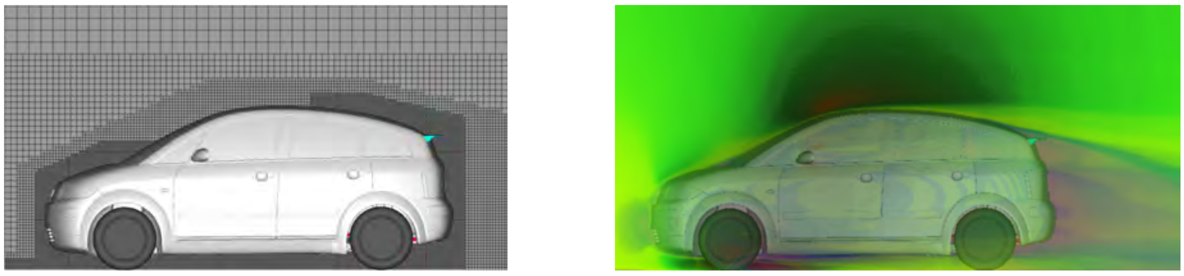


Figure 2.21. An AMR grid used for CFD simulation with refinement around the surface geometry [VSE06].

In our model, we will take the best of both approaches by combining an octree with tiles of regular grids (bricks) traversed on the GPU for volume-raycasting. Bricks allow the voxel data to be stored in a 3D texture. In this way, one can benefit from several hardware related advantages, like direct 3D addressing, trilinear interpolation, and a 3D coherent texture cache mechanism. The multiresolution structures presented previously are view-independent but do not provide continuous temporal transition between different subdivision levels. Therefore, discontinuities appear during the exploration of such datasets, whereas our approach adapts resolution continuously. It provides a complete 3D MIP-mapping scheme (with quadrilinear interpolation) with smooth temporal transitions to a different subdivision levels.

2.5 Out-of-core data management

Multiresolution approaches allow only a subset of the whole dataset selected for a given point of view to be present in memory for rendering. The reason why such rendering can still be efficient is that the part of a scene needed for rendering is often much smaller than the entire dataset. However, this subset can still represent a large amount of data in our context. Thus, transferring it entirely inside the video memory at each frame would not allow us to achieve real-time performance.

In order to reuse data loaded inside video memory from frame-to-frame, some sort of caching must be achieved. Cache systems can be used to turn temporal coherence to profit. Indeed, most data present in memory for a given frame are still visible in the next frame, and new data progressively appear. Taking advantage of this temporal coherence is the goal of our data management scheme (Sec. 7.3) that is in charge of maximizing data reuse among frames, while loading newly required data and evicting unused ones from memory.

In this section, we will first define some generic concepts relative to caching and virtual memory systems (Sec. 2.5.1). We will then describe previous works most related to our problem in the domain of out-of-core scientific visualization (Sec. 2.5.3) and texture streaming for real-time applications (Sec. 2.5.2).

2.5.1 Caching : Virtual memory based paging systems

When the size of the dataset that must be accessed is larger than the size of available memory, one way to transparently provide access to this dataset is to rely on some form of virtual memory. The concept of virtual memory has been used for a long time inside all modern operating systems [SGG08]. The purpose of virtual memory is to abstract the physical organization of the memory by giving the illusion of a continuous memory address space, while allowing arbitrary physical allocation, potentially into a complex hardware memory hierarchy (e.g. system memory, disk, network...). Virtual memory provides the ability to transparently implement complex caching schemes, with data silently moved at different levels of the physical memory hierarchy. A caching scheme makes it possible to temporary keep data located inside a large slow memory (called the *backing store*) inside a limited amount of fast memory so that it can be served faster for subsequent accesses.

Virtual memory works by adding a level of indirection between the physical memory and the address space used by an application. Both physical and virtual address spaces are usually subdivided into equally sized sets of memory called *pages*, that serve as an atomic unit for memory management. The term *cache* is usually used to designate the hardware implementation inside a microprocessor of a caching scheme with data cached inside small very fast memories embedded on the chip. The caching scheme implemented inside operating systems for virtual memory (between system memory and disk) is usually called *paging mechanism*.

In our context, we want to provide access from the GPU to a large dataset located in system memory (or defined implicitly as we will see in Section 7.3) and cached inside the video memory. Our data management scheme described in Chapter 7.3 implements a virtual memory paging mechanism with *demand paging*.

As we will see in the next section, virtual memory systems with paging mechanisms have been used for out-of-core scientific visualization since the beginnings of software-based single-threaded rendering on the CPU. In this section, we quickly introduce concepts linked to virtual memory and caching mechanisms and used in the literature.

Caching and paging concepts

Page table A *page table* is used to track the mapping from virtual to physical memory pages (that are generally equally sized). Like the page tables used in microprocessor's caches, page table data structures must efficiently map a block-contiguous, sparsely allocated large virtual address space onto a limited amount of physical memory. When a client to the virtual memory system accesses memory via a virtual address, the system first converts this address to a virtual page index. This virtual page index is converted into a physical page address using the page table, and finally the offset of the requested element into the virtual page is added to the physical page address in order to get its physical address.

Demand-paging A *page-fault* (or *cache miss*) happens whenever a virtual page is accessed while it is not physically present in the cache. In this case, the requested page must be loaded from the higher size, lower bandwidth data storage (the *backing store*). Such behavior is called *demand-paging* and allows data to be loaded only when needed, on-demand from the application. This leads to a *demand-driven* loading scheme.

Replacement policy When a page needs to be loaded into the physical memory (from the higher size, lower bandwidth memory *backing store*), another page already resident has to be selected for replacement (it is *evicted* from the cache). The policy by which a physical page is chosen to be recycled is called the *replacement policy*. Standard page-replacement policies used inside operating systems can be found in [Tan08]. The choice of an optimal replacement policy for a given application depends on the kind of access pattern that will made on the cache. In our case, as for all ray-tracing applications, data access are mostly random access (with a 3D locality). In this case, the *Least Recently Used* (LRU) page replacement policy is acknowledged as being the best general choice. With this policy, the page that has been accessed the least recently is selected for replacement when a new page needs to be loaded inside the cache.

Write back With traditional virtual memory systems, when a page is replaced inside the cache, the data it contains must be saved into the larger *backing store* in order to preserve it for later use. In our rendering application, such saving is not necessary since the data loaded inside the cache are read-only and will not be modified during the rendering operation. Thus, our virtual memory and caching system is read-only (cf. Chapter 7.3).

2.5.2 Texture streaming

In the context of real-time rendering for interactive applications, recent works have focused on using application managed GPU memory regions to store and update texture data depending on view-dependent and visibility informations. Current graphics APIs provide a basic swap mechanism between CPU memory and video memory. Textures are treated as atomic resources: even if only a small part of the texture is used, it is entirely loaded in video memory. Moreover only textures fitting in GPU memory are handled. This swapping mechanism cannot be used in practice because it results in poor rendering performance.

User-centric multi-scale streaming

The *Clipmap* architecture, presented by Tanner et al. [TMJ98], relies on a dynamic multiresolution representation centered around the user to render very large textures for terrains. This representation makes it possible to cache textures of arbitrarily large size in a finite amount of physical memory for rendering at real-time rates. The idea is to keep in video memory only a view-dependent subset of the MIP-map pyramid of full texture (cf. Figure 2.22). This subpart is stored as a stack of nested textures of the same resolution, but covering increasing areas around the viewer. An incremental update is done at each frame on this hierarchical texture representation in order to account for the movement of the user by updating. In this approach, the management of the video memory texture storage is trivial, since it is updated at the beginning of each frame with whatever texels are required for the current frame, based on the movement of the user. However, the system targets datasets that even do not fit inside the main system memory, and texture data must be streamed from disk. In case of fast motions, this can lead to situations where the texture data required for a given frame are not present in main memory. In order to ensure interactivity and not to limit the viewpoint speed, the method relies on lower resolution data present in memory in order to render the frame while the loading from disk is performed asynchronously. While this approach works well for texturing a single terrain with simple topology, it does not fit well with the texturing of multiple complex objects. In addition, it only accounts for LOD based on the distance to the viewer, but did not handle any sort of occlusion. Thus, this kind of viewer-centric approach would not be usable in our context and would lead to a huge amount of non-necessary data to get loaded.

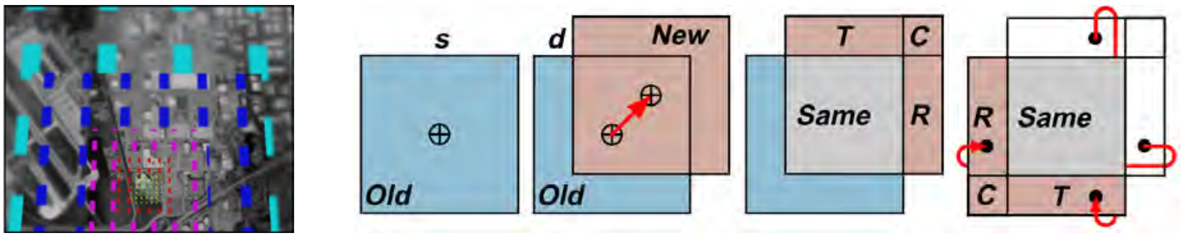


Figure 2.22. *Left:* Clipmap concentric resolution bands centered on the position of the user. *Right:* Illustration of the update process inside a clipmap using toroidal addressing. *Source: [TMJ98]*

Tile-based streaming

One approach to managing large textures is to subdivide the MIP-map pyramid levels into regular grids [GY98, CE98], defining texture tiles that will be loaded or cleared on demand. Usually a priority rule also determines in which order the tiles must be loaded. The size of the tiles is chosen so that each tile can fit into GPU memory. This organization of texture data is depicted in Figure 2.23. The main difficulty consists in detecting which parts of the texture pyramid are needed for rendering from a given viewpoint.

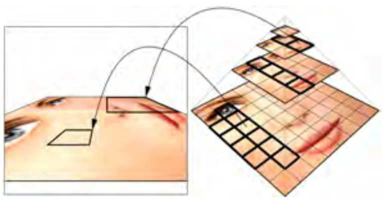


Figure 2.23. Illustration of a tiled MIP-map pyramid representation with correspondence between screen space (left) and texture space (right). *Source: [LDN04]*

Cline et al.[CE98] caches texture data in video memory by splitting the MIP-mapping pyramid of a large texture into smaller textures (corresponding to the tiles). A caching strategy is described to swap the small textures between main memory and GPU memory. Needed parts of the texture are determined by a geometric computation on each polygon. Polygons must be split according to the tiling of texture space for rendering and the discontinuities introduced by the splitting make linear interpolation difficult, since different MIP-map levels are stored in separate textures.

Lefebvre et al. [LDN04] proposed to use a tiled MIP-map pyramid to texture meshes and to manage textures larger than the GPU memory. To do so, they perform a progressive loading of the tiles inside a *Tile Pool* stored in video memory as a large 2D texture. Parts of the texture (the tiles) that are needed for a given view-point are detected using a *Texture Load Map* (TLM). A TLM is a map of the texture space containing visibility information for each tile. Its computation is based on conservative rasterization of the triangle geometry directly in texture space as illustrated in figure 2.24. Only front facing triangles located inside the camera view frustum are rendered within the TLM, and occlusion culling is handled using a *shadow buffer* like algorithm. The problem with this visibility approach is that it requires the geometry to be rendered multiple times and with a costly *fragment shader* in order to fill all MIP-map levels of the TLM. In addition, the TLM needs to be downloaded to the main memory for processing by the Texture Cache, introducing important latencies.

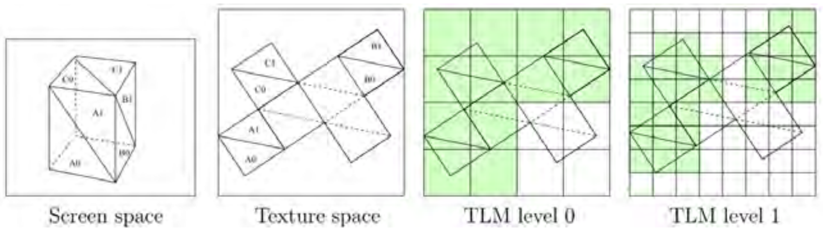


Figure 2.24. The Texture Load Map component of [LDN04] computes the set of visible texture tiles, marked in green, by conservatively rasterizing the geometry in texture space. *Source: [LDN04]*

The management of the Tile Pool is handled by a *Texture Cache* running on the CPU and that implements an LRU replacement policy of the tiles. When a non-loaded visible tile is detected in the TLM, this Texture Cache forwards the tile request to a *Texture Producer* in charge of asynchronously uploading the tile inside the texture memory. This cache does not scale well to very large texture data due to the fixed tile decomposition done at each level of the MIP-map pyramid. For very large textures, the indirection grids used to reference the tiles and the TLM can become too large to be stored in video memory. In addition, many communications and synchronizations have to be done between the main memory and the video memory since the management of the cache is done on the CPU.

Direct tracking of data usage during rendering

In the context of the rendering of complex scenes stored on a distant server, Goss et al. [GY98] proposed to modify the graphics hardware of this time in order to track the usage of texture data directly during the rasterization. This makes it possible to request only visible parts of texture and quickly stream them to the client machine. This approach is based on tile-based decomposition of the textures MIP-map hierarchy (cf. section 2.5.2). A counter is associated with each tile of a texture, and this counter is automatically incremented whenever a fragment using the tile is written inside the framebuffer. It is decremented whenever another fragment using another texture tile passes the z-test and overwrites the pixel value. This counter makes it possible to prioritize the dynamic loading based on the number of pixels that need a given texture data. This approach requires a deep modification of the texturing hardware while it is restricted to a very specific usage that is the rendering of opaque meshes using a unique texture, that is a very unlikely scenario nowadays. It does not easily scale to very large textures due to the fact that it requires scanning the list of tile counters in software in order to trigger loadings (requiring the list to be entirely transfered to the system memory). In addition, the whole texture pyramid is assumed to fit entirely into video memory, which is not possible when dealing with very large textures.

As detailed in Chapter 7, we also propose a data management approach based on direct tracking of data usage during rendering. In contrast to Goss et al. approach, our method runs directly on current generation GPUs without the need for hardware modifications. We implemented it as a data-parallel process using the compute mode of modern GPUs (Sec. 1.3.2). It does not require any CPU intervention, is very flexible and can adapt to many rendering schemes. With our approach, each ray used for rendering (either primary or secondary) directly emits data requests and provides data usage information to a cache mechanism. This provides fine-grained visibility detection.

2.5.3 Out-of-core scientific visualization

The field of out-of-core methods, also referred to as "external memory algorithms" is large and actually dates back as far as 1950s. Out-of-core rendering of very large datasets has been a widely studied topic for years in the domain of scientific visualization. It is still of high importance today, for all cases where the entire model or scene does not fit into memory. A large portion of the paging algorithms described in the literature are application-specific schemes designed to solve a particular caching problem. In the following, we will discuss the approaches that are most recent and most closely related to our work. A comprehensive survey of older out-of-core techniques for software visualization can be found in [SjCC+02].

Early work for software rendering

In the area of software rendering for scientific visualization, early approaches like [CE97] suggested using an application-managed cache mechanism in order to provide a virtual memory for rendering data stored on disk. Such an approach is based on the idea that during the exploration of a large scene, a visualization algorithm needs only to traverse a small subset of the entire scene at a given time. Their system builds upon the operating system LRU (Least Recently Used) replacement policy. They demonstrate how application-controlled demand paging provides significantly better performance than simple reliance on operating system virtual memory. They exploit on-demand loading of pages (application-controlled *demand paging*) when they are used for rendering and show how using fine-grained page size provides better overall performance. In addition, they suggest using a runtime translation of the data loaded from disk in a packed format into an unpacked format in system memory optimized for rendering. We will see in Section 7.4 that our GPU-based data production pipeline can exploit a similar optimization when transferring data between the system memory and the video memory.

Distributed software ray-tracing

Many approaches exploit a PC cluster to render data sets that are too large to fit into the main memory of a single PC. Such an approach is called *distributed rendering*. Wald et al. [WSBW01] have demonstrated the feasibility of interactive ray tracing of large triangle scenes on cluster-based systems. They rely on a central data server to provide data to all the rendering nodes of the cluster. These data are cached locally by the rendering nodes in order to limit costly data transfer through the network. This cache is explicitly managed by the application. It relies on a decomposition of the BSP-tree storing the scene into tiles (called *voxels*). These tiles are managed into a fixed-size geometry cache in local system memory using a least recently used (LRU) strategy. They proposed to "suspend" rays that would cause a page fault in the local cache and load the required data asynchronously over the network while tracing other rays in the meantime. The stalled rays then get "resumed" once the data is available. As we will see in Chapter 7.3, we rely on a similar approach in our GPU paging system.

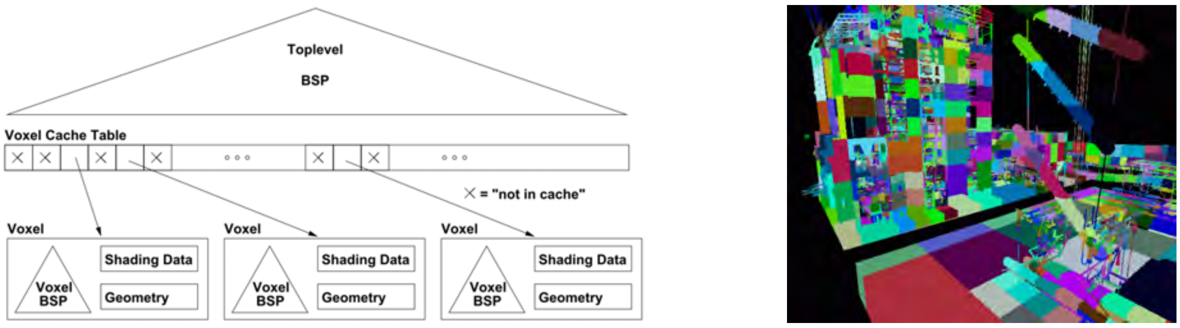


Figure 2.25. Illustration of the cached BSP-tree structure used in [WSBW01]. *Left:* Local caching in cluser nodes of sub-parts of the global BSP-tree grouped into tiles called *voxels*. Voxels are the smallest entity for caching purposes. *Right:* Display of their high-level BSP tree by color coding geometry kept in each voxel.

In the domain of interactive ray tracing of iso-surfaces, DeMarle et al. [DPH+03] improves on Wald et al. by eliminating the bottleneck implied by a central data server. This approach builds upon a cohesive program that is capable of accessing the aggregate cluster memory as a global memory space, with all nodes serving some of the data to the rest of the nodes.

Single machine software ray-tracing

Wald et al. [WDS05] show how to efficiently ray-trace large-scale polygonal models on a single commodity desktop PC. They improve on [WSBW01] by using a combination of automatic OS-based memory management and demand loading in order to detect and avoid page faults due to access to out-of-core memory. They show that a fully manually managed cache (as in [WSBW01]), that manages large sub-parts of the scene (stored into a BSP-tree) of several thousand triangles, leads to much too coarse granularity to scale to very large scenes. In addition, reducing the managed page granularity would have carried too much overhead to their mono-threaded cache management scheme. Instead, they simply relied on the memory management of the operating system that provides transparent direct addressing of data stored on disk, and demand loading memory on a per-page basis. The advantages are a fine-grained paging using small pages as well as an automatic handling of race-conditions in the case of multi-threaded access. Such an approach requires to pre-compute the whole data structure off-line and to store it on disk in the exact same binary form as the one used for rendering. The problem when relying on the OS paging mechanism is that data are automatically paged-in on demand upon accessing it, and the resulting page fault stalls the rendering thread until the data are available.

In order to always maintain interactivity, they proposed to shortcut the automatic on-demand paging of the system to allow loading to be performed in a separate loading thread without stalling the rendering threads. Page fault are detected by the rendering threads before accessing data on a per-tile granularity (with tiles grouping multiple pages) using a dedicated structure. When a page fault is detected, the rendering thread triggers a fetching of a tile using a request queue. It then simply stops its execution and relies on a simplified *proxy* representation to provide an approximated color to the computed pixel. Proxies are lightfield-like pre-rendered approximations of the model, at several LOD levels. They are kept in memory in order to always stay available. For complex models, the loading from disk usually does not fit inside a single frame and has to be spread among multiple frames. Thus, this approach ensures interactive rendering (3-7 fps at video resolution for the Boeing 777 dataset), but it takes several seconds to get all data loaded to get a complete frame as illustrated in Figure 2.26.

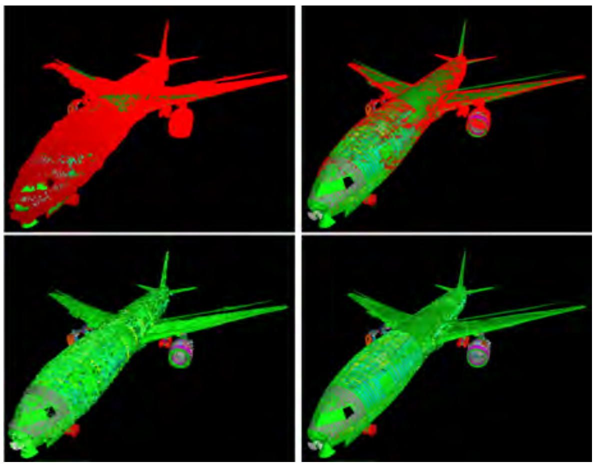


Figure 2.26. Illustration of the quality approximation of [WDS05] during startup time on the Boeing 777 dataset. Top row: No proxy information, canceled rays displayed in red. Bottom row: Using geometry proxies. Left: immediately after startup, lot of pixels used proxy. Right: after loading for a few seconds. Even then only a fraction of the model has been loaded. *Source: [WDS05]*

GPU-based rasterization

The problem with the approaches presented previously is that the lack of multiresolution data forces the algorithm to access large parts of the dataset to produce a single frame even for distant views. Moreover, these methods rely on software ray-tracing implemented on a cluster, and do not exploit GPU acceleration.

In the context of massive model visualization using standard graphics hardware, Gobbetti et al. [GM05] proposed with their Far Voxels approach to use a level-of-detail (LOD) hierarchy based on an axis-aligned BSP-tree [3] of the triangle data. They exploit an idea similar to Wald et al. [WDS05] proxy geometries as a way to filter distant groups of triangles, with the surface representation kept for close-up renderings. The leaves of the BSP-tree partition the original triangle mesh into chunks of a fixed maximum number of triangles (around 5000), while inner nodes contain a view-dependent voxel representation precomputed off-line. Voxels are generated from a discretization of the bounding box of the node with a regular grid. Due to the nature of the BSP-tree, all nodes of the tree have different size and so must be discretized with a different number of voxels. Non-empty voxels are stored as a list of points linked by each inner-node of the tree. Rendering is done using a breadth-first front-to-back traversal of the structure on the CPU, sending primitives to the GPU for rasterization. Triangles located in the leaves of the tree are rasterized and inner node's voxel data are rendered using a splatting approach (cf. section 2.3.2). Frustum culling as well as LOD computation (to find the correct depth based on the distance to the viewer) are computed on the CPU during the traversal of the structure. Visibility culling of occluded nodes is performed using hardware occlusion queries (cf. section 2.4.3).

This approach stores the entire pre-computed data structure on disk and fully relies on the automatic paging mechanism of the operating system to access it during rendering. An application managed RAM buffer is used to keep the most recently rendered nodes inside system memory, but no detail is given in the paper on the way it is managed. The storage of the rendering primitives (triangles and splatted points) in the GPU's video memory is done using an OpenGL's Vertex Buffer Object (VBO). This buffer keeps the last rendered primitives available in video memory using what seems to be a simple FIFO replacement policy, but no special care is taken for efficient data reuse and transfers.

[3] Binary Space Partitioning

This approach suffers from many popping artifacts due to the lack of spatial interpolation of voxel data, and from the lack of continuous transition between LOD levels and between voxel and triangle representations. In addition, the pre-processing of such data is very time consuming.

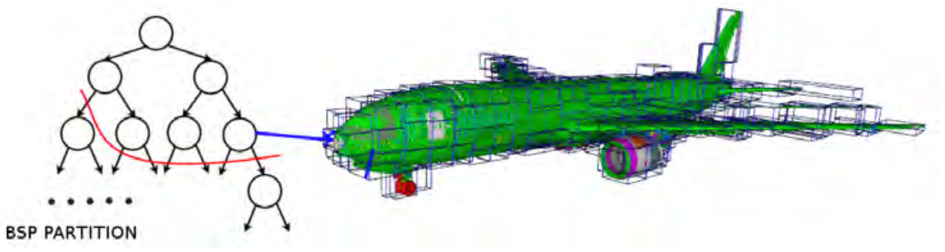


Figure 2.27. . *Source: [GM05]*

GPU-based volume ray-casting

Our approach was developed in parallel with the work of Gobbetti et al. [GMAG08] that focuses on scientific visualization applications (thus, do not follow the same goal). Similarly to us, Gobbetti et al. proposed a multiresolution voxel data structure based on an octree combined with regular bricks of voxels, as well as a full GPU-based traversal of this structure for rendering (cf. chapter 6). While we share similarities with this work, our approach provides better rendering quality, more compact storage, higher performance and more precise visibility detection as well as a more efficient data management. First, Gobbetti et al. do not provide a correct filtering of the multiresolution data since they do not allow quadrilinear filtering (no smooth transition). Their octree data structure implementation is less compact than ours (16 times) and thus provides less efficient storage and cache usage during traversal. In addition, their visibility detection relies on occlusion queries which corresponds to the first approach we tried and gave up due to its lack of efficiency in our context (Sec. 2.4.3).

Finally, their data management and caching scheme heavily relies on a serial processing on the CPU with a cloned data structure in system memory. Our approach relies on a fast data-parallel scheme implemented entirely on the GPU, removing all synchronization with the CPU and that does not require a cloned data structure in system memory. As in all previous approaches, their rendering scheme requires the hierarchical structure to indicate the correct LOD for a given point-of-view, forcing an update at each frame. In contrast, our scheme computes directly the required LOD level per-ray during rendering, allowing a lazy update of the structure with only the newly required parts getting updated at each frame.

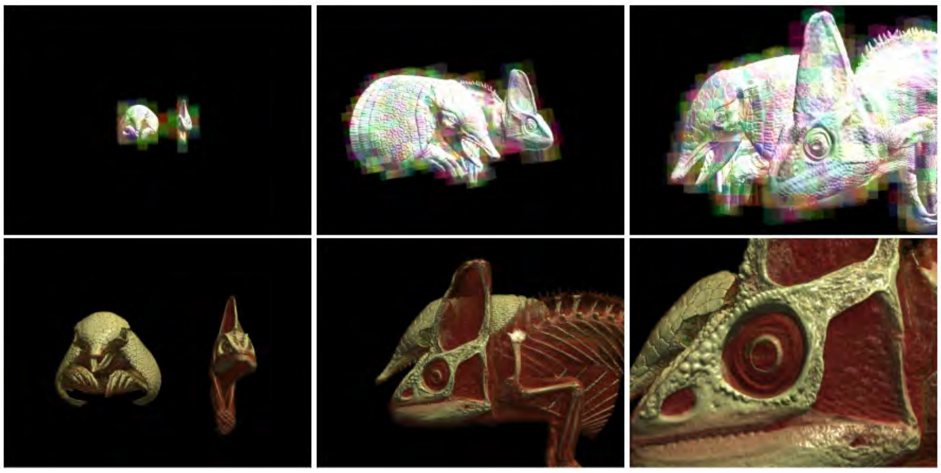


Figure 2.28. Illustration of medical data rendered using the out-of-core volume ray-casting system presented in [GMAG08].

Part I

Contributions: core model

Chapter 3

The GigaVoxels rendering pipeline

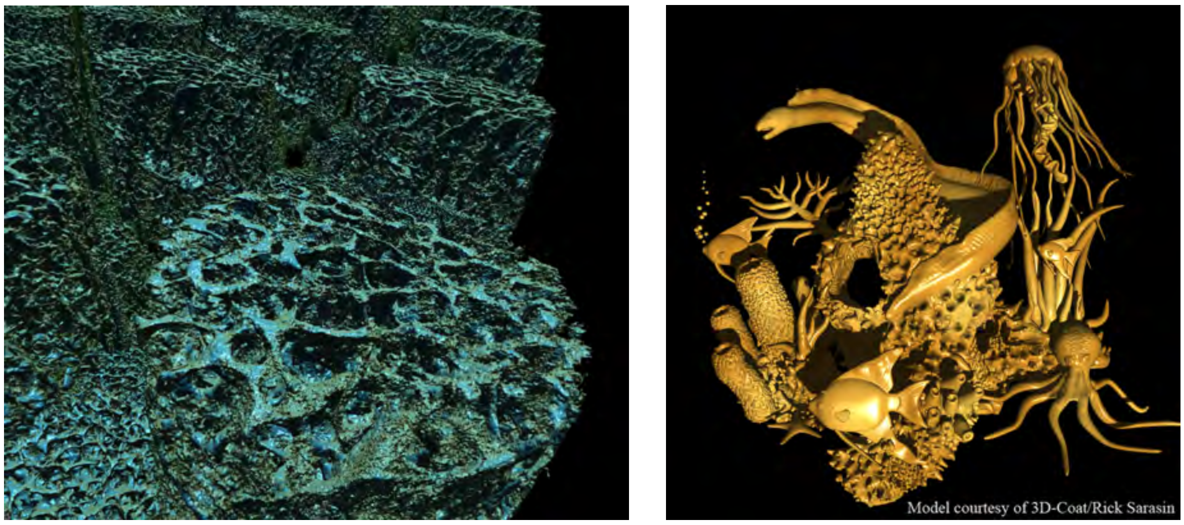


Figure 3.1. Examples of scenes composed respectively of 8192^3 and 2048^3 voxels rendered interactively using our engine.

In this chapter, we present an overview of our voxel-based GPU rendering pipeline that makes the display of large and detailed volumetric objects and scenes very efficient. Voxels increase the amount of displayable detail significantly beyond the limits of what can be achieved with polygons and support transparency effects natively (which is a major issue in real-time rendering).

Our pipeline deals with the three main problems that appear when rendering voxel based representation on the GPU: the rendering problem, the storage problem and loading inside video memory. It proposes a new compact data structure, as well as an efficient rendering algorithm and an out-of-core streaming and data generation scheme, both designed as efficient data-parallel tasks entirely running on the GPU. It is capable of rendering objects at a level of detail 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. As a result, our model obtains interactive to real-time performance and demonstrates the use of extreme amounts of voxel data for interactive applications, which is applicable in many different contexts. With such an approach, we demonstrate in Chapter 6 that voxels can achieve higher performance than triangle-based representations for very complex scenes.

3.1 Global scheme

Let's start with a naive consideration. If the volume is small, GPUs allow an efficient rendering by simply stepping through the dataset stored in a 3D texture as it is done by traditional volume ray-casting approaches detailed in Section 2.3.2. In this way, one can benefit from several hardware related advantages, like direct 3D addressing, trilinear interpolation, and a 3D coherent texture cache mechanism (Sec. 1.3.4). For larger volumes, there are two problems: first, the algorithm would be slow due to many steps that need to be taken in large datasets and, second, the whole dataset will not fit into the GPU memory.

Previous out-of-core approaches for rendering massive amounts of data (cf Section 2.5.3) mainly focused on caching data located on disk into the main system memory. At the time they were written, the limited amount of system memory and the limited bandwidth between disk and system memory were the main bottlenecks in interactive application. Little care was taken about the caching of data inside the video memory. Nowadays, massive amounts of data can be easily stored directly in system memory, and the bottleneck is shifted to the limited amount of video memory and the limited bandwidth with the large system memory (cf. Section 1.3).

By organizing the data in a spatial subdivision structure (Sec. 2.4.2) in video memory, empty or constant parts of the volume can be left unsubdivided, which can already represent a significant compaction in our typical scenes (cf. Section 1.4.2). This also allows us to skip empty spaces during ray-casting, speeding-up greatly the process. In addition, such structures allow a multiresolution rendering scheme to be used (Sec. 2.4.4). Thus, rendering resolution can be adapted to the distance to the viewer. Distant parts can be replaced by lower mipmap levels, leading to a lower resolution (a different level-of-detail, *LOD*), thus lower GPU memory requirements. Since volume resolution, and hence the size of stored voxels, is adapted to the distance to the observer, one can ensure that all voxels stored for a given point of view are sized to project approximately on the area of one pixel on the screen.

Our goal is to provide a scheme that is able to scale to very large and detailed scenes. In this context, the whole scene and all its scales can not be permanently kept inside the limited amount of available video memory, even compacted inside a multiresolution spatial subdivision structure. One insight is that, for a given point of view, the entire volume does not need to be in memory for rendering. Only visible voxels need to be present, out-of-frustum as well as occluded data do not need to be loaded. The subset of data required for a given point of view has to be determined using a visibility detection algorithm (cf. section 2.4.3). Ideally, and thanks to the multiresolution scheme, only a few voxels per pixel would need to be present in the video memory.

During the exploration of a scene, only newly visible voxel data need to be loaded inside the video memory. Therefore, we developed an efficient streaming scheme able to dynamically load voxel data inside the video memory. This scheme is entirely driven by demands emitted directly by the rendering algorithm, and ensure the loading of the minimum amount of data needed for a given frame. Also, visible data already present in memory must be kept loaded, while invisible ones can be replaced. This loading requires an out-of-core loading scheme (cf. Section 2.5.3) as well as a caching scheme (cf. Section 2.5.1) maximizing the reuse of data from frame-to-frame.

3.2 Overview

We will now detail the different parts of our GigaVoxels GPU rendering pipeline depicted in Figure 3.2. Our model builds upon the hypothesis that the total amount of voxel data required for a given point of view is less than the total amount of video memory. In such a case, only a few new data need to be loaded in video memory at each frame, and data already loaded can be reused during the exploration of a scene. If it is not the case, a caching scheme becomes ineffective and most of the advantages of our system for real-time applications are lost. However, as we have seen in Section 1.4.2, most of the scenes we are interested in are made of layers of voxel data that, even if they are not opaque, quickly become opaque when accumulated. Thus, this ensures that for each pixel of the screen, only a few semi-transparent voxels needs to be accumulated before a pixel becomes totally opaque, blocking the view to subsequent data.

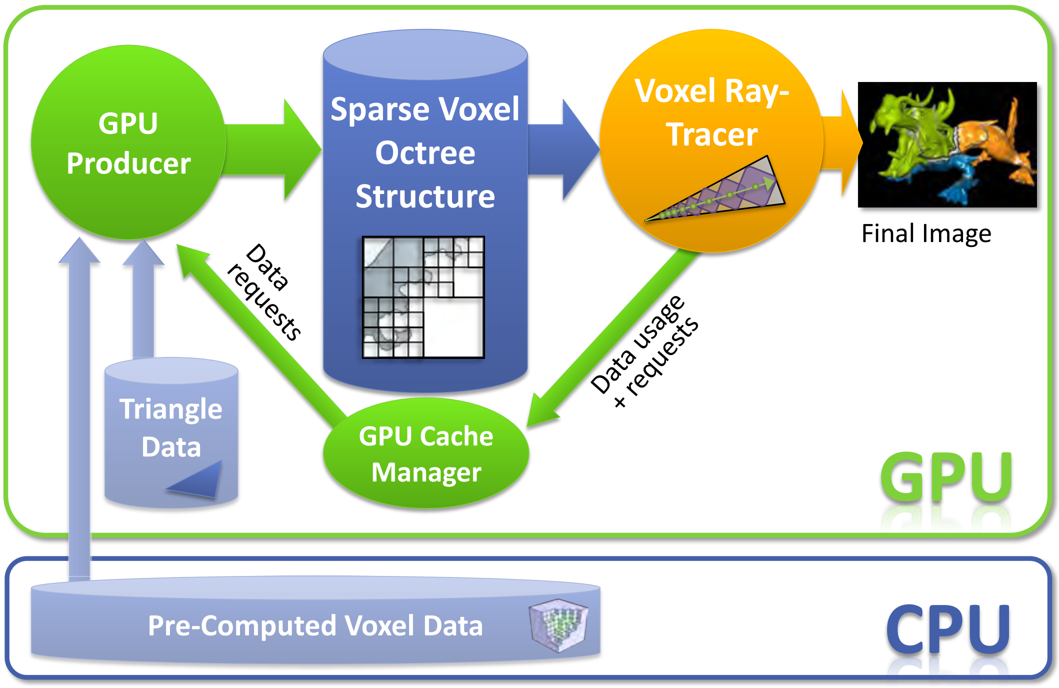


Figure 3.2. Global view of the GigaVoxels rendering engine. Blue parts are stored data, green elements are process related to the data management, the orange element is dedicated to the rendering, and pale blue elements are examples of optional data sources.

Data structure

The data structure has two goals: first to provide a pre-filtered geometry representation allowing high-quality alias-free rendering (cf. chapter 4), and second to provide a compact storage for a very large amount of voxel data, with fast access for rendering.

Our pipeline is centered around a sparse multiresolution structure storing the voxel data. As detailed in Section 2.4, previous data structures of this kind (designed to be traversed on the GPU) stored voxel data directly in the tree, thus preventing the implementation of fast and accurate filtering. Instead, we designed an octree-based GPU structure, which is convenient to represent and to traverse on the GPU [LHN05b, LHN05a] and is well adapted to store regular data like voxels. It is combined with small regular grids allowing fast hardware filtering, and easy data manipulation. We will detail this structure as well as its efficient GPU implementation in Chapter 5.

Rendering

Our octree-based structure is used by a voxel rendering algorithm based on ray-tracing. This algorithm directly traverses the data structure from front to back for each pixel of the screen, and ensures that only the few visible voxels will be accessed for each pixel. It is detailed in chapter 6.

One of the main particularities of our rendering approach compared to previous works on multiresolution rendering (Sec. 2.4.4) is that it does not need the data structure to indicate the required level of detail (LOD) for a given point of view. Instead, LOD is dynamically evaluated by each ray, allowing the data structure to contain more data kept cached in video memory for future use, as well as arbitrary ray directions, secondary rays (for shadows or reflection/refractions) and free spatial instancing of the same data structure.

Caching and streaming

The data structure is stored in a video memory region managed as a cache by a *Cache Manager*, in order to ensure data reuse during the exploration of a scene. This cache is based on a full LRU (Least Recently Used) replacement policy (cf. Section 2.5.1) entirely implemented as a data parallel process on the GPU. This cache manager implements a paging mechanism (cf. Section 2.5.1) that allows the efficient loading of the missing data required for rendering, while recycling the oldest ones.

In contrast to previous approaches that relied on complex visibility detection algorithms (cf. section 2.5.3) to determine newly required data and data used for rendering, our solution inherently computes the visible parts of the scene during rendering. Our approach makes use of the ray-tracing to directly emit data requests and usage information per ray. Since our ray-tracing approach ensures an ordered traversal of the data, with only visible voxels actually touched, this provides us with fine-grained on-demand loading.

We call this scheme *ray-guided streaming*. It allows us to transparently support many features that were previously difficult or even impossible to support: loading data for arbitrary ray directions, curved rays, arbitrary LOD (for instance allowing depth-of-field effects as presented in Section 8.3), secondary rays (as exploited in our soft shadow application presented in Section 8.3), as well as free spatial instancing of the same data structure (cf. Section 8.2). Such possibilities were not easily allowed by previous approaches. Our whole cache management scheme is detailed in Chapter 7.

Voxel generation and loading in the GPU cache

One of the problems in using voxel representation as a natural rendering primitive is to generate these voxel data and to efficiently transfer them inside the data structure in video memory. With previous approaches, data were pre-computed and stored on disk to be loaded dynamically in system memory and transfered in video memory from the CPU at runtime (cf. Section 2.5.3). In Section 7.5.3, we show that such transfer initiated from the CPU is very slow when many elements need to be transfered, due to high latencies between copy operations. It also requires a lot of communication and synchronization between the CPU and the GPU.

Instead, with our approach, data stored in system memory are loaded in parallel directly from the GPU, and written in video memory also in parallel from the GPU. We also allow data to be dynamically generated on the GPU. Thus, this allows voxel data to be either precomputed in a pre-process and kept available in system memory or generated on-the-fly during rendering (either fully procedurally or from another representation, for instance a triangle mesh). This also allows compressed representations to be stored in system memory, loaded by the GPU and transformed on the fly into the voxel representation needed for rendering, before being written in our data structure.

This task is managed by the last component of our pipeline: the *GPU producer*. It is a fully GPU process in charge of loading data inside the data structure, on-demand from the cache manager (and thus from the rendering). It is a user-defined component that can implement any voxel production or loading scheme. We demonstrate direct loading of pre-computed voxel data from the system memory, voxelization from a triangle mesh, fully procedural generation as well as combinations of these approaches. GPU producers are detailed in Chapter 7.

3.3 Technological choices

In order to develop this pipeline and implement it on the GPU, several technological choices had to be made. One of the difficulties of this thesis has been the fast evolution of the GPU technology, while many choices had do be done at the beginning of the thesis, 3-4 years ago. But these choices mainly affected the implementation of our pipeline, while the fundamental design and the major technical choices we made stayed valid, thanks to their strong fundamental justification.

Due to these technological changes, especially in the way of programming the GPU with the introduction and evolution of CUDA and the compute mode presented section 1.3.2, 3 very different prototype implementations of our pipeline have been developed during this thesis. Our final model builds upon CUDA to implement the cache manager and the producers, while the rendering algorithm is implemented both in CUDA and in OpenGL GLSL [Khr].

3.3.1 Preliminary GPU performance characterization

In order to make the relevant technical choices, one of the first tasks for this thesis has been to characterize various performance behaviors of the GPU. Indeed, while graphics and compute APIs provide a view of the features provided by graphics hardware, many details are still hidden to the programmer and particularly performance bottlenecks and fast paths are not always easy to determine and are usually dependent on the GPU vendor and generation.

Thus, it is important to determine these hardware behaviors in order to make the right implementation choices. To do so, we relied on unitary tests of specific hardware features. Especially, we tested the scheduling of fragment shader threads as presented in Appendix A.2 in order to speed up our ray-casting algorithm. We also tested the behavior of texture caches depending on various texture types as presented in Appendix A.1, in order to make a choice of texture storage for our data structure (Chap. 5). Some publications also provide precious insight on the performance of various GPU features and instructions of the computing ISA *(Instruction Set Architecture)*, like [WPSAM10].

Chapter 4

Volumetric geometry representation and pre-integrated cone tracing

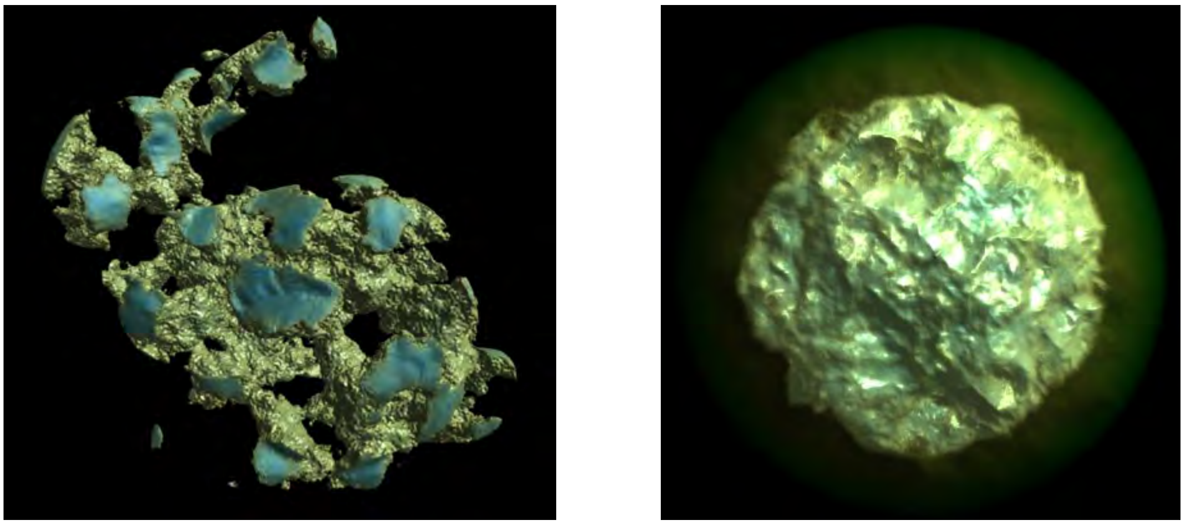


Figure 4.1. *Left:* Triangle mesh voxelized at 2048^3 resolution, amplified with procedural noise and rendered at 70FPS on a GTX280 with GigaVoxels. *Right:* Fully procedural sphere with a semi-transparent shell perturbed with a procedural noise and rendered at 100FPS.

In this chapter, we first describe our volumetric geometry and material representation, as well as the pre-integrated rendering model based on approximate cone-tracing that we propose in order to ensure an accurate and alias-free rendering of pre-filtered multi-scale scenes.

As we have seen in Section 1.1.2, the idea of voxel based pre-filtering is to represent object geometry using a density model stored inside a voxel MIP-map pyramid. As we will see in Section 4, accurate geometry pre-filtering does not only imply simple density averaging, but also supposes a pre-integration of the visibility in order to ensure a correct occlusion between filtered data.

We will demonstrate how we build our pre-filtered geometry model by starting from the integration over a screen pixel footprint of the classical volume rendering integration formula along a single ray, and we will detail how this double integration can be split and pre-computed in order to be stored inside a MIP-map representation.

We will also describe how we transform a surface-based object representation into a volumetric representation more easily filterable.

4.1 Overview

While classical B-rep surface definitions can not be linearly combined, we replace them by a statistical distribution of densities stored as voxels into a regular subdivision of space, that can be linearly pre-filtered inside a MIP-map pyramid, as explained in Section 1.1.2. In Section 4.6, we will see how we transform a surface-based object representation into a volumetric representation more easily filterable.

Most of the previous work on volume rendering (cf. Section 2.3) has been done in the context of scientific visualization. These approaches usually render semi-transparent the participating medium. In such a context, only a single scalar value is usually stored per voxel, and is transformed at runtime into optical coefficients (colored light energy and opacity, using a *transfer function*) for volume integration (cf. Section 2.3.1).

In our context, we want to store filtered material and geometry information at multiple scales, that will allow a quick rendering of very-complex geometry, and to dynamically compute the light interaction at all scales during rendering. Typically, we want at least a filtered density (or an opacity based on the density), a material color and a normal information per voxel. Storing and interpolating such information poses particular problems that have not been handled previously.

In addition, the problem of pre-filtering solid geometry information at multiple resolutions inside a volumetric representation has never been really posed in previous work. Multi-resolution approaches (Sec. 2.4.4) concentrate on reducing the memory occupancy of volume representation, but do not focus on providing a correct pre-integration of the characteristics of the data at smaller scales. Usually, a simple averaging of high resolution values is used. Such filtering is usually enough to render scientific data, but does not account for self occlusions inside the filtered volumes.

As we will see in Section 4.2, accurate geometry pre-filtering also supposes a pre-integration of the visibility in order to ensure a correct occlusion between filtered elements. Thus, exact pre-integrated data that must be stored per-voxel depends on the model used for rendering. In order to get a complete pre-integrated and pre-filtered voxel-based geometry model allowing a fast antialiased rendering, three key elements will be defined in this chapter:

- A voxel-based volumetric geometry representation.
- A multiresoltion pre-integration of this geometry representation.
- A model for cone-tracing based on the multiresolution representation in order to provide a precise, alias-free rendering.

4.2 Pre-integrated cone tracing

Since first discussed by Crow [Cro77] in the middle of 1970s, aliasing has been a major problem in rendering. One screen pixel is associated with more than just a line in space. It actually corresponds to a cone because a pixel covers an area and not a single point on the screen (as illustrated in Figure 4.2). This is typically the source of aliasing that arises when a single ray is used per-pixel to sample the scene.

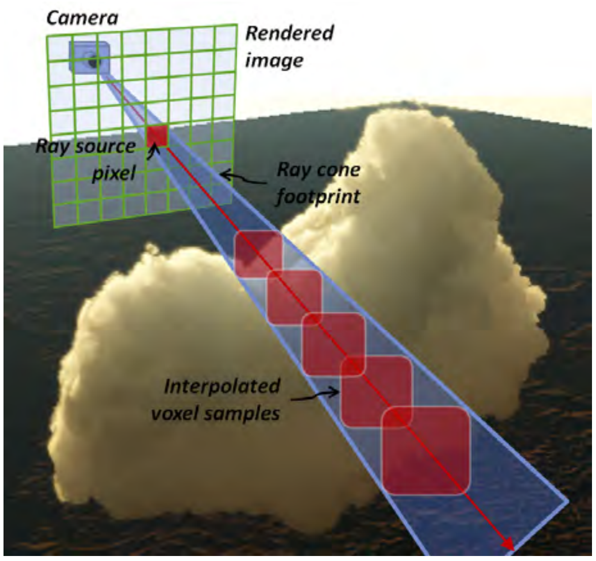


Figure 4.2. Illustration of the cone-shaped beam of light generated by the perspective projection and going toward a given pixel. We approximate it using a single ray launched from the pixel and sampling a pre-integrated representation of the scene geometry.

In order to integrate the incident radiance coming from a cone footprint, one needs to integrate the incoming radiance $I(D,\omega)$ coming from all ω directions in the cone:

$$I = \int_{\Omega} I(D,\omega)d\omega \tag{4.1}$$

Classical supersampling approaches (cf. Section 2.1.1) discretize this integral with multiple rays to deal with aliasing, which lead to a very costly evaluation. Other approaches like cone or beam tracing (Sec. 2.1.1) provide faster antialiasing, but rely on an analytical definition of the geometry and can not be applied in a general case. Instead, we propose a model that is based on a single ray per pixel, and deals with the aliasing by relying on a pre-filtered geometry representation stored inside a MIP-map pyramid.

This approximated cone tracing is done under the assumption that we can approximate the integration over a cone of visibilities, with the integration along a single ray of pre-integrated spatial visibilities. We base our model upon the classical volume rendering integration model explained in Section 2.3.1. It describes the integration of both the visibility and the reflected energy along a ray inside a participating medium.

In the next section, we will describe step by step how we build this pre-integrated rendering model, together with assumptions we make on the rendered data in order to allow it.

4.2.1 Volume pre-integration theory

Volume density model

The physical model of light transport into a participating medium that is used by the classical volume rendering integration scheme has been described in Section 2.3.1. In this model, the medium is described using two main optical coefficients: the total absorption (also called *extinction*) $\chi = \kappa + \sigma$, the sum of the true absorption coefficient κ and the out-scattering coefficient σ; and the total emission $\eta = q + j$, the sum of the true emission coefficient q and the in-scattering coefficient j.

In our model, we consider the total absorption coefficient $\chi(s, r)$ on a point s along a ray r. The participating medium represents a filtered geometry that rarely emits light on its own, but instead more often scatters (in the view direction) some energy coming from an external light source. Thus, we do not consider the true emission coefficient q. Instead, we consider the in-scattering coefficient $j(s, r)$ that corresponds to the energy reflected (scattered) by our filtered surface and material model on a point s along a ray r in the direction of the eye.

As we will see in Section 4.5, our geometry pre-filtering model will not directly store the scattering energy j, but instead the material properties allowing us to compute it dynamically at render time (based on any lighting model, local or global). However, we will first build our model based on χ and j themselves, and we will explain later (in Section 4.5) how the lighting computation can be factored-out of the pre-integration.

Toward a volume pre-integration model

In this explanation, we first assume a parallel projection leading to orthographic beams generated by pixel footprints, we will see later how to extend this model to perspective projection with cone-shaped beams. We build upon the notations introduced in Section 2.3.1.

Our goal is to build a pre-integration model for the computation of the average energy $I(D, P)$ accumulated over the footprint of a pixel P on the screen. This can be modelized as the integration of the volume rendering integral (Sec. 2.3.1, equation 2.11) along a ray r over the footprint of the pixel P as described by equation 4.2 and illustrated in Figure 4.3. In order to keep equations simples, we first discuss this model within the parametrization of orthographically aligned rays emitted by a given pixel. We will see later how to transform our results expressed in this ray-based parametrization in order to get a world-space parametrization that allows a static storage of pre-integrated values inside our voxel MIP-map pyramid.

$$I(D, P) = \int_{r \in P} \int_{s_0}^{D} j(s, r) e^{-\int_s^D \chi(t,r)\, dt} ds\, dr \tag{4.2}$$

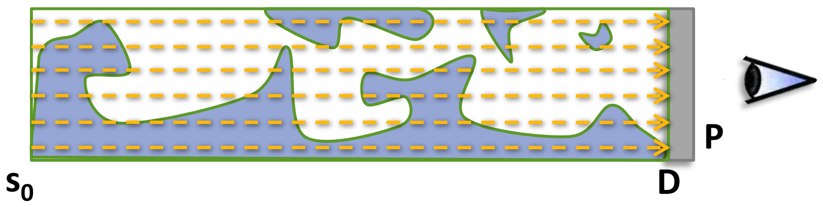


Figure 4.3. Integration over a pixel footprint of the volume rendering integration along orthographic rays.

What we propose is to replace the integration over a pixel of the ray-based volume rendering integration function, by a pre-integration inside a set of cubical volumes of this function in object space as illustrated in Figure 4.4. Our goal it to store a discrete set of these pre-integrated volumes per-voxel inside a 3D MIP-map pyramid, in order to use them for fast rendering.

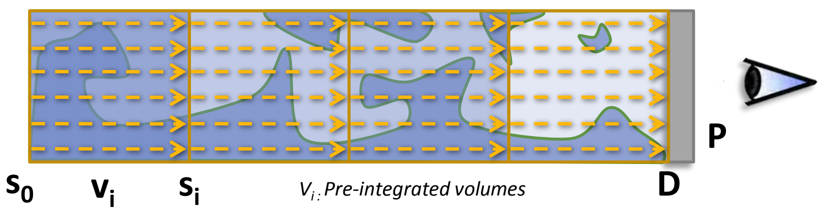


Figure 4.4. Accumulation for a pixel of the pre-computation inside sub-volumes of the volume integration.

Step-by-step pre-integration

We will describe step by step how we reach this goal and what are the approximations and hypothesis we make on the data in order to allow it.

First, we need to split the volume integration along a single ray into a discrete number of sub-integrations that can be precomputed. In order to simplify the equations, we denote $\tau(s1, s2, r)$ the *optical depth* that corresponds to a local path integration of the total extinction between $s1$ and $s2$ along a ray r:

$$\tau(s1, s2, r) = \int_{s1}^{s2} \chi(t, r)dt \tag{4.3}$$

And we denote $Q(s1, s2, r)$ the local path integration of the in-scattering energy:

$$Q(s1, s2, r) = \int_{s1}^{s2} j(s, r)e^{-\tau(s,s2,r)}ds \tag{4.4}$$

Based on these notations, we split the integration along rays on the interval $[s_0, D]$ into a discrete number of local sub-paths $[s_i, s_{i+1}]$, for both the energy integration part and the sub-integration of the attenuation:

$$I(D, P) = \int_{r\in P} \sum_{i=0}^{n} \left(Q(s_i, s_{i+1}, r) \cdot e^{-\sum_{j=i+1}^{n} \tau(s_j, s_{j+1}, r)} \right) dr \tag{4.5}$$

These discrete summations correspond to the marching process done along rays in classical ray-casting rendering approaches presented in Section 2.3.2.

Our final goal is to bring the integration over a pixel P inside the two discrete summations, in order to apply it directly to the sub-paths integration of both Q and τ. Once applied to these sub-paths, that will allows us to pre-compute them and to store them inside our hierarchical voxel representation.

For the first step, we rely on the Fubini's theorem. Based on the continuity of all functions (all parameters are defined continuously in space), we can swap the discrete sum and the integral over the whole expression:

$$I(D, P) = \sum_{i=0}^{n} \left(\int_{r\in P} Q(s_i, s_{i+1}, r) \cdot e^{-\sum_{j=i+1}^{n} \tau(s_j, s_{j+1}, r)} dr \right) \tag{4.6}$$

We will now use a general hypothesis on our data the impact of which will be detailed in greater depth in Section 4.4. This hypothesis is that, on a given ray r, a sub-path integration of energy $Q(s_i, s_{i+1}, r)$ is decorrelated from $e^{-\sum_{j=i+1}^{n}\tau(s_j,s_{j+1},r)}$, the integration of the total absorption happening on the whole path between s_{i+1} and the position of the eye. This means that these two values do not have a statistical dependence. Within this hypothesis, thanks to the definition of the statistical correlation $correlation(a(), b()) = \int a()b() - \int a() \int b()$, we can replace the integral of the product of the two decorrelated terms by the product of their integral, as expressed in equation 4.7

This hypothesis is linked to a more global hypothesis we make on the distribution of densities in our scene (cf. Section 4.4). The in-scattered energy j and the total absorption χ are linked together on a given point in space through our volume density model (Sec. 4.2.1), both being multiplied by the density coefficient ρ. Thus, the correlation between values along a given ray could come either from the distribution of density, or from the distribution of the initial absorption or in-scattered energy values. We will detail in section 4.4 in which cases this correlation can appear. However in a general case, we consider the decorrelation hypothesis as acceptable.

$$I(D,P) \approx \sum_{i=0}^{n}\left(\left(\int_{r\in P} Q(s_i, s_{i+1}, r)dr\right)\cdot\left(\int_{r\in P} e^{-\sum_{j=i+1}^{n}\tau(s_j,s_{j+1},r)}dr\right)\right) \tag{4.7}$$

Thanks to this hypothesis, we already obtain a formulation of the in-scattered energy in the form we were looking for, pre-integrated over a pixel P between the parameters $s1$ and $s2$: $\overline{Q}(s1, s2, P) = \int_{r\in P} Q(s1, s2, r)dr$.

We now need to get a similar expression for the pre-integration of the optical depth. To do so, we need to bring the integration over P inside the inner sum of optical depths computed in the exponential. Instead of trying to pull the integration over P up in the exponential, we pull down the sum from the exponential into a product of exponentials:

$$I(D,P) \approx \sum_{i=0}^{n}\left(\left(\int_{r\in P} Q(s_i, s_{i+1}, r)dr\right)\cdot\left(\int_{r\in P} \prod_{j=i+1}^{n} e^{-\tau(s_j,s_{j+1},r)}dr\right)\right) \tag{4.8}$$

In fact, this formulation as a product of exponentials corresponds exactly to the way transparency $T(s1, s2, r) = e^{-\tau(s1,s2,r)}$ is classically computed with a discrete integration on a given path along a ray (cf. section 2.3.1). Thus, we chose to pre-integrate transparency instead of the optical depth itself. With this formulation, the only operation remaining in order to get a formulation of the pre-integrated transparency is to swap the integral over P and the product operator.

Once again, we rely on a correlation hypothesis. This time, the correlation hypothesis is a bit more restrictive, since we need all successive optical depths $T_j = e^{-\tau(s_j,s_{j+1},r)}$ to be decorrelated. This means that along a given ray, successive optical depths (pre-computed along sub-paths) do not have a statistical dependence. This hypothesis implies that initial densities (coming from the geometrical model) should be distributed as randomly as possible inside a beam, on large distances[1]. We will see in Section 4.4 that this is usually relatively true in the scenes we are interested in, and that violating this hypothesis only has a limited impact on the rendering quality.

By definition of the correlation, $correlation(a(), b()) = 0 \Rightarrow \int a()b() = \int a() \int b()$. Thus, thanks to our non-correlation hypothesis, one can swap the integral over P and the product operator :

[1] However for short distances, correlation would be handled correctly since the integration would be pre-computed correctly inside the pre-integrated sub-paths

$$I(D,P) \approx \sum_{i=0}^{n} \left(\left(\int_{r\in P} Q(s_i, s_{i+1}, r) dr \right) \cdot \prod_{j=i+1}^{n} \left(\int_{r\in P} e^{-\tau(s_j, s_{j+1}, r)} dr \right) \right) \tag{4.9}$$

We now get the two functions we want to pre-compute and store in our pre-integrated voxel representation, the average in-scattered energy $\overline{Q}$ and average transparency $\overline{T}$:

$$\overline{Q}(s1, s2, P) = \int_{r\in P} Q(s1, s2, r) dr \tag{4.10}$$

$$\overline{T}(s1, s2, P) = \int_{r\in P} e^{-\tau(s1, s2, r)} dr \tag{4.11}$$

That leads us to the following render-time integration scheme that will be implemented by the rendering algorithm presented in Chapter 6:

$$I(D,P) \approx \sum_{i=0}^{n} \left(\left(\overline{Q}(s_i, s_{i+1}, P) \right) \cdot \prod_{j=i+1}^{n} \left(\overline{T}(s_j, s_{j+1}, P) \right) \right) \tag{4.12}$$

4.2.2 Discrete composition scheme

Thanks to the model we just presented, and based on equation 4.12, it is possible to evaluate the radiative light $I(D,P)$ coming from a beam, towards a pixel P, by sampling along a single ray.

To do so, we define a composition scheme that can be used to incrementally compute $I(D,P)$. For a pixel P, we define $\overline{Q}_i = \overline{Q}(s_i, s_{i+1}, P)$ and $\overline{T}_i = \overline{T}(s_i, s_{i+1}, P)$, as well as accumulated energy $\hat{I}_i$ at position i and transparency $\hat{T}_i$. This leads to the following front to back compositing scheme, with the final $I(D,P) = \hat{I}_0$:

$$\hat{I}_i = \hat{I}_{i+1} + \hat{T}_{i+1}\overline{Q}_i \tag{4.13}$$

$$\hat{T}_i = \hat{T}_{i+1}\overline{T}_i \tag{4.14}$$

with the initialization:

$$\hat{I}_n = \overline{Q}_n \tag{4.15}$$

$$\hat{T}_n = \overline{T}_n \tag{4.16}$$

4.2.3 World-space definition

Based on the model we just described that is expressed in ray-space, we want to pre-compute the two functions $\overline{Q}$ and $\overline{T}$ in world-space, so that they can be stored per-voxel in our MIP-map representation. Thus, we redefine these two functions depending on a position $\mathbf{p}$, a direction $\mathbf{d}$, a surface area of integration s and a length of integration l:

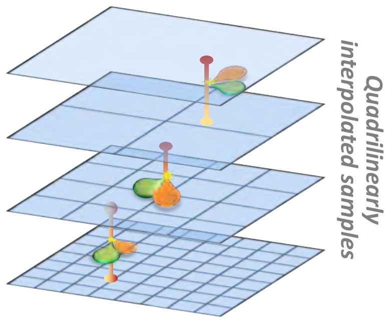
$$\overline{Q}(\mathbf{p}, \mathbf{d}, s, l) = \int_{r\in(s,\mathbf{p},\mathbf{d})} Q(s, s+l, r) dr \tag{4.17}$$

$$\overline{T}(\mathbf{p},\mathbf{d},s,l) = \int_{r\in(s,\mathbf{p},\mathbf{d})} e^{-\tau(s,s+l,r)}dr \tag{4.18}$$

We will see in the next section how we discretize and store these 8D functions inside our voxel MIP-map pyramid.

4.3 MIP-map pre-integration model

Our goal is to discretize our pre-integrated energy function $\overline{Q}(\mathbf{p},\mathbf{d},s,l)$ and pre-integrated transparency function $\overline{T}(\mathbf{p},\mathbf{d},s,l)$ for non-overlapping cubical volumes stored in a set of regular voxel grids of decreasing resolutions organized as a 3D MIP-map pyramid (Fig. right). Thus, we want to pre-compute these functions for a discrete set of positions $\mathbf{p}$ regularly distributed in space, and a set of length of integration l corresponding to the size of the voxel, and thus determining the level in the pyramid. As we want to maintain cubical voxels, we link this length of integration l to the surface area parameter s that we pre-compute for square areas $s = l^2$. Thus, the pre-computation over large pixel integration footprints leads to long pre-integrated lengths, but this allows a storage inside cubical voxels and provides a multiresolution scheme.



Mipmap pyramid of pre-integrated values

The pre-integrated transparency $\overline{T}$ stored per-voxel in our volumetric representation corresponds to the transparency of the volume when seen through a section that has the same size is the voxel it is stored in. Similarly, the pre-integrated in-scattered energy $\overline{Q}$ corresponds to the energy added along a ray by the filtered materials represented inside a voxel. We will see that in practice, we do not want to pre-integrate and store $\overline{Q}$ directly per voxel, but instead we want to store the shading parameters allowing to dynamically compute $\overline{Q}$ at render time (Sec. 4.5).

With this representation, the only parameter remaining per voxel to fully represent $\overline{Q}(\mathbf{p},\mathbf{d},s,l)$ and $\overline{T}(\mathbf{p},\mathbf{d},s,l)$ is the direction of integration $\mathbf{d}$. We propose two models for its representation: a simple anisotropic model storing an omnidirectional approximation of these two pre-integrated functions (Sec. 4.6.3), as well as a more precise model storing a few discretized directions of integration that are interpolated during rendering (Sec. 4.6.4).

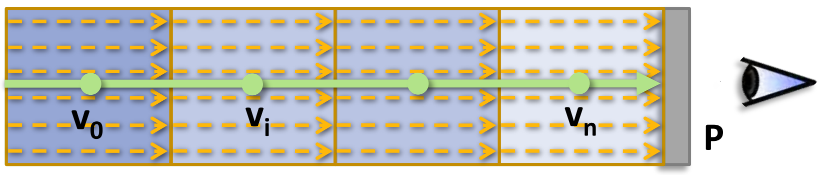


Figure 4.5. Accumulation along a single ray launched for given pixel of the pre-integrated sub-volumes stored inside the MIP-map pyramid.

4.3.1 Quadrilinear interpolation

At render time, one ray is launched per pixel of the screen. Values are sampled along this ray inside the MIP-map pyramid, at a level-of-detail (LOD) chosen in order to get a pre-integrated surface s corresponding to the pixel size. Values for continuous volume positions $\mathbf{p}$ and continuous footprint areas s (and thus continuous length of integration l, since s and l are linked) are reconstructed by interpolation of the values stored inside the MIP-map pyramid. As we will detail in Section 5, we want to rely on the GPU texture filtering hardware for this computation (cf. Section 1.3.4), which limits our reconstruction filter to a quadrilinear interpolation. However, relying on a quadrilinear reconstruction filter supposes that the sampled values can be reconstructed linearly.

We first discuss the interpolation inside a given MIP-map level in order to reconstruct values at continuous positions. If we consider a single ray, the pre-integrated transparency $\overline{T}$ does evolve bilinearly on the plane orthogonal to the ray (parallel to the screen), and thus can be bilinearly interpolated on this plane. Indeed, it is on this plane that the linear integration (pixel footprint averaging) of the pre-integrated sub-paths is done. However, this pre-integrated transparency $\overline{T}$ does not evolve linearly in the direction of the ray. In this direction, the transparency evolves exponentially due to the volume rendering integration, and thus can not be linearly interpolated. The same linear interpolability issue applies to the in-scattered energy $\overline{Q}$ that can be linearly interpolated orthogonally to a ray, but do not in the direction of the ray due to the visibility integration in this direction. Since we don't want to use a totally custom interpolation scheme (to benefit from the hardware trilinear texture interpolation), we rely on a linear interpolation in all axes. In practice, the impact on the rendering precision of this approximation appeared to be negligible in most cases.

The remaining interpolation is the one that needs to be done between two MIP-map levels, in order to reconstuct a continuous surface of integration s (since LOD footprints are quantified), and linked length of integration l. Transparency evolves exponentially with the length of integration l. Thus, since both $\overline{Q}$ and $\overline{T}$ contain a transparency value, these values do not evolve linearly between two MIP-map levels. However, we chose to interpolate them linearly, once again for efficiency reasons, and given the low impact on rendering quality in most cases.

4.3.2 Cone-shaped beams

As presented so far, our pre-integration model only allows the evaluation of the volume rendering integral inside orthographic beams generated by a parallel projection. We will now explain how we rely on the MIP-map pyramid representation detailed in the previous section and storing our pre-integrated energy and visibility functions to approximate cone-shaped beams generated by perspective projection.

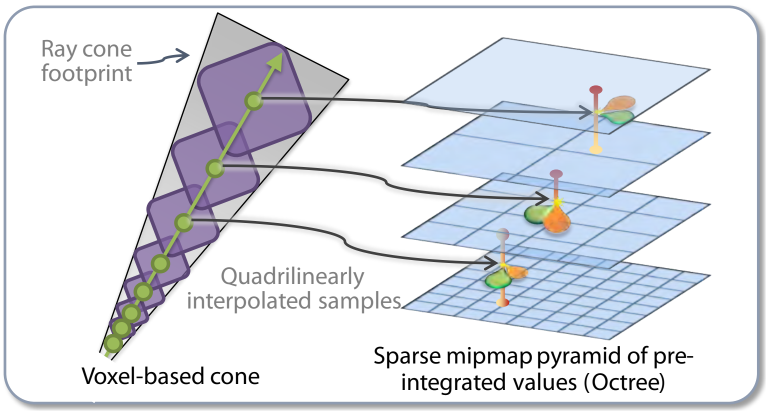

Ray cone footprint
Quadrilinearly interpolated samples
Voxel-based cone
Sparse mipmap pyramid of pre-integrated values (Octree)

Figure 4.6. Illustration of one ray sampling using quadrilinear interpolation in the voxel mipmap pyramid.

In practice, for anti-aliasing, we rely on very thin cones (generated for each pixel) that can be considered as locally cylindrical. We approximate cone-shaped beams by simply varying the LOD used in the MIP-map pyramid when sampling along a ray. The idea is, for each sample, to use the LOD providing the surface of integration s approximating the section of the cone at the point where the sample is taken. This scheme is illustrated in Figure 4.7. As we have seen in the previous section, interpolation must be used between the values stored for discrete area s in the levels of the MIP-map pyramid, in order to get the exact area required for each sample.

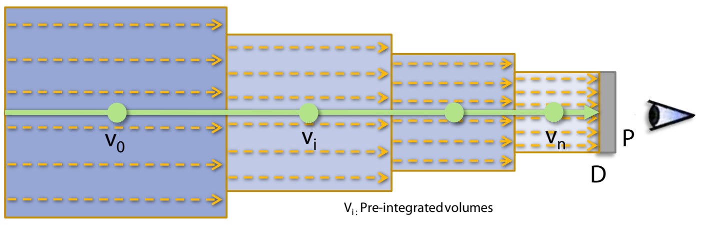


Figure 4.7. Illustration of a cone shaped beam approximated using our voxel-based cubical pre-integration. It is computed efficiently using only a few samples taken along a single ray.

As illustrated in Figure 4.7, the local projection inside pre-integrated sub-volumes corresponds to an orthographic projection, and not a perspective projection as would be expected. This simplification leads to a parallax not taken into account locally in each beam part. This error increases as the size of the pre-integrated volumes increases. In our typical case using thin cones for antialiasing, the small parallax error that can appears inside sub-volumes integrations can be considered as negligible. However, this error increases as much as the aperture of the cones increases.

Cone footprints

The second approximation we make with our approach is to approximate the cone footprint, with cubically shaped sub-volumes aligned with the main axis of our MIP-map representation, as illustrated in Figure 4.7. The first thing to note is that we consider the cone footprint because we represent pixels as disks-shaped elements in order to simplify the representation.

Since values read in the MIP-map pyramid are interpolated both spatially and in resolution between MIP-map levels, the exact footprint of a cone inside the voxel representation is not trivial to estimate. In order to get an idea of this footprint, we display in Figure 4.8 the relative accumulated weight of each voxel (splatted inside the maximum resolution of the MIP-map pyramid) in the quadrilinear interpolation that is done each time they are accessed. Figure 4.8(a) presents these weights inside a volume rendered in 3D for a cone aperture of 11.25°, and Figure 4.8(b) for an aperture of 22.5°. The same experiment is presented for a 2D cut inside a volume in Figure 4.8(c). Even if the footprint of a single cone appears blocky, it in fact approximates a Gaussian distribution centered around the sampling ray. Thus, this approximation converges on a good quality sampling when multiple beams are sent from neighboring pixels. Indeed, sampling a volume with such cones in fact globally ensures energy-conservation. This is illustrated in Figure 4.8(d) that presents the same experiment for a set of rays launched on an entire line of pixels. One can observe that voxels appear regularly sampled, with a weight depending only on their distance to the screen (as expected).

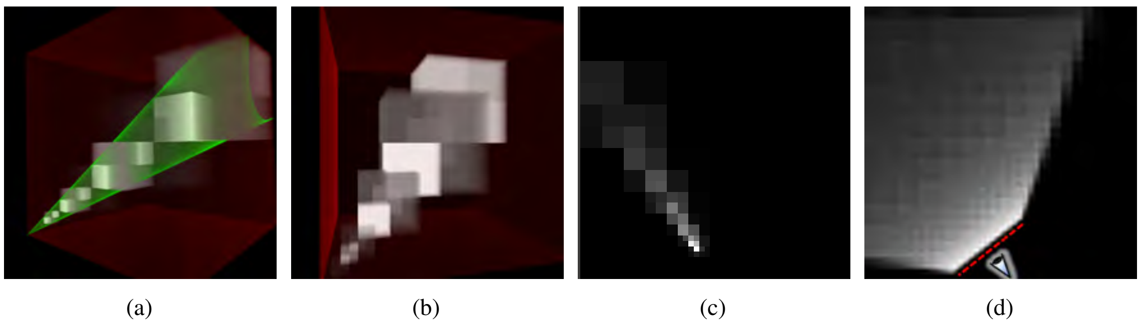


Figure 4.8. Display of the footprint of pre-integrated voxel cones, in 3D inside a volume (a, b) and inside a single slice of a volume for a single ray (c) and for a set of rays launched on a whole line of pixels (d).

4.4 The decorrelation hypothesis

When we defined our pre-integrated cone-tracing model in Section 4.2.1, we relied on two decorrelation hypotheses. These hypotheses on the definition of the scenes we can render accurately define the limits of our model. However, we will see in this section that these hypothesis do not represent a strong restriction on the data we can represent, and that the impact of violating these hypothesis on the rendering quality is very limited in typical scenes.

Correlation-aware filtering is still an open problem that goes beyond the scope of this work and was not handled in this thesis. It is the topic of another thesis currently in progress.

4.4.1 Decorrelation of densities along a beam

As we have seen in Section 4.2.1, we base the definition of both the in-scattered energy q and the total absorption coefficient χ upon a volume density distribution model. The two decorrelation hypotheses are in fact related to a more general hypothesis on this distribution of densities in our scene.

To ensure the two decorrelation hypotheses, it is enough to ensure that inside a given beam generated from a screen pixel, densities coming from the geometrical model are randomly distributed on medium-scale distances (larger than the pre-integrated sub-paths)[2]. With this hypothesis, it is possible to ensure that the energy Q in-scattered in a ray by a given pre-integrated sub-path is decorrelated from the total absorption happening from this sub-path back to the eye (first decorrelation hypothesis used for equation 4.7). It also ensures that successive pre-integrated transparencies T along a ray are decorrelated (second decorrelation hypothesis used for equation 4.9)

4.4.2 Impact of the correlation

This constraint on the distribution of densities inside ray-beams means that in order to produce an accurate rendering, initial densities coming from the geometrical model should be distributed as randomly as possible when seen through a beam. For instance, this hypothesis is violated when observing long alignments of similar densities (partial occupancy), like a wall seen from the edge (as illustrated in Figure 4.10c). However, correlation of densities is correctly handled locally within the pre-integrated volumes, so problems appear only for large-scale correlations, when multiple prefiltered volumes are correlated along a ray. Such a large scale correlation is less likely to happen than local correlations.

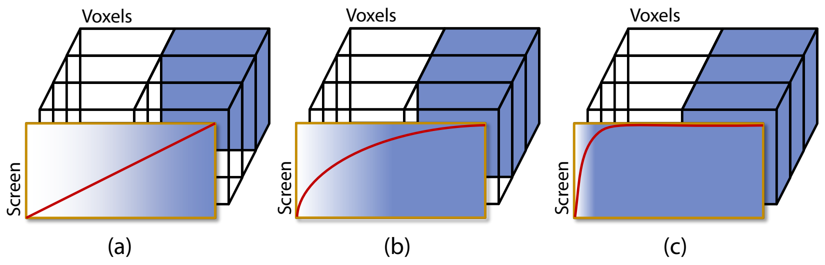


Figure 4.9. Illustration of the result on the screen of volume integration with a lateral per-voxel linear interpolation, with one opaque voxel (a) two correlated opaque voxels (b) and three correlated opaque voxels (c). Correlated cases lead to a non-linear integrated lateral opacity (depicted as a red curve on screens) that diverges from the expected linear result, since (b) and (c) screen results should be the same than (a).

Large scale density correlations lead to an over-estimation of the integrated occlusion ($\alpha = 1 - T$) due to the violation of the second decorrelation hypothesis. This situation is illustrated in Figure 4.9

[2]Shortscale correlations can be accounted in the pre-integration

for a simple exaggerated case. An alignment of linear gradients of opacity in voxel space seen from the edge should result in the same gradient in screen space if integrated correctly. However with our model, such an alignment would result in an exponential gradient in screen space, leading to an over-estimation of the opacity. This error tends to make filtered objects appear a little bit larger in screen-space than they should be. However, since we use thin cones to approximate the energy coming through pixels of the screen, in the worst case the maximum error that can occur due to correlated situation is no more than the width of one pixel on the screen, which is nearly unnoticeable. The problem becomes worse when the aperture of the cones is increased (for instance when approximating a depth-of-field effect as detailed in Section 8.3).

On the other side, the violation of the first decorrelation hypothesis (equation 4.7) would result in occluded energy to be considered. This problem is potentially compensated partly by the over-estimation of the opacity. This can lead to the integration in the final color of a pixel of color contributions coming from occluded materials. But once again, since we rely on thin cones, this error is practically imperceptible in most cases.

4.4.3 Typical cases

In addition to the limited impact of correlation on the rendering precision when using thin cones, this problem of correlation appears only in specific situations that are not very common. Three typical rendering situations (corresponding to the typical scenes we are interested in, cf. section 1.4.2) are depicted in Figure 4.10. In situation (a), short density alignment are encountered, leading to very small rendering errors. In situation (b), rays traverse only a single pre-integrated voxel, leading to zero integration error. Situation (c) is the only situation where important errors can be observed: despite traversing many partly occupied voxels ($\alpha = (1 - T) = 0.5$), the final total opacity of the selected pixel should be $\alpha = 0.5$, but we overestimate it.

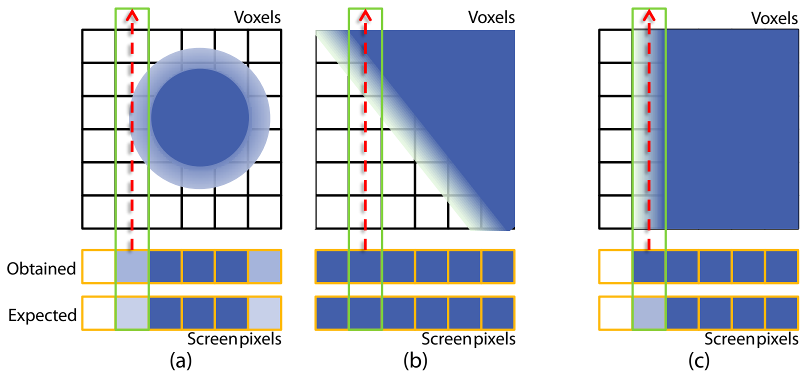


Figure 4.10. Illustration of three typical rendering situations with the comparison between obtained and expected screen-space results for a cylindrical beam launched for each pixel.

4.5 Pre-filtering shading parameters : toward a multiresolution reflectance model

The model as described thus far assumed that the reflected energy j was entirely known before rendering, and we stored its pre-integrated value $\overline{Q}$ in the pre-computed MIP-map representation. In order to allow fully dynamic lighting computation, we will now explain how it is possible to factor the lighting computation out of the pre-integration of j inside $\overline{Q}$, and to pre-integrate only material parameters.

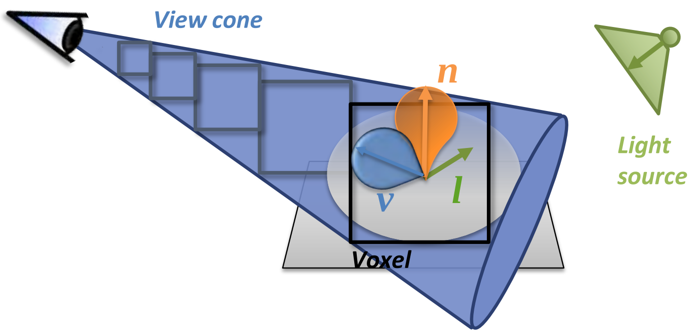


Figure 4.11. Illustration of our local shading model based on pre-filtered surface and material parameters.

Each voxel at a given LOD of the pre-integrated MIP-map pyramid must represent the light behavior of the lower levels - and thus, of the whole scene span it represents. As we have seen in Section 2.1.2, texture pre-filtering techniques rely on the fact that, under some linearity assumptions, it is possible to factor-out from the shading integration some shading parameters and to pre-filter them separately in order to produce antialiased rendering. This operation is possible only when the shading parameters that are factored-out have a linear relationship to the final color, meaning they are involved only linearly in the shading function.

4.5.1 Material parameters

Many different lighting models and material parameters can be used. In this section, we will only describe the parameters used to compute a simple local shading model, assuming single scattering. This is the model we used for most of the examples presented in this thesis. A more complex model allowing indirect lighting will be described in Chapter 9. This simple model is based on a single vector of material colors C_{RGB}, a surface normal N and a global analytical BRDF (Phong). Since the material color has a linear relationship to the final color in the shading computation, it can be freely factored-out and pre-integrated separately. We store a vector of pre-integrated color values $\overline{C}_{RGB}$ for each voxel of our representation and filter it during the MIP-map construction described in Section 4.6. Things are more difficult concerning the normals, as we have seen in previous work (Sec. 2.1.2), a simple vector-based normal description can not be linearly filtered. Thus, we rely on a normal distribution function stored per-voxel and representing the filtered distribution of surface normals in a given volume. The simple normal distribution model we chose to use is described in Section 4.5.2.

More precise material definitions and varying BRDFs models could be used, but this was not the focus of our work.

4.5.2 Simple normal distribution function

While the material color can be freely pre-integrated, it is not the case for the surface normal information N that is needed to compute a local shading model. As we have seen in Section 2.1.2, filtering normal map representations has been a long studied problem. Normal map filtering approaches are based on the idea of representing the normal information using a statistical distribution of normal directions over a surface (called *normal distribution function*, NDF), that can be linearly filtered. Accurate NDF representations (based on spherical harmonics for instance) have been proposed in the past [Fou92b, HSRG07, BN11]. While such representations are totally allowed by our model and are required for a high degree of filtering (when many objects are aggregated), we considered them too memory-intensive for moderate filtering. Instead in our examples, we choose to store only isotropic Gaussian lobes characterized by an average vector D and a standard deviation σ as proposed by

Toksvig [Tok05]. In order to ease the pre-filtering (computation of the MIP-map pyramid) and interpolation, the variance is encoded via the norm $|D|$ such that $\sigma^2 = \frac{1-|D|}{|D|}$. This representation supposes an anisotropic sub-geometry distribution, a moderate filtering and a single filtered face per-voxel.

4.5.3 Local shading model

At render time, the actual pre-integrated in-scattered energy $\overline{Q}$ is computed for each sample taken inside the voxel representation. This computation is done by applying a local shading model (in our examples a simple Phong), based on the vector of material colors $\overline{C}_{RGB}$, the filtered normal distribution (NDF) $\overline{N}$ and the directions to the eye, and to the point light source.

For this, we have to account for the variations in the embedded directions and scalar attributes. For relatively large cone apertures, the span of the cone that is currently accumulating the current voxel must also be taken into account as illustrated in Figure 4.12. As shown in [Fou92b, HSRG07], this can conveniently be translated into convolutions, provided that the elements are decomposed into lobe shapes. In our case, we have to convolve the NDF, the BRDF and the span of the view cone, the first already being represented as Gaussian lobes (cf. previous Section).

We consider the Phong BRDF, *i.e.,* a large diffuse lobe and a specular lobe which can be expressed as Gaussian lobes. Nonetheless, our lighting scheme could be easily extended to any lobe-mixture BRDF. As previously discussed, the NDF can be computed from the length of the averaged normal vector $|N|$ that is stored in the voxels, via the approach proposed by [Tok05] ($\sigma_n^2 = \frac{1-|N|}{|N|}$).

We fit a distribution to the view cone, by observing (Fig. 4.12), that the distribution of directions going from a filtered voxel towards the origin of a view cone is the same as the distribution of directions from the origin of the cone towards the considered voxel. We represent this distribution with a Gaussian lobe of standard deviation $\sigma_v = cos(\psi)$, where ψ is the view cone's aperture.

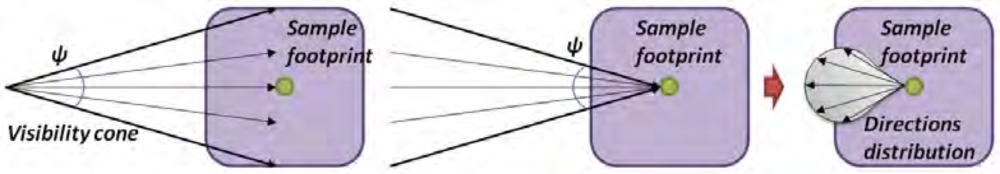


Figure 4.12. Illustration of the direction distribution computed on a filtered surface volume from an incident cone.

4.6 Practical implementation of the model

4.6.1 Initial voxelization of surface geometry

B-rep representation is the standard representation of shapes in computer graphics. Although if our model supports any input representation, an important element to determine is how a B-rep geometry is transformed into a volume density representation. Generally, we represent watertight (closed) geometry with a solid voxelization (filling inside objects volume). Non-watertight geometry (like the leaves of a tree) can also be voxelized simply with a surface voxelization, with each intersected voxel assigned an arbitrary density (usually full).

Our goal is to obtain an average material color C_{RGB}, a normal distribution N and a transparency T for the maximum resolution level of our MIP-map multiresolution representation. Then, we will be able to filter and pre-integrate these parameters inside the inner levels of the MIP-map pyramid (cf. next sections). This voxelization is ensured by our GPU producer model presented in Section 7.4.

For each voxel at maximum resolution, a total density ρ is computed by approximating the union of the volume of the intersection of all objects (in case of watertight geometry) with this voxel. An average absorption coefficient κ' is also estimated simply based on the transparency of all intersecting objects. Taking κ' into account during the voxelization makes it possible to represent and filter semi-transparent objects. Following the density model presented in Section 4.2.1, both κ' and ρ are used to compute a weighted absorption $\kappa = \kappa'\rho$. It is used to compute an average transparency $T = e^{-\kappa\Delta_x}$ for the voxel, with Δ_x the size of a voxel. On the other hand, an average vector of material colors C_{RGB} is computed by averaging the vectors of material colors of every intersecting geometry, scaled by their ratio of density relative to the total density ρ of the voxel. Similarly, we also compute a normal distribution N combining the average normal (in the voxel) of all intersecting geometry scaled by their ratio of density.

In practice, we store C_{RGB} and T as an RGBA value, with the opacity component $\alpha = 1 - T$. The RGB color component is an opacity weighted color $RGB = \alpha C_{RGB}$ in order to provide a correct linear interpolation that does not suffer from bleeding of transparent colors. Following the simple model described in Section 4.5.2, the normal distribution N is stored as a simple 3D vector N_{xyz}. It is also pre-multiplied by α for a better interpolation.

4.6.2 Isotropic and anisotropic voxel representations

In the MIP-map model for pre-integrated cone tracing presented in Section 4.3, the discretization of the parameter **d** defining the direction of pre-integration inside a given voxel was not defined. We propose two practical implementations for the discretization of this parameter. The first implementation neglect this direction parameter and considers isotropic per-voxel values. The second proposes a simple approximation of anisotropic per-voxel values based on a discretization along the six principal directions.

4.6.3 Compact isotropic voxels

The advantage of the isotropic voxel representation is its compactness, since only one value needs to be stored for each parameter maintained per-voxel. This is the implementation we use for nearly all examples shown in this thesis, apart in Chapter 9.

This representation is a rough approximation of our theoretical pre-integration model. The MIP-map pyramid is built from top to bottom, starting from the highest resolution. Instead of pre-computing the actual volume integration model for the parameters $\overline{T}$, $\overline{C}_{RGB}$ and $\overline{N}$ (the normal distribution), we neglect the visibility inside pre-integrated volumes. Thus, we simply apply a traditional MIP-mapping scheme. In order to compute a lower resolution voxel from the immediately higher resolution ones, we simply average the parameters (presented in the previous section) of the 8 higher resolution voxels.

In practice, this implementation works pretty well for antialiasing of primary rendering rays that are relatively thin. However, it becomes insufficient when large cones are required. This is the case for the depth-of-field model presented in Section 8.3.2, or the indirect lighting model presented in Section 9.

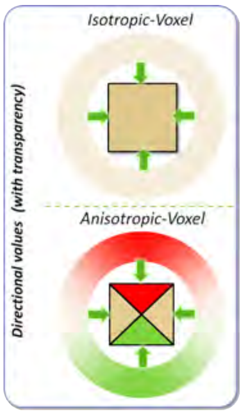
Rendering with isotropic voxels

The transparency T, and also the opacity $\alpha = 1 - T$, are values that depend on the length of the traversed medium. With the approach we just described, the opacity α stored in a voxel at a given resolution in the MIP-map pyramid is not defined for the scale of this voxel. Due to the averaging we use to compute the MIP-maps, it is defined for the scale Δ_x of a voxel at the highest resolution. The nice thing is that this allows us to adapt the opacity sampled inside our representation during ray-casting to the actual step taken between successive samples and corresponding to the actual length of integration of each sub-volume. Thus, if opacity values are sampled along a ray separated by a distance Δ'_x, these values must be corrected with $\alpha' = (1 - \alpha)^{\Delta'_x/\Delta_x}$.

4.6.4 Anisotropic voxels for improved cone-tracing

While the voxel representation we describedso far already provides a good approximation of the visibility when tracing thin cones, it poses several quality problems for large cones that we propose to address. To do so, we propose a directional voxel model that implements our pre-integrated geometry model presented in Section 4.3 much more precisely.

The first problem with the isotropic model known as the two red-green walls problem is illustrated in Figure 4.13. This problem is linked to the way we rely on averaged values in the octree as a pre-integrated visibility value for a given volume. With this approach, when two opaque voxels with different colors (or any other value) -coming from two plain walls for instance- get averaged in the upper level of the octree, their colors get mixed as if the two walls were semi-transparent. The same problem occurs for opacity, when a set of 2x2x2 voxels that is half filled with opaque voxels and half filled with fully transparent ones, the resulting averaged voxel will be half-transparent. This wrong estimation of the opacity and the wrong mix of materials can cause visible artefacts when using too large cones (as it is the case for the indirect lighting approach presented in Chapter 9).



Ideally, we would like to accurately encode the directional parameters $\mathbf{d}$ for both the $\overline{T}(\mathbf{p}, \mathbf{d}, s, l)$ and the $\overline{Q}(\mathbf{p}, \mathbf{d}, s, l)$ functions (cf. Section 4.2) inside our MIP-map representation. This would allow a mipmapped voxel to be fully opaque when seen normally to the half set of opaque voxels, and half-transparent when seen tangentially to the set (Fig. right). In the two walls case, we would like the mipmapped voxel to be red when seen from the red side, green when seen from the green side, and a mix of the two when seen tangentially.

Storing values for a precise discretization of $\mathbf{d}$ would be very costly. To approach that goal, we propose an anisotropic voxel representation that is built during the MIP-mapping process, when building or updating the octree with irradiance values. Instead of a single channel of non-directional values, voxels store 6 channels of directional values, one per major direction. As illustrated in Figure 4.13, a directional value is computed by doing a step of volumetric integration in depth, and then averaging the 4 directional values to get the resulting value for one given direction. At render time, the voxel value is retrieved by linearly interpolating the values from the 3 directions closest to the direction the voxel is viewed.

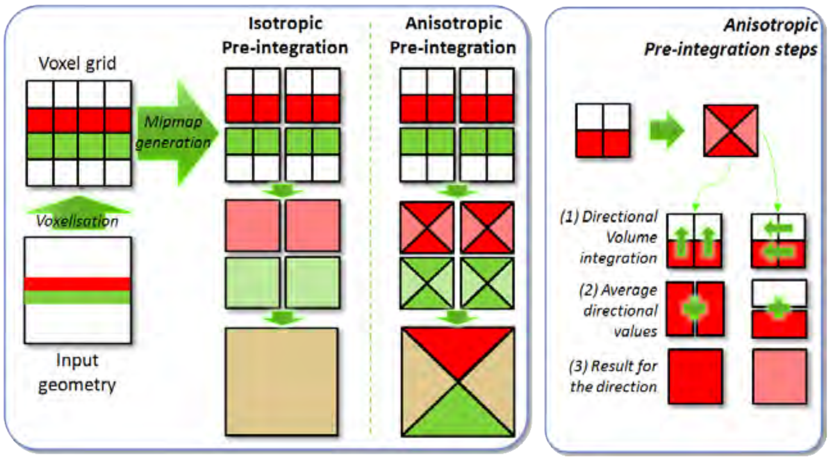

Voxel grid
Isotropic Pre-integration
Anisotropic Pre-integration
Mipmap generation
Voxelisation
Input geometry
Anisotropic Pre-integration steps
(1) Directional Volume integration
(2) Average directional values
(3) Result for the direction

Figure 4.13. Ilustration of the voxel MIP-mapping process without (left) and with (right) anisotropic voxels and directional pre-integration. The directional integration steps is illustrated in the right green box.

We chose this simple representation instead of a more complicated one because it is fully interpolable and can be integrated easily to our voxel storage (cf. Chapter 5). In addition, this directional representation needs only to be used for voxels that are not at the full resolution and are not located in the last level of the octree. Thus, storing directional values for all the properties present in our structure only increases the memory consumption by 1.5×.

A comparison of image quality between isotropic and anisotropic pre-integrated voxel representations is presented in Figure 4.14. Voxels are sampled directly from a proxy surface geometry and voxel view direction is provided by the normal of the proxy geometry. One can observe that many more details can be captured and reproduced thanks to the isotropic voxel representation.

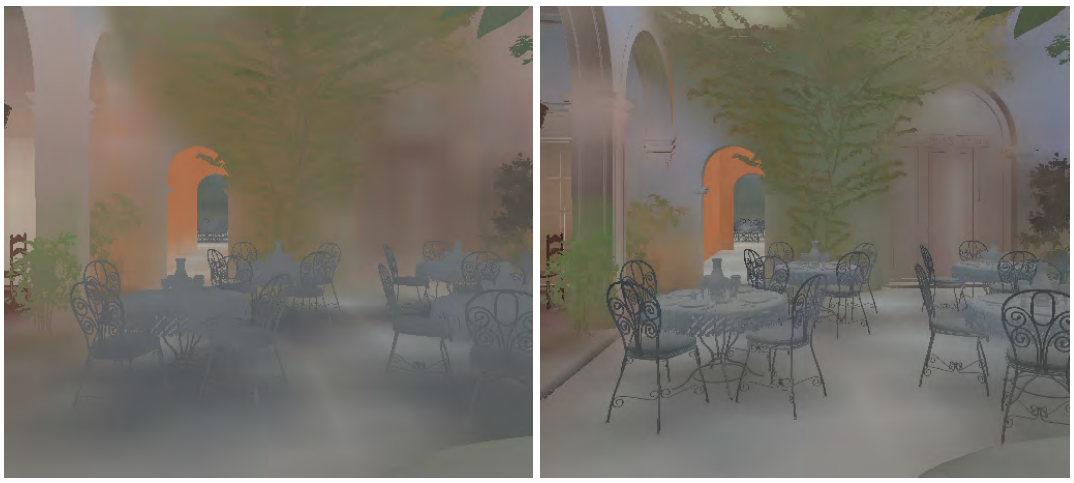


Figure 4.14. Image comparison with direct voxel sampling on a proxy surface geometry from the octree between isotropic voxels (left) and anisotropic voxels (right). Observed voxel direction is given by the normal of the geometry.

4.7 Conclusion

In this chapter, we presented our new voxel-based filtered geometry representation together with its multiresolution pre-integration scheme. This representation is the foundation for our approximate cone-tracing approach providing a fast estimation of visibility and light transport integration inside a beam using only a single ray. This voxel model will be stored inside our sparse hierarchical GPU data structure presented in chapter 5, and the cone tracing scheme will be implemented by our rendering algorithm presented in chapter 6.

Even if our pre-integrated cone tracing model does not always produce very precise results, especially in case of large cones, it provides the great advantage of always computing smooth results. Indeed, the main problem with stochastic supersampling approaches (Sec. 2.1.1) is that they generate a lot of noise, especially when evaluating large cones (for indirect lighting for instance), which is a lot worse for realism than a global imprecision. We will see in sections 8.3 how this nice property can be exploited to very efficiently estimate soft shadows and depth-of-field effects. We will also demonstrate in chapter 9 how our voxel cone tracing model can be used to estimate indirect lighting in real-time.

Chapter 5

Data Structure

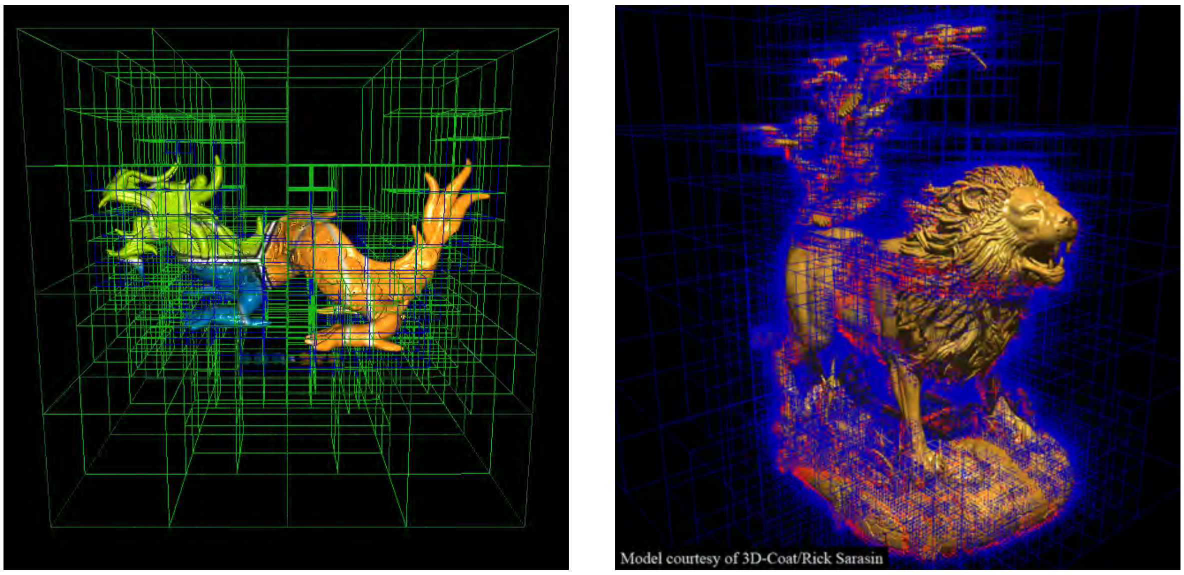


Figure 5.1. Spatial partitioning of the octree structure. *Left:* A voxelized version of the Stanford XYZRGB-Dragon model (1024^3 voxels) rendered at around $80FPS$. *Right:* A 2048^3 voxels lion model modelized directly in voxels using 3D-Coat [Shp11] and rendered at $60 - 80FPS$ with GigaVoxels on NVIDIA GTX 480.

In this chapter, we present our generalized data structure and discuss its condensed representation on the GPU. The design of a data structure suitable for manipulating very large data sets is one of the three key elements (along with the rendering algorithm and the out-of-core streaming mechanism) of our voxel rendering system.

Our structure provides a compact storage of the dataset, allows a fast traversal for rendering operations (Sec. 6) and is easy to modify, allowing interactive updates (Sec. 7). It provides a fast logarithmic complexity access to the entire dataset and all MIP-map resolutions. It is also designed to allow the storage of multiple scalar or vector material and shading parameters in order to allow the implementation of arbitrary shading models.

5.1 The octree-based voxel MIP-map pyramid: A sparse multi-resolution structure

An adaptive space subdivision is essential to render large volumetric scenes and to adapt to memory limited environments. Such a hierarchical representation 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 reduces enormously the memory consumption and avoids storing the entire information on the GPU.

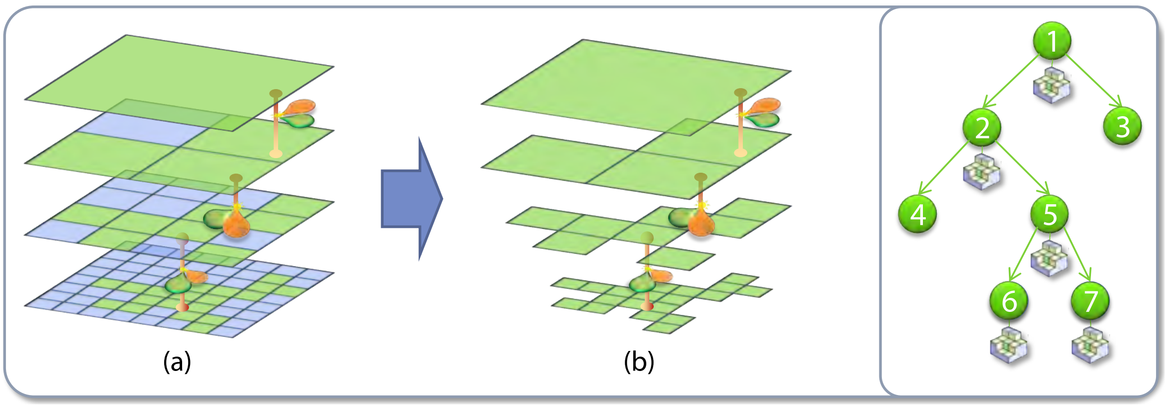


Figure 5.2. Illustration in 2D of our 3D sparse MIP-map pyramid representation, (a) Full MIP-map pyramid, (b) Sparse MIP-map and an illustrative corresponding octree structure.

5.1.1 Octree + Bricks representation

The structure we designed is a combined structure, mixing both the advantages of a hierarchical representation and a regular structure. It is based on a generalized octree structure, for hierarchical space subdivision (similarly to [LHN05b]), with a small voxel volume associated with each non-empty/non-constant node. These small volumes are low resolution regular grids we refer to as *bricks*. A *brick* is a small voxel grid of some predefined size M^3 (usually $M = 8$) that approximates the part of the original volume that corresponds to an octree node (similarly to [CB04a, GMAG08]). For example, the brick for a root node would be an M^3 voxel approximation of the entire data set. This kind of scheme corresponds to a hierarchical bricking scheme as described in Section 2.4.4.

This data representation thus combines the memory efficiency of an adapted structure with small 3D texture units that can be efficiently cached and allows us to exploit the hardware support of 3D textures to accelerate rendering (Sec. 6). Figure 5.3 summarizes the data structure of our approach and will be of help during the following discussion. In 2007 when it was designed and presented, we were among the first to propose such a kind of structure that can be efficiently traversed entirely on the GPU.

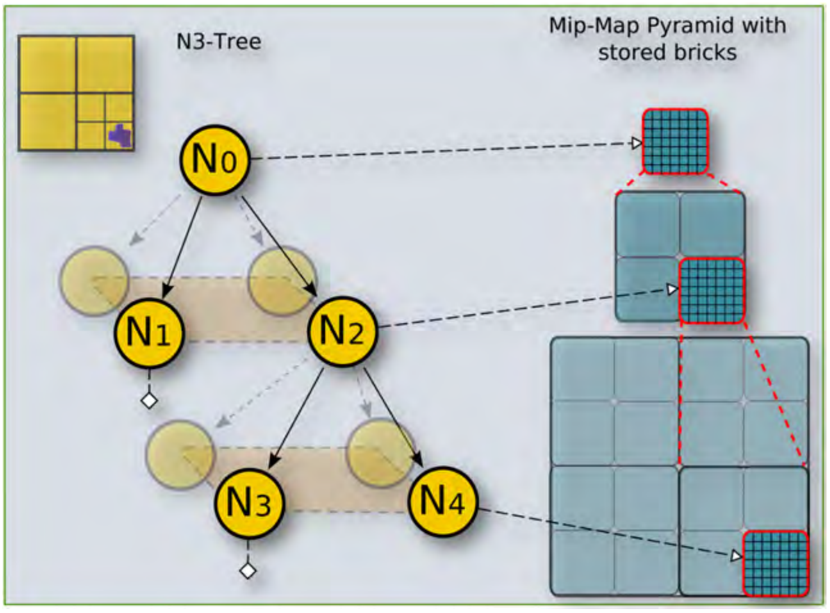


Figure 5.3. Our sparse voxel octree data structure. This structure stores a whole voxel scene or object filtered at multiple resolutions. Bricks are referenced by octree nodes providing a sparse MipMap pyramid of the voxels.

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 3D textures and, thus, benefit from hardware-based interpolation and cache coherence. Another advantage is the constant size of the node and brick data types. Consequently, it is simple to store them densely in memory pools on the GPU (Sec. 5.2.1). This facilitates the update mechanisms which are crucial to ensure the presence of the data needed to produce the output image (Ch. 7).

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. The bricks linked by interior nodes of the octree contains filtered data obtained by downsampling higher resolution data located in child nodes. This filtered data will also enable us to perform high-quality filtering via mipmaps (cf. Section 4.2).

5.1.2 Constant regions and frequency-based compression

Instead of storing a brick pointer, we further allow each node of the octree to store a constant value. Storing a single value in the case of almost homogeneous regions (empty or core) reduces memory requirements enormously. In addition, during rendering we can avoid traversing constant regions by directly computing the region's contribution by using the single constant value.

Boada et al. [BNS01] presented a similar approach with the same data structure but introduced the idea of using a frequency-based level-of-detail selection based on a data homogeneity measure when building the set of octree nodes to be stored for rendering (the cut).

More generally, the maximum level of subdivision of the octree is determined by the frequency of variation of the data (homogeneity) in the spirit of [BNS01]. Indeed, low frequency data like smooth gradients can be accurately reconstructed by linear interpolation of low resolution voxel data, and do not require a deep subdivision of the octree. Such data variations are detected during the building of the structure (cf. Chapter 7) and used as the main criteria to determine the level of subdivision of the tree.

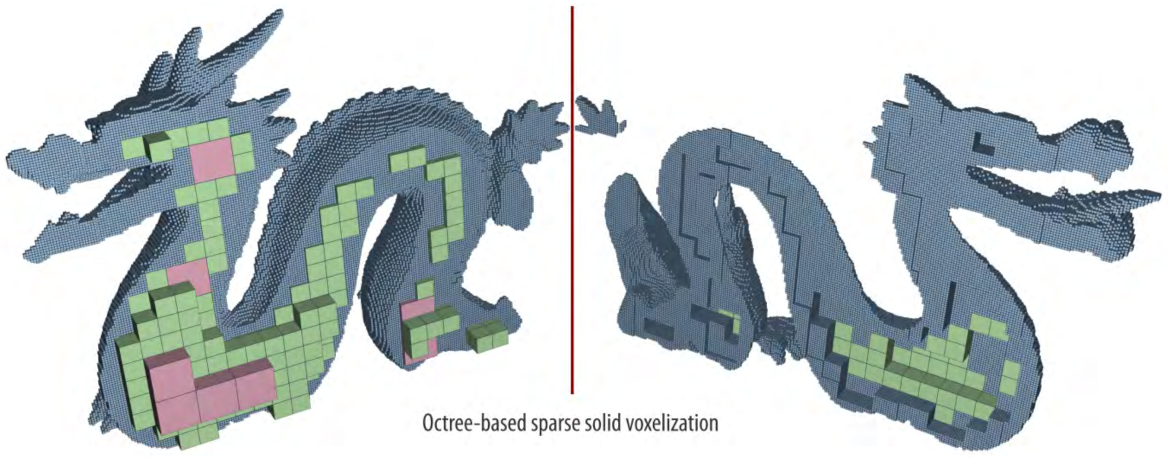


Figure 5.4. Illustration of constant regions encoding inside a voxelized mesh. *Source: [SS10]*

5.1.3 The N^3-tree : A Generalized Octree

The organization we just presented is flexible enough to rely on generalized octrees called N^3*-trees* (similar to [LHN05a]), partitioning space and subdivided through an adaptive LOD mechanism driven by visibility, data frequency and distance to the current point of view. Low-frequency regions are not subdivided (LOD of visible leaves is the coarsest of distance-based and content-based LODs).

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 algorithm's 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.

5.1.4 The bricks

The 3D bricks linked by the nodes at each level of the octree are used to provide a fast and interpolated access to the actual voxel data. They are stored inside texture memory allowing them to benefit from the support of hardware accelerated interpolation and 3D-local cache. Each brick voxel can store multiple channels of values used for rendering, usually at least a color, an opacity and a normal distribution (for shading). As illustrated in Figure 5.5, brick voxels can be located either at the center of octree nodes, or at the corners.

Brick borders and voxel centering

In order to ensure a correct trilinear interpolation by the hardware at brick boundaries, some voxels must be replicated between adjacent bricks. Replication-free interpolation schemes have been proposed in the past [LLY06], but this comes at the cost of a high runtime access overhead that we want to avoid. In the case of node-centered voxels, an additional one voxel border must be added around all the bricks, replicating neighboring bricks voxels. This border introduces a lot of redundancy in the stored voxel data as illustrated in Figure 5.6. The redundancy is especially important when small bricks are used, and can induce an important storage overhead. Thus, such a configuration must be used with sufficiently large bricks, in our case we use bricks of at least 8^3 voxels (without border).

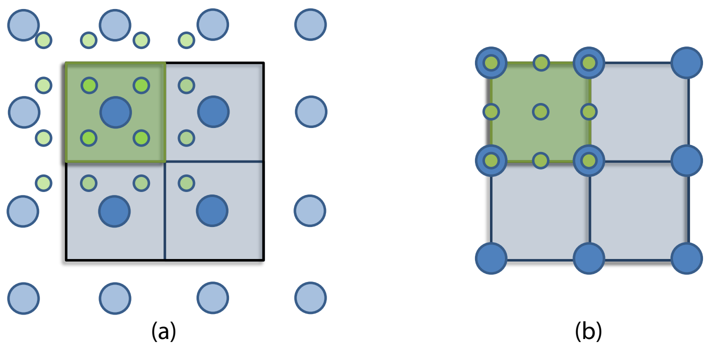


Figure 5.5. Comparison between bricks with voxels on node centers (a) and voxels on node corners (b). We show two bricks (blue an green) et different levels of the tree, brick voxels are represented by colored circles.

Corner-centered voxels allow us to reduce this overhead by providing all data necessary for a correct interpolation inside octree nodes, but using one less voxel on each dimension. Such a brick configuration allows using much smaller bricks and so provides better data compaction and empty-space skipping. However, corner-centered voxels induce a more complicated filtering process when computing the MIP-map pyramid. While MIP-mapping node-centered voxels can be done using a simple box filter, corner-centered voxels require the use of a 3^3-Gaussian weighting kernel which, for our case, is an optimal reconstruction filter [FP02a].

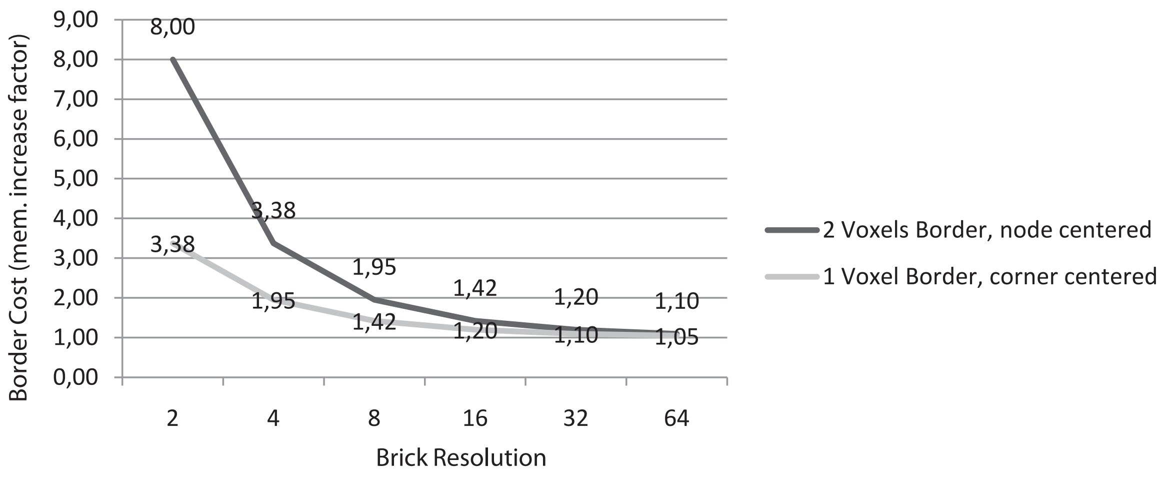


Figure 5.6. Factor of increase in size induced by the replication of voxels between bricks depending on the brick resolution and both in case of node-centered and corner-centered configurations.

Brick neighbors for interpolation

In order to ensure a correct interpolation between data located in different neighboring nodes of the octree, special care must be taken with the voxels located on the border octree nodes. Indeed, as illustrated in Figure 5.7, if any voxel located on the boundary of a brick is not "empty" (or does not have the same value as the constant value of the neighboring node), then a brick must be attached to the neighboring node at the same LOD in order to ensure a correct interpolation of any sample taken inside the neighboring node near the boundary. Thus, within this constraint, an unsubdivided neighbor could get subdivided in order to provide a brick at the same resolution. Note that this constraint can be relaxed with very few and imperceptible visual artifacts by allowing the neighboring node to be subdivided one level higher than the node containing the boundary voxel, providing a brick at half the resolution of the neighboring brick.

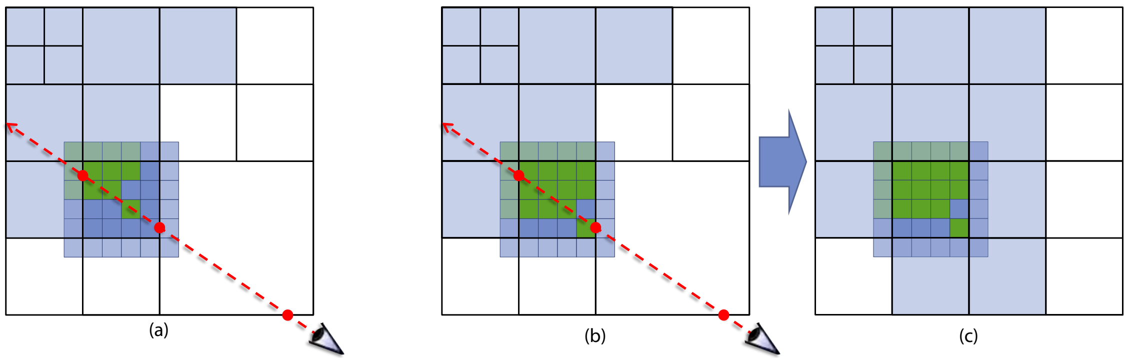


Figure 5.7. Illustration of a configuration where no neighboring brick is needed (a), nodes containing a brick are colored blue, "non-empty" voxels inside the displayed brick are colored green and "empty" voxels are colored blue. Illustration of a configuration where additional neighboring bricks are needed (b) due to voxels located on the boundary of the node. Modified octree with necessary octree subdivision and new bricks (c).

5.2 GPU implementation

Figure 5.8 summarizes the data structure of our approach and might be helpful during the following discussion of the implementation.

Our octree structure is implemented as a pointer-based tree. Other approaches like [GMAG08] (developed in parallel with ours) requires eight pointers to be stored for each node (in the case of an octree) for the child, as well as pointers to neighboring nodes. Our structure is much more compact and requires only a single pointer per node for the child. To make this possible, nodes are organized into blocks of N^3 nodes sharing the same parent node and stored contiguously in memory. 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 is coherence largely improved during the traversal, but also memory requirements are largely reduced. The main focus of our implementation has been to keep the structure as compact and simple as possible, in order to provide a minimal memory occupancy, good caching behavior and allow fast incremental updates (Sec. 7).

Even though our structure does not exhibit neighbor pointers between adjacent nodes, we will show in Section 6.1 that an efficient traversal remains possible. Each tree node also contains a pointer towards a brick or is indicated as constant/empty space (homogeneous volume).

5.2.1 Structure storage: the pools

Our data structure is stored in pre-allocated GPU memory regions we call *pools*. These pools are used as application managed caches (cf. Section 7.3), 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. The *node pool* stores the octree as a pointer-based tree and the bricks are organized into a *brick pool*.

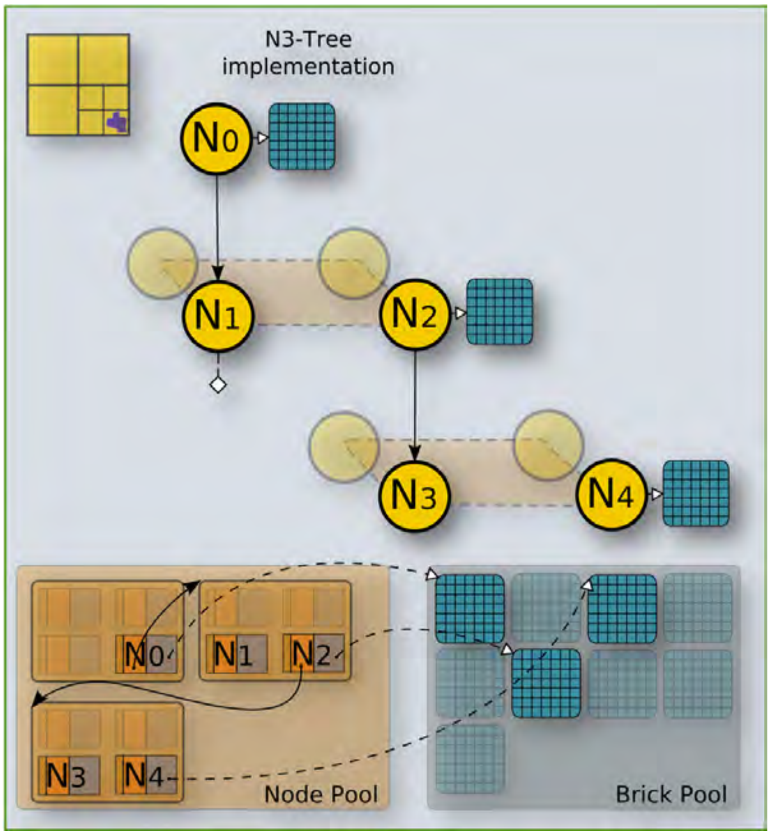


Figure 5.8. Our sparse voxel octree structure and its storage inside video memory. Nodes of the octree are stored in $2 \times 2 \times 2$ node tiles inside a *node pool* located in video memory, Bricks are stored inside a *brick pool* implemented as a large 3D texture.

Node pool

The node pool is implemented in global linear GPU memory (cf. Section 1.3). It is accessed from the GPU (a CUDA kernel or a Shader) through an 1D linear cache that can be either the texture cache on Pre-Fermi GPUs or the generic global cache hierarchy. In order to provide a good cache behavior when accessing the structure, a single entity in the node cache actually regroups the $N \times N \times N$ sub-nodes of the same parent node, instead of storing each node separately. We call these $2 \times 2 \times 2$ nodes organization a *node tile*. To enhance traversal efficiency during rendering, both parts of a node description (sub-node pointer and data, cf. Figure 5.10) 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. This kind of memory organization, where the members of a set of structures are grouped and stored continuously in memory is a classical approach in parallel computing. This memory organization is called *Structure Of Arrays* (SOA), as opposed to *Array Of Structures* (AOS) that interleaves structure members in memory.

Brick pool

On the other hand, the brick pool is implemented in texture memory (cf. Section 1.3), in order to be able to use hardware texture interpolation, 3D addressing, as well as a caching optimized for 3D locality (cf. Section A.1). Brick voxels can store multiple scalar or vector values (Color, normal, texture coordinates, material information ...). These values must be linearly interpolable in order to ensure a correct reconstruction when sampled during rendering. Each value is stored in separate "layer" of the brick pool, similarly to the memory organization used for the node pool.

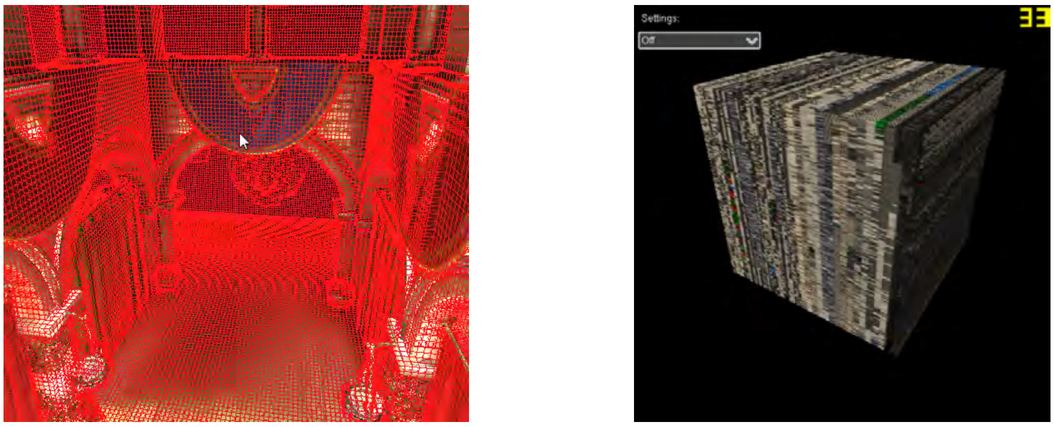


Figure 5.9. Illustration of the packing of non-empty bricks of our sparse voxle octree inside a 3D texture-based brick pool.

5.2.2 Octree nodes encoding

Although the octree structure memory occupation is much lower than for the bricks, having a compact representation is still very important for rendering, in order to reduce the amount of data read during the traversal of the tree, and to maximize cache efficiency. In addition, updates to the structure (Chap. 7) can also be achieved more efficiently and with less bandwidth usage.

Figure 5.10 shows the data elements in an octree node. Each node is composed of a data entry, and a sub-node (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 N^3 (usually eight) sub-nodes. It is exactly this property that allows us to only rely on a single child pointer. Because all sub-nodes are stored contiguously in memory, one can address any single one by adding a constant offset to the pointer.

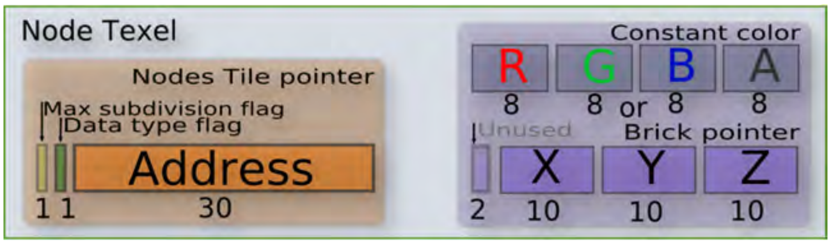


Figure 5.10. Compact octree node encoding into two 32bits values.

Our structure produces a very compact octree encoding as each node is represented by only two 32bit integer values. The first integer is used to define the octree structure, the second to associate volume data. Of the first, one bit indicates whether a node is *terminal*, which means *refined to a maximum*, or 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 sub-nodes.

Pointers representation and scalability

One can see the 30 bit pointer representation we are using as a limiting factor for the scalability of our approach. Indeed, such pointers when used as traditional pointers would only allow us to address 1GB of data. That means that the size of the *pools* we are manipulating would be limited to 1GB, while current generation video cards loads between 1GB and 4GB of video memory. This restriction would not be currently a problem for the *node pool* that does not require big storage compared to the *brick pool* that actually stores voxel data. It would indeed be a problem for the *brick pool* that should be able to fill nearly all the available video memory. In addition, we would like our implementation choices to stay valid for some years with the amount of embedded video memory increasing quickly.

To overcome this problem and increase the number of elements we can address, we chose not to directly store GPU memory addresses (in Bytes) of individual elements (nodes or voxels). Instead, we chose to reference the groups of elements stored inside the *pools*. These groups are *node tiles* for child pointers and *bricks* of voxels for the brick pointer.

The actual encoding we are using depends on the natural addressing of the actual storage (Texture or video memory) of the referenced data:

- **Node pointer**: the index in the pool of the *node tiles* is stored. Each *node tiles* is made of N^3 nodes and each node is $8Bytes$. For an octree ($N = 2$), our representation can manipulate *node pools* of $2^{30} \times 8 \times 2^3 Byte = 64GB$.

- **Brick pointer**: bricks are stored in pools located in a 3D texture memory. 3D texture memory is addressed using a three coordinates vector representing the XYZ position of the voxel in the texture. We chose to keep this 3D representation in order to maintain a fast conversion into natural addresses. A brick pointer encodes the 3D index of a brick in the brick pool with three 10bit XYZ coordinates as illustrated in figure 5.10. In case of bricks of size 8^3 and containing $4Bytes$ voxels (`RGBA8`), this representation allows to manipulate *brick pools* of $2^{30} \times 8^3 \times 4Byte = 2TB$.

5.3 Structure characteristics

In order to analyze the main characteristics in terms of compaction of voxel data, we present the analysis of the structure generated for 2 typical scenes, the Sponza and the Coral scene presented in the right figure. Figure 5.11 presents the memory occupancy of both the bricks (left) and the octree structure (right) depending on the resolution of the brick, and the voxel centering scheme that is used (node-centered or corner-centered voxels, cf. Section 5.1.4). We limited the octree subdivision to the level 9, providing a virtual resolution of 512^3. One RGBA vector with one Byte per component (RGBA8) is stored per voxel. With a dense storage inside a regular grid, a MIP-map hierarchy with a maximum resolution of 512^3 storing RGBA8 values per voxel takes 1023MB in memory.

For these two typical scenes, we observe that the optimal configuration in terms of storage cost is the use of 4^3 voxels bricks with corner-centered voxels. For the Sponza scene, this configuration leads to a total storage cost (bricks+octree) of $68.48MB + 1.93MB = 70.41MB$ for the whole sparse MIP-map pyramid. In this case, our structure requires only 6.88% of the storage required by a dense representation. For the Coral scene, our representation requires only 4.9% of the storage of a dense representation. We also observe that in this configuration, for corner-centered voxels, the overhead induced by the border (for correct inter-brick interpolation) is around 95%, thus nearly doubling the memory requirement.

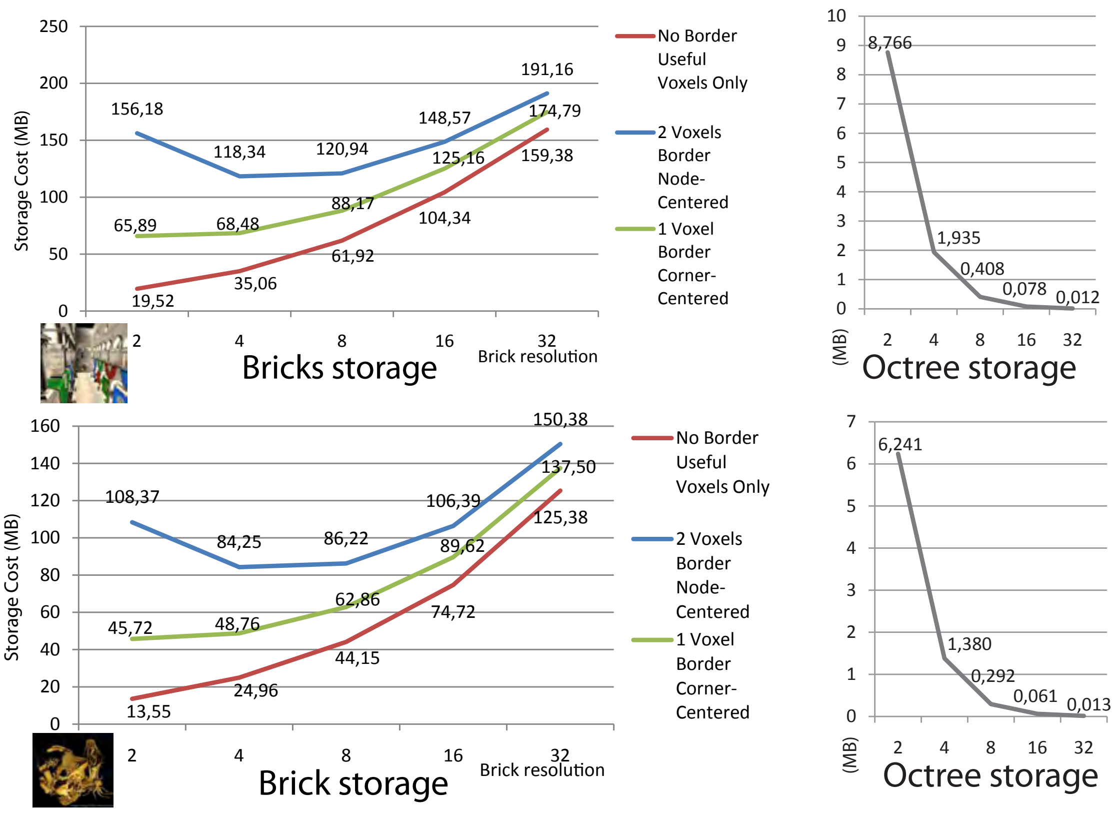


Figure 5.11. Comparison of the storage cost of our octree + brick structure depending on the resolution of the bricks and the voxel centering scheme for the Sponza scene (top) and the Coral scene (bottom).

Chapter 6
Rendering

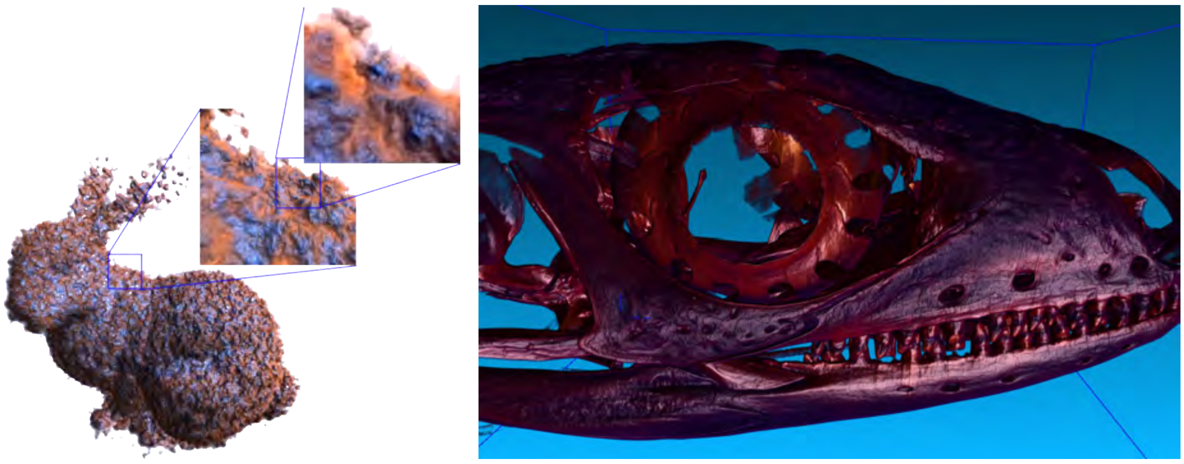


Figure 6.1. Examples of renderings done with GigaVoxels. *Left:* Real-time rendering of a voxelized mesh object enhanced with a procedural noise (30FPS on NVIDIA 8800GTS). *Right:* Interactive rendering (20FPS on GTX280) of a voxel based medical dataset generated from CT scan (2048^3, 32GB on disc).

In this chapter, we will discuss how to render the volume representation of a scene stored inside the octree-based structure described in the previous chapter. This rendering implements, for each pixel of the screen, our pre-integrated voxel cone tracing algorithm presented in Chapter 4. The rendering as described here assumes that the data structure contains all data necessary for rendering. Refinement and update mechanisms will be discussed in the next chapter.

6.1 Hierarchical volume ray-casting

The color of each pixel is evaluated by traversing the structure using volume ray-casting [KW03a, RGW+03, Sch05], executed on the GPU using either a CUDA kernel or a fragment shader. In order to proceed in parallel, we generate one thread per screen pixel, each tracking one ray through the structure as illustrated in Figure 6.2. Each ray implements our cone tracing model described in Section 4.2 in order to evaluate the volume-rendering integral (cf. Section 2.3.1) and to produce an accurate antialiased rendering as illustrated in Figure 6.3.

Basically, view rays are initialized on the near plane of the current view, covering the whole screen and traversing the scene front to back. Starting from the near plane, 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 (similarly to early ray termination in [Sch05]).

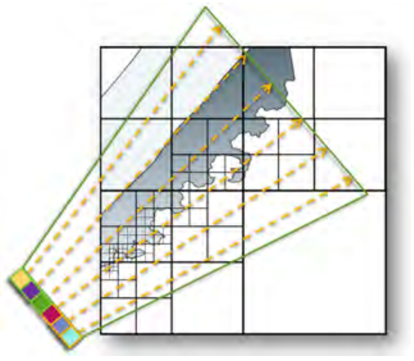


Figure 6.2. Illustration of the volume ray-casting process launching one ray per pixel and traversing the hierarchical structure.

Alternatively, when adding volumetric details on a triangle mesh for instance, a proxy surface can be used to initialize the view rays. An approximate geometry that contains the non-empty areas of the volume can be rasterized, delivering the origins and directions of the rays. This also provides a starting point and an exit point for each ray, speeding-up the skipping of empty spaces. This proxy geometry can also simply be the bounding box of the scene, when observed from the outside.

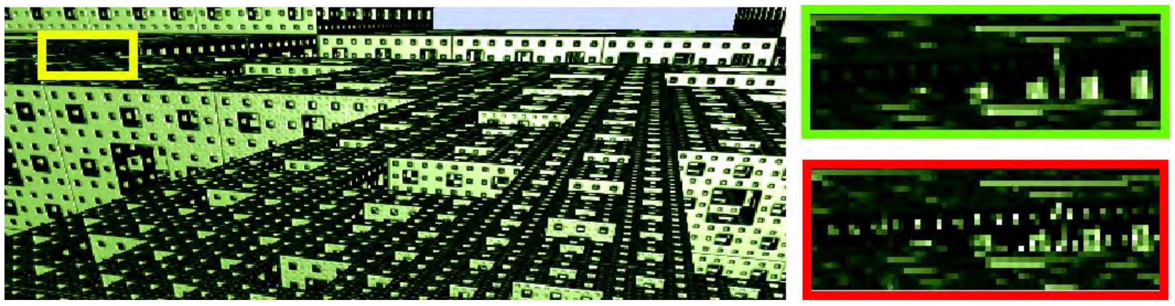


Figure 6.3. Our method (top) based on pre-integrated voxel cone tracing does not show the aliasing artifacts of standard ray-casting with one ray per pixel (bottom).

6.1.1 Global scheme

Each view ray implements the cone-tracing algorithm presented in Section 4.2 that provides us with an accurate multiresolution rendering scheme. Our data structure (presented in Chapter 5) encodes a sparse MIP-map pyramid in order to provide different levels of details (LOD). As we have seen in Chapter 4, each such level contains pre-integrated volumes at a different resolution.

To trace a cone and compute the color of a pixel, successive samples need to be taken along the view ray inside our sparse MIP-map pyramid. The LOD (Level-Of-Detail) of each sample is chosen in order to account for the correct pre-integrated volume, and correctly approximate the footprint of the cone as detailed in Section 4.2 and illustrated in Figure 6.4.

For each sample, pre-integrated geometry and material parameters $\overline{C}_{RGB}$ and $\overline{N}$ are transformed (through the application of the local shading model presented in Section 4.5.3) into a pre-integrated in-scattered energy $\overline{Q}$ that can be accumulated. The accumulation scheme implements our discrete front to back composition scheme based on $\overline{Q}$ and $\overline{T}$ and described in Section 4.2.2.

As detailed in Chapter 5, our sparse data structure is made of an octree with a brick or a constant value linked to each node. The main problem when implementing this scheme is to traverse efficiently the octree structure as well as the associated bricks on the GPU.

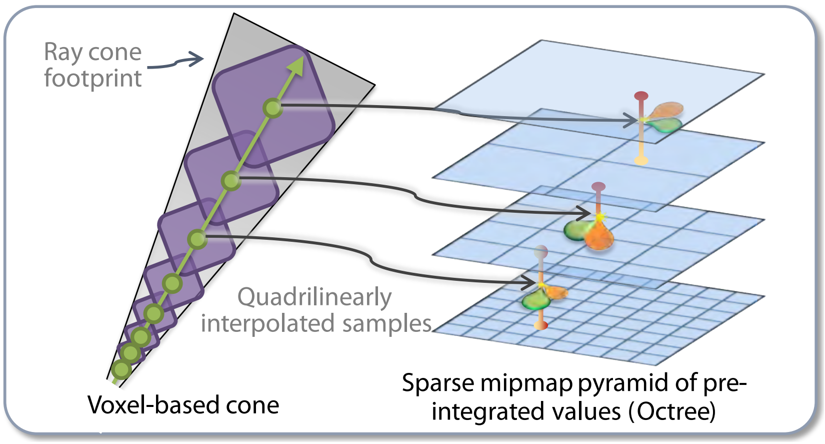

Ray cone footprint
Quadrilinearly interpolated samples
Voxel-based cone
Sparse mipmap pyramid of pre-integrated values (Octree)

Figure 6.4. Illustration of one ray sampling using quadrilinear interpolation in the voxel mipmap pyramid.

6.1.2 Traversing the structure

Our goal is to take advantage of our octree based data structure to collect efficiently successive samples along a ray, and quickly skip constant or empty regions. The idea is for each ray, to traverse the octree structure, find each successively intersected brick of voxel at the correct resolution, and sample inside it. Our octree ray traversal algorithm builds upon the previous work on spatial data structure traversal for GPU ray-tracing applications that is presented in Section 2.4.2.

On the CPU, an efficient way to traverse a hierarchical data structure is to rely on a recursive traversal based on a stack. Until recently, it was not possible to implement such a stack efficiently on the GPU, due to the lack of fast and large enough dynamically indexable memory accessible from a shader (or a CUDA kernel). We tested this approach on SM4 generation GPUs (cf. Section 1.3) which provide a dynamically indexable *local memory*, but it was totally inefficient. On the same hardware generation, we also tried approaches using a stack stored in the *shared memory* [NVI11a] and shared between tiles of rays. But this approach also appeared slower than the simpler approach we adopted.

Instead, we use a stackless algorithm that descends iteratively from the tree root. This algorithm is similar in the spirit to the *kd-restart* algorithm presented in [HSHH07] and developed for kd-tree traversal. This allows us to recover each successively needed node (containing a brick or a constant value) along a ray by doing a top-down traversal (descent) in the tree. It is particularly efficient mainly because it is simple, requires few hardware registers [1], and highly benefits from the memory organization of the octree structure for cache coherency. This scheme is illustrated in Figure 6.5.

[1]The number of hardware registers used by a *shader* or a CUDA kernel is a very important factor of performance on current generation GPUs [Kir09]

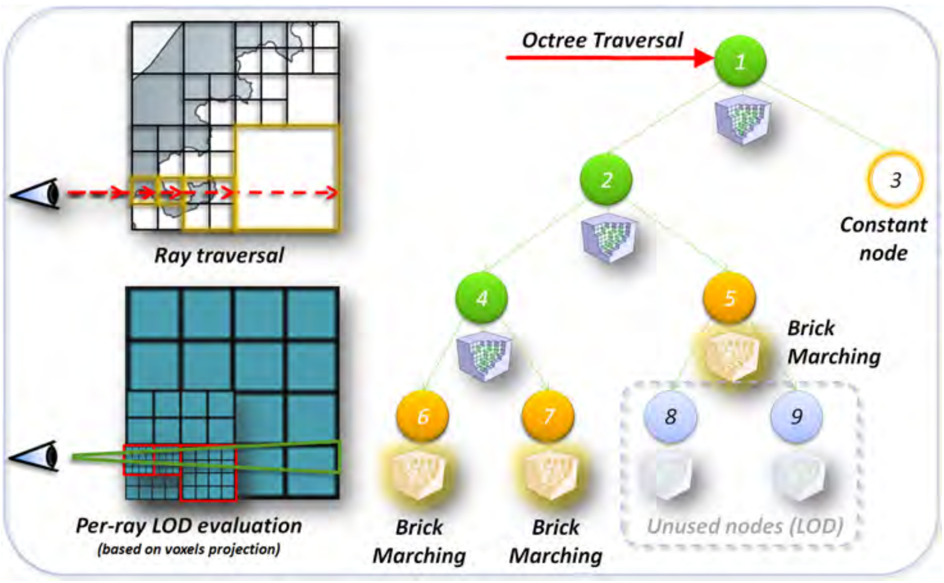


Figure 6.5. Illustration of the structure traversal with bricks marching and LOD computations based on projected voxel size.

For each descent, a ray starts from the root node and then proceeds downwards as described in Section 6.1.4. The descent stops at a node whose resolution is sufficient enough for the current view, based on the LOD criteria. Such a node either represents a constant region of space, or contains a brick (cf. Chapter 5). For a constant node, the distance between entry and exit points in the node is used to analytically integrate the constant material color along the ray, and skip the constant region with a large step. For a brick, we rely on standard ray-marching into a regular grid (cf. Section 2.3.2) to accumulate samples as described in Section 6.1.5.

One important observation is that our traversal does not need the structure to indicate correct level-of-details (while it is necessary in other methods, for instance in [GMAG08]). The needed voxel resolution is determined during the traversal based on the cone-tracing model described in Section 4.2. As we will see later, this is a key feature to minimize update operations.

6.1.3 GPU scheduling and traversal efficiency

Our rendering algorithm is split into two main tasks: octree top-down traversal and brick sampling. These two tasks are performed in one unique pass using a big kernel (or one fragment shader) alternating between both tasks. Generally, using such a big kernel on the GPU results in high register usage that provides less occupancy of the computation resources and so less efficiency. But in our case, this approach proved more efficient than making these two steps separate passes.

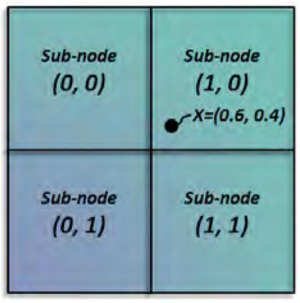
Indeed, in order to improve branching coherency and decrease register usage of the shader (a critical parameter on current GPUs), we tried to split this big shader into two smaller shaders. this leads to a tree traversal step interleaved with a brick marching, communicating through the video memory. This multi-pass approach proved less efficient than the unique kernel approach on SM4 GPUs (cf. Section 1.3). This is due to the penalty of having to write and read local ray data into the video memory. Indeed, such a memory access consumes hundreds of cycles [WPSAM10] of latency, and keeping all these data in register finally appeared more efficient.

6.1.4 Descending in the Octree

A descent in the octree structure has do be done for the first sample taken along a ray, and then each time a sample position ends up outside the current node. This descent is fast because, following [LHN05a], a point's coordinates in texture space can be used directly to locate each successive sub-node that needs to be followed in order to get to the node containing the point. This scheme is illustrated in Figure 6.6.

Descent algorithm

Let's explain the details for selecting the sub-node of a given node, at each step of the descent of the octree. As we have seen in Section 5.2, we call a *node-tile* the $2 \times 2 \times 2$ children of a given node that are stored contiguously in the node pool. A node-tile is pointed by a unique address stored in the parent node. Thus, a 3D offset inside a node-tile as well as its address is enough to select a child.



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

Figure 6.6. 2D localization into a node tile to find the index of the sub-node where x lies.

Let $\mathbf{x} \in [0, 1]^3$ be a point's local coordinates in a node's bounding box and c be the pointer to its children (node-tile). Thus, the 3D offset in the sub-tile of the child that contains $\mathbf{x}$ can be simply obtained by $\mathbf{off} = int(2\mathbf{x})$, the integer part of $2\mathbf{x}$ on each axis. E.g., in 1D for one axis with $x_{axis} \in [0, 1]$, there are two possible offset values for the children, namely 0 ($x_{axis} < 0.5$) and 1 ($x_{axis} >= 0.5$). Since the node pool in which we store the nodes is allocated in a linear memory, the 3D offset $\mathbf{off}$ needs to be transformed into a linear offset $off = \mathbf{off}_x + 2\mathbf{off}_y + 4\mathbf{off}_z$. To descend into the sub-node containing x, we can thus use the pointer $c' = c + off$. We then update $\mathbf{x}$ to $2\mathbf{x} - \mathbf{off}$ and continue the descent[2].

Level of detail

We will now detail how we chose the LOD (Level-Of-Detail) of the node to stop at when descending the octree. In order to simplify computations, we approximate voxels as spherical elements and screen pixels as disks. As explained previously, for each successive position sampled along a ray, we want to get a voxel whose size will correctly approximate the section of the cone at the current position (cf. chapter 4). The aperture of this cone is defined by the field-of-view of the current camera, and all along a ray its diameter projects to the size of pixel on the screen.

Thus, we want the size of each successively sampled voxel to project on the size of a pixel on the screen. The size of a voxel is determined by the depth in the octree (LOD) of the brick it is stored in, as well as the global resolution M of the bricks. When descending the octree to find a node, we stop at the first level containing voxels whose projected size is smaller than a pixel on the screen. More precisely, at each step of the 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 global brick resolution. We estimate its projection size

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

from the viewpoint of the near plane by $V_{proj} := V\frac{n}{d}$, 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 $V_{proj} < P$, the descent can be stopped.

As detailed in Section 4.3.1, continuous voxel sizes are reconstructed by linear interpolation between the two closest discreet levels of resolution. Thus, in addition to the brick located in the currently selected node (that corresponds to the immediately higher level of resolution), the brick located inside its parent node must also be provided. To make this brick available for marching (cf. next Section), we have to keep track of the parent node of the current node at each step of the descent.

6.1.5 Brick marching

During the ray-traversal, for each brick node crossed by a ray, we evaluate the volume rendering equation by sampling along the ray from the 3D texture part corresponding to the brick. More details on this kind of ray-marching can be found in [EKE01, Sch05, EHK+06].

This marching scheme inside bricks implements our discrete front to back composition scheme based on pre-integrated geometry and material properties described in Section 4.2.2. During this marching, samples are taken inside the two bricks obtained during the descent of the octree. They are interpolated linearly in order to reconstruct a quadrilinear interpolation in order to provide continuous voxel footprints. As we have seen in chapter 5, bricks are stored in a *brick pool* located in texture memory. Thus, for each sample taken in a brick, we rely on the fast and highly optimized texture sampling hardware to do the trilinear interpolation of the data stored per voxel. We show in Figure 6.8 a comparison of rendering quality between renderings with and without trilinear interpolation.

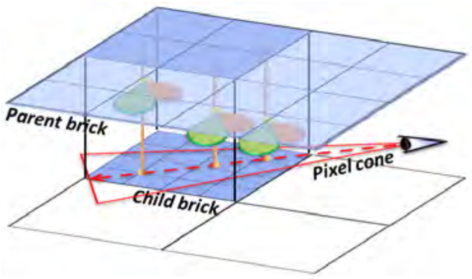


Figure 6.7. Illustration of the brick marching process for one ray with quadrilinearly interpolated samples.

The brick marching step stops once the coordinates of the next sample ends-up out of the current node, or a maximum opacity is reached for the accumulated color. Once the ray leaves the brick, we use the new ray position as the origin for the next top-down descent in the octree. Brick marching must also stop when the LOD of the next sample falls outside of the range of LODs provided by the two current bricks. In this case, a new top-down descent must also be performed in order to get lower resolution bricks.

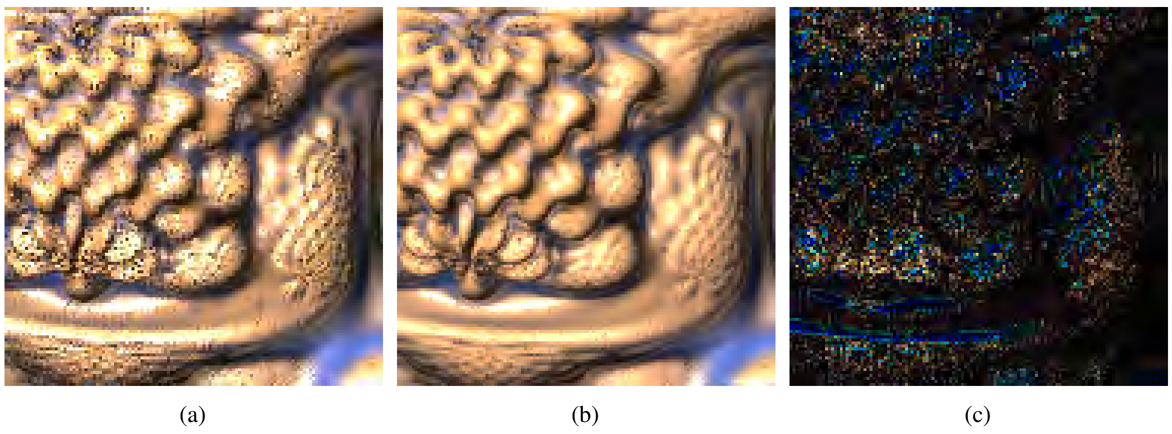


Figure 6.8. Comparison of rendering quality without (a) and with (b) trilinear interpolation of the samples taken inside the bricks. Figure (c) display the difference between the two images with a 2× amplification.

Stepping distance and adaptive sampling

Following the Nyquist-Shannon sampling theorem [Nyq28], at least one sample must be taken for each peak and another for each valley of the original signal. At the highest frequency that can be represented inside a regular voxel grid, each voxel can either represent a peak or a valley of the signal we sample. Thus, the stepping distance between two successive samples along a ray must be chosen to correspond at least with the size of a voxel in the brick, in order to ensure a correct reconstruction of the original signal with minimal sampling.

As we have seen in Section 4.6, both an isotropic and an anisotropic voxel scheme can be used. The stepping distance used between two successive samples taken along a ray (and so in successive bricks) depends on this scheme:

- When relying on the anisotropic voxel scheme (Sec. 4.6.4), a directional pre-integration of the material and geometry parameters is provided per voxel. This pre-integration has been computed specifically for a length of integration l corresponding to the size of the voxel it is stored in. Thus, the distance between two samples taken in a brick must respect this pre-integration distance to ensure a correct reconstruction. Since we interpolate linearly between two samples to reconstruct the required voxel size, we use this interpolated voxel size as the stepping distance taken between two successive samples. This allows us to use the minimum amount of sampling along a ray, reducing the number of texture access and thus providing high performance, while maintaining high rendering precision.
- When relying on the isotropic voxel scheme (Sec. 4.6.3), both the material color, normal distribution and opacity are simply averaged per-voxel. Thus, in-depth opacity and occlusions are not correctly handled in the pre-integration. As we have seen in Section 4.6.3, this imprecision can be compensated during the marching process by using a smaller stepping distance than the voxel size. In this case, the stored opacity must be corrected to correspond to the actual length of integration (stepping distance). In practice, we usually use a stepping distance $d = \frac{1}{3}V$, with V the interpolated voxel size. This leads to more texture access, but better rendering quality.

Both schemes compute the inter-sample stepping distance as a factor of the voxel size. Since we make this voxel size always projecting to the size of a pixel on the screen, this provides us with an adaptive sampling scheme, reducing the sampling frequency when going away from the camera. As we have seen in Chapter 5, the octree subdivision is also adapted to the frequency of the dataset. Thus, since our sampling scheme follows the stored voxel resolution, it also adapts to low-resolution regions of the dataset.

Brick marching optimizations

Depending on the data stored in the voxel representation, various optimizations can be done to enhance the efficiency of the ray-marching. An important performance penalty appears when the structure is configured with a large brick resolution M. In this case, large empty regions can be marched in the bricks, generating costly texture accessed for data that will not contribute to the final color.

To optimize this case, we rely on the higher bricks available for the MIP-mapping mechanism (quadrilinear interpolation) to detect fully transparent regions and proceed with larger steps in this case. To allow this, the lower resolution brick is always sampled first when fetching a value. If this value appears to be fully transparent, one can know that a step at least the size of a voxel in this lower resolution brick can be taken safely.

In some applications, one can rely on another optimization. In addition to the geometry and material properties, an information on the distance to the closest non-empty voxel can be stored per voxel in constant or empty regions. This distance can be used similarly to proximity clouds [CS94] in order to skip large empty spaces in the volume during marching inside a brick. Some applications like the procedural noise amplification presented in Chapter 8 require such a distance to an isosurface to be stored. In this case, using it for such optimization is a welcome side effect.

6.1.6 Empty or constant space skipping

As we have seen, when a constant node is traversed by a ray, its constant value is analytically integrated on the distance traversed inside the node, and the node is skipped directly to proceed with the next descent in the octree to find the next traversed node. However, care must be taken when such skipping is done as illustrated in Figure 6.9. Indeed, while a given node can contain a constant value, its parent node necessarily contains a brick (if it did not, the children nodes would not have been created). This brick in the parent node necessarily contains the constant value of the child node in all voxels covering this sub-node. However, due to the linear interpolation, a value sampled in the parent brick in this area does not necessarily correspond to this constant value, but can be interpolated with another voxel located outside the constant area (as illustrated in Figure 6.9). This situation can happen in a limited area the size of half a voxel in the parent brick, at the boundary of an empty node. Thus when skipping an empty node, this thin border must not be skipped. Instead it must be sampled with the values interpolated between the constant node value and the values read in the parent brick.

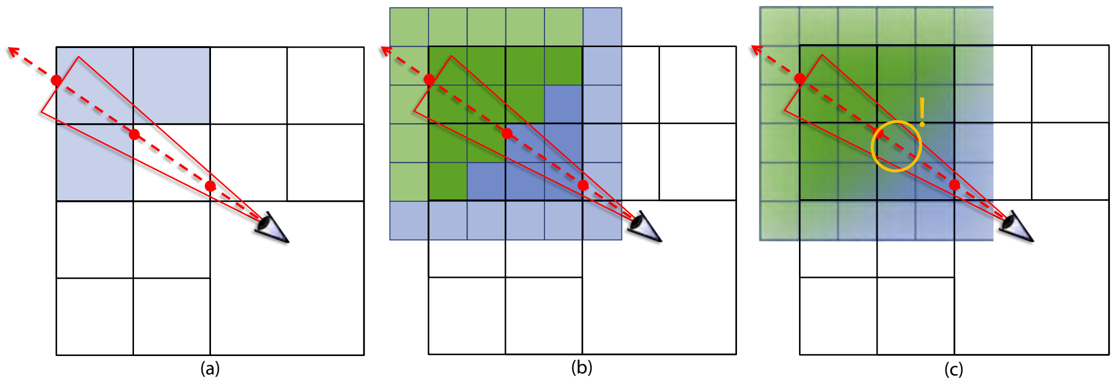


Figure 6.9. Care must be taken with empty space skipping when MIP-mapping is used. White nodes are empty while blue nodes contain bricks. When a ray skips a node because it contains a constant value (a), artifacts can arise from the loss of information that would have been used in the upper node's brick if the empty region had been sampled (b, c).

Finally, when skipping an empty node, care must also be taken to ensure that the large step taken still keeps the required LOD level (to approximate the cone footprint) inside the range available in the constant node. If not, values that should have been sampled in higher bricks will be missed.

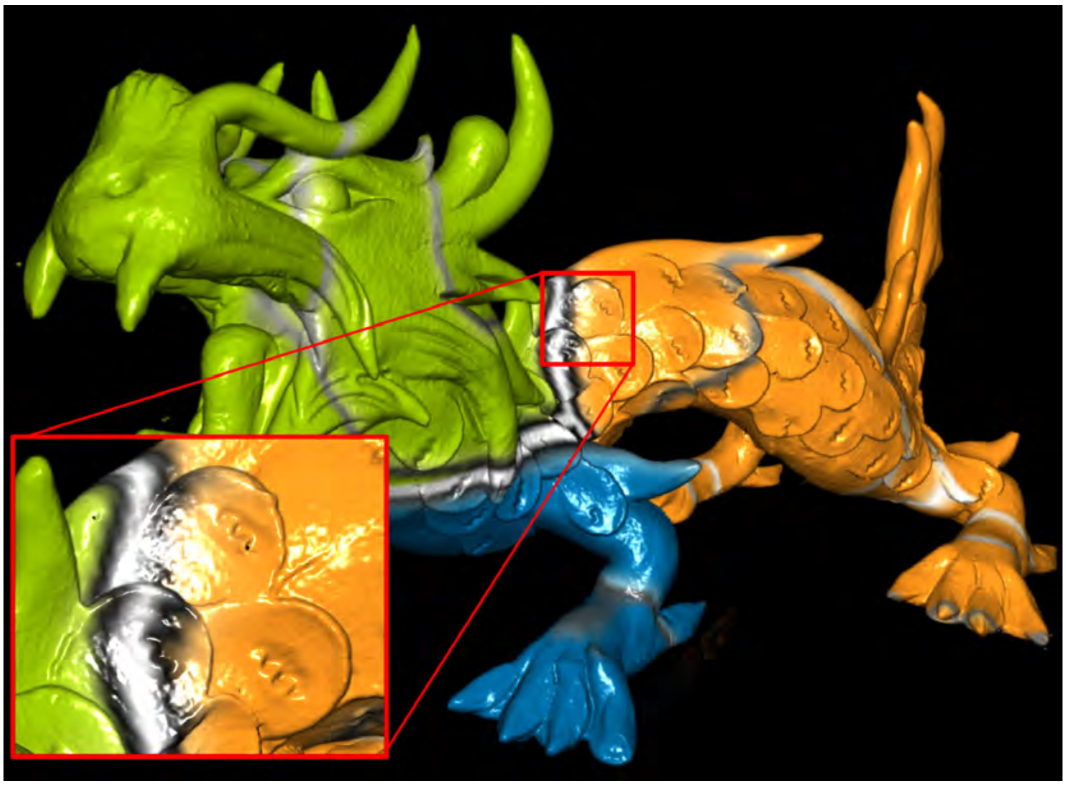


Figure 6.10. The XYZRGB-dragon voxelized at 2048^3 voxels into our multiresolution sparse voxel octree structure and rendered at 60FPS on a GTX280 with our approach using voxel cone tracing.

6.2 Integration with traditional CG scenes

Our rendering approach can be transparently integrated into traditional CG scenes rendered using rasterization through compositing, as illustrated in Figure 6.11. To do so, our renderer takes the current color buffer and depth buffer as input, and provides them updated with new voxel objects rendered with our approach.

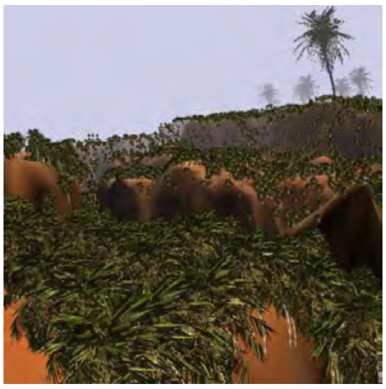


Figure 6.11. Scene mixing a rasterized terrain with voxel trees rendered with GigaVoxels.

All opaque rasterized objects of the scene must be rendered before voxel rendering. This allows us to optimize voxel rendering by adjusting the entry point and exit point of each ray based on the depth present in the depth buffer, and thus allows skipping occluded parts. Semi-transparent objects must be split between objects in front of voxel objects, that are rendered before, and objects behind voxel objects, that are rendered after (with a front-to-back compositing scheme). Once again, such scheme allows us to optimize voxel rendering by initializing the accumulated color and alpha of each ray with the input RGBA color of the pixel (provided in the color buffer), and thus to stop quickly rays that saturate opacity.

6.3 Performance analysis

6.3.1 Performance comparison with rasterization

In order to evaluate the performance of our rendering algorithm, we compared it with the performance of the GPU rasterization pipeline on two complex scenes: the 12M triangles SanMiguel scene presented in Figure 6.12, and the 13.5M triangles Coral scene presented in Figure 6.12. These experiments were done on an NVIDIA GTX480. Both scenes were pre-voxelized into a 9 level octree, and we compared average frame rates depending on the distance of observation. A 16× MSAA (multi-sampling) was used on the SanMiguel scene and a 32× MSAA on the Coral scene in order to ensure a similar rendering quality between both approaches (as can be verified on the accompanying screen shots). We compared performances on 5 increasing viewing distances, with the average number of triangles projected per pixel ranging from 50 to 1000. We observe that in such situation, the performance of our approach is between 3.2 and 41 times faster than the rasterization approach. In addition, while the performance of the rasterization approach decreases with the increase of the number of triangles per pixel, our approach tends to be faster as much as the object covers a smaller area on the screen. More analysis would be required in order to evaluate precisely the number of triangles per pixel starting from which our approach is faster. We estimate it to be around 20-30.

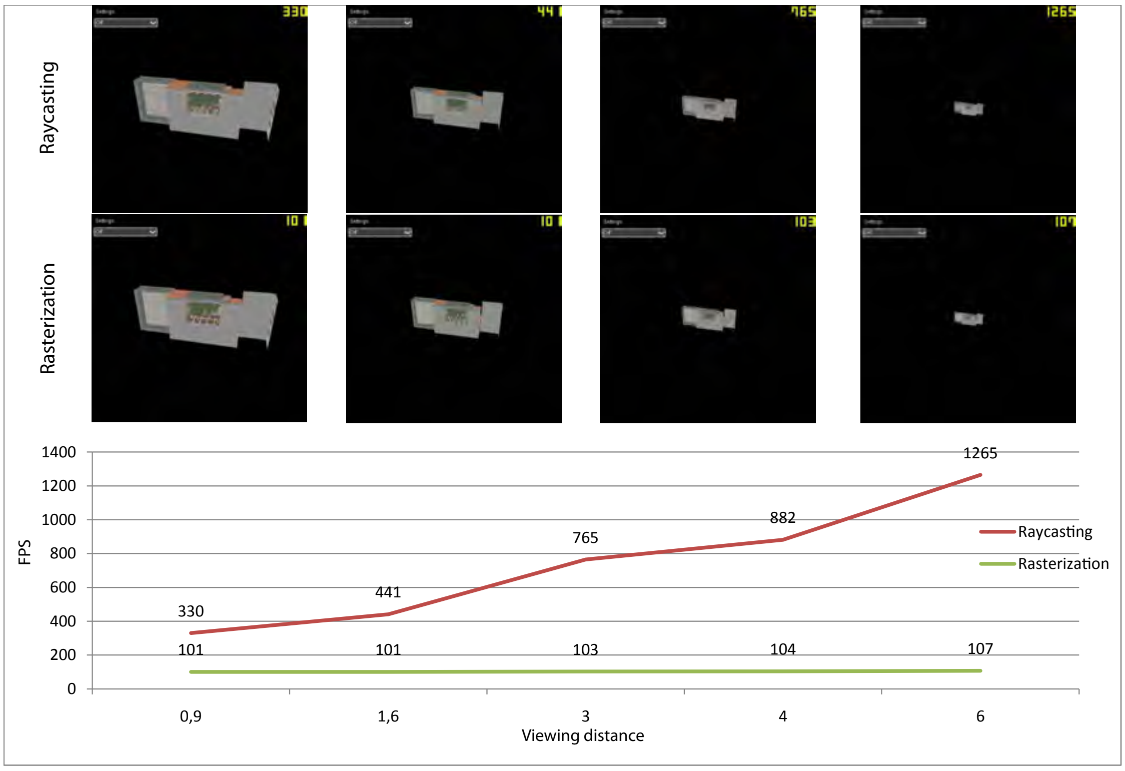


Figure 6.12. Comparison of rendering speed (FPS) between our sparse voxel octree raycasting rendering and standard rasterization for the same rendering quality on the San Miguel scene. 512^2 framebuffer. Rasterization: 12M triangles, 16X MSAA. Raycasting: 9 levels octree, 3^3 voxels bricks. NVIDIA GTX480.

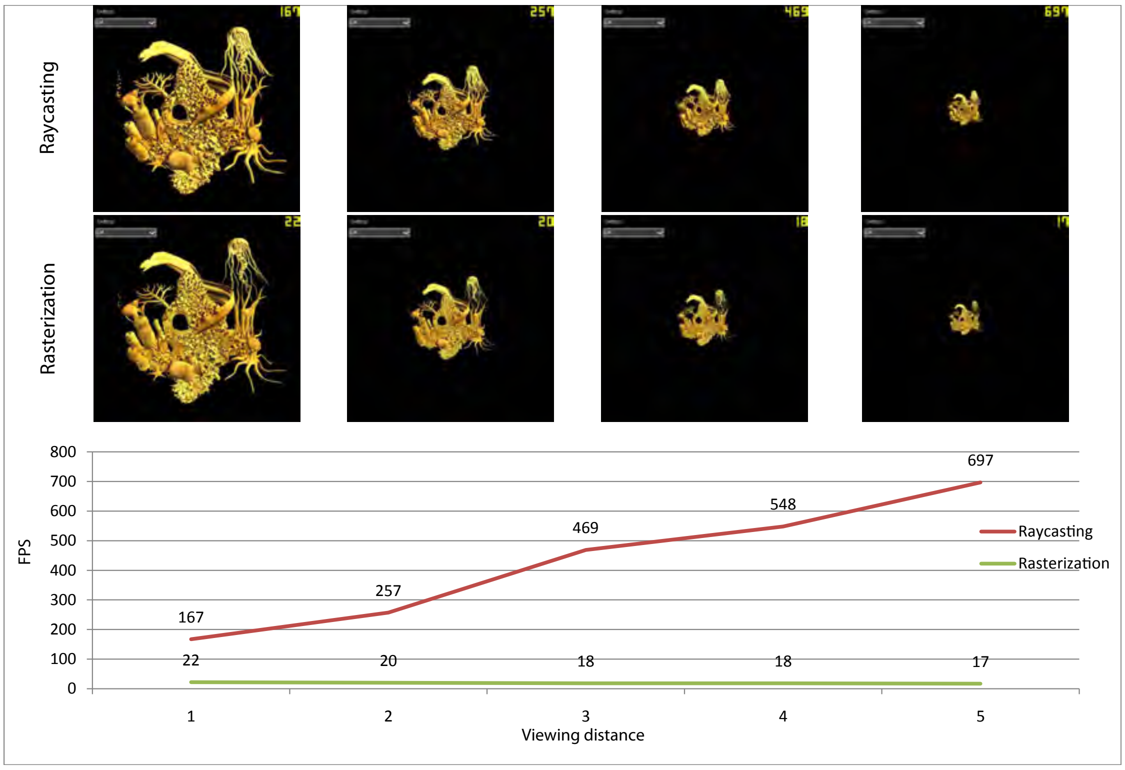


Figure 6.13. Comparison of rendering speed (FPS) between our sparse voxel octree raycasting rendering and standard rasterization for the same rendering quality on the Coral scene. 512^2 framebuffer. Rasterization: 13.5M triangles, 32X MSAA. Raycasting: 9 levels octree, 3^3 voxels bricks. NVIDIA GTX480.

6.3.2 Rendering costs and silhouettes

We analyzed rendering costs on a typical scene depending on the brick resolution M and the location of the rays on the screen. These experiments were done on an NVIDIA GTX480. Figure 6.14 shows in grayscale the time taken by our ray-casting procedure for each pixel of the screen. We observe that rays traversing the silhouette of the object tend to be more costly than other rays. This is due to the fact that a high octree subdivision is encountered in this area, making rays traversing many nodes without being stopped early by opaque materials. We compared the cost of silhouettes and interior rays on a typical scene (a Mandelbulb [Whi09]), a 3D Mandelbrot fractal, cf. Figure 6.15(c)) with no limit on the depth of the octree. Figure 6.15(b) shows the average time (in nano seconds) per ray passing through the silhouette and interior rays, depending on the brick resolution M used in the structure. Figure 6.15(a) shows the global rendering performance (in *Frames Per Seconds*, FPS) of the same scene depending on the brick resolution. All measures have been made with a Phong shading computation.

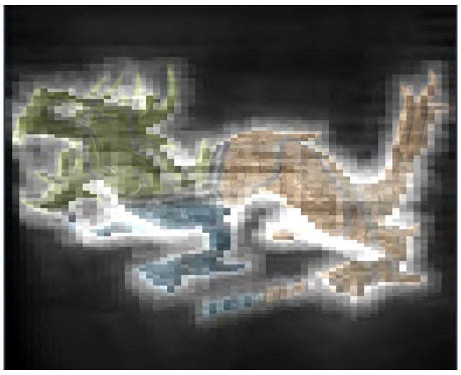
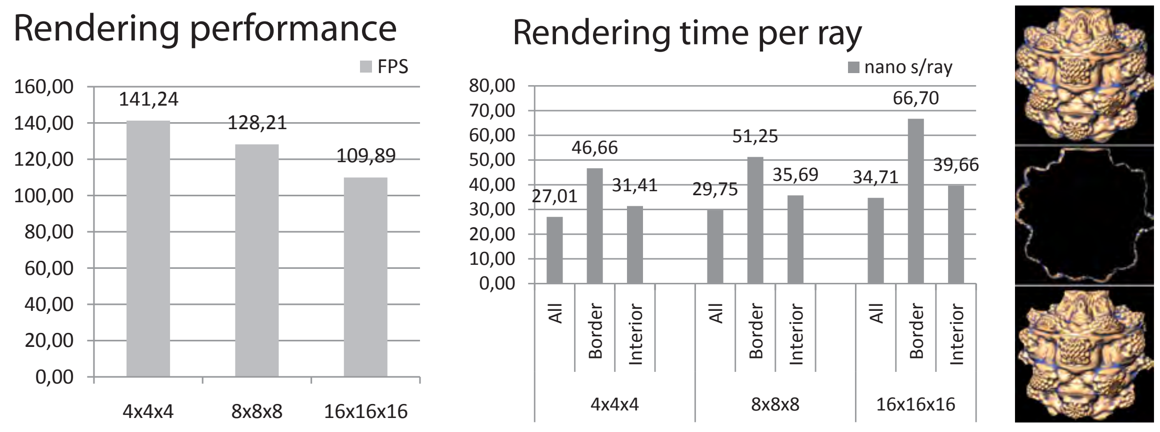


Figure 6.14. Display of ray execution times in grayscales for a 1024^3 voxels scene and 16^3 bricks. Time is measured per tile corresponding to the SIMD execution entities (warps) of the GPU.

The first observation is that the smallest brick resolution M provides the highest rendering performance (141FPS for 4^3 bricks). An increase of the brick resolution M leads to a decrease of rendering performance, and an increase of the average time per ray, whatever their location on the screen. As we have seen in Section 5.3, small bricks (but still $\geq 4^3$) also tend to provide the most compact storage.

The second observation is that silhouette rays tend to be on average 1.5× more costly than interior rays. In addition, the increase of the brick resolution tends to make silhouette rays even more costly, which is logical given that silhouette rays have to traverse more transparent or semi-transparent voxels in large bricks.



(a) (b) (c)

Figure 6.15. Average rendering performance on a GTX480 of our approach on the Mandelbulb scene depending on the brick resolution (a). Comparison of the rendering cost per ray (in ns) between rays traversing the silhouette (called *"border"*) and rays traversing the interior of the object, depending on the brick resolution (b). Figure (c) shows the full data set (top), parts considered for silhouettes (middle) and parts considered for interior (bottom).

We also evaluated the relative cost per-ray between the traversal of the octree structure itself (Sec. 6.1.4) and the marching of the bricks (Sec. 6.1.5), depending on the brick resolution M. Results for the Mandelbulb scene are presented in Figure 6.16. For small bricks, the rendering cost is largely dominated by the traversal of the octree, while as expected, when increasing the brick resolution, total time tends to be dominated by the marching of the bricks.

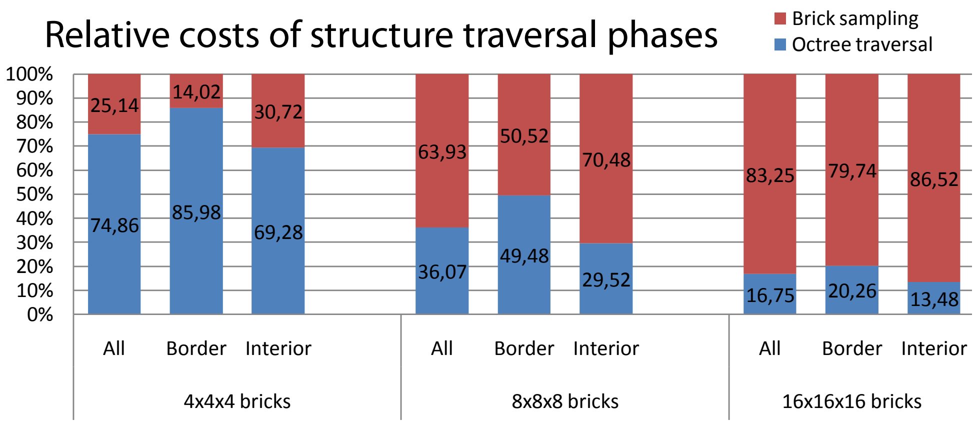


Figure 6.16. Average relative costs of the octree traversal and marching of the bricks rendering steps, for rays located on the silhouette, on the interior and on the whole image, depending on the resolution of the bricks. Tested on a GTX 480.

Chapter 7

Out-of-core data management

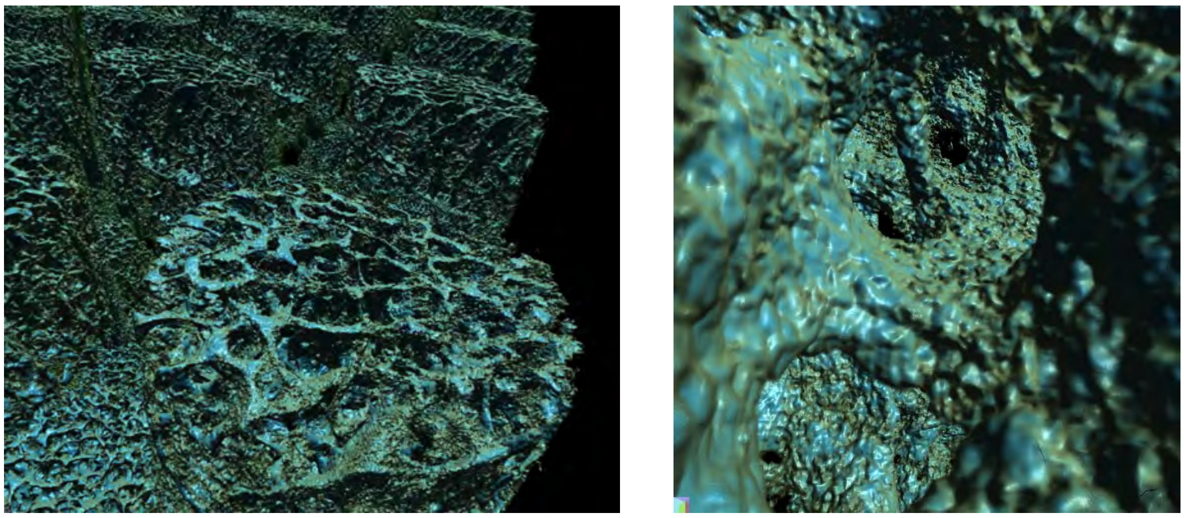


Figure 7.1. Examples of a very large 8192^3-voxel scene composed of medical data and rendered at 20-40FPS using GigaVoxels on an NVIDIA GTX 280 GPU.

In previous chapters, we described our GPU based data structure and we have seen how to use it efficiently for rendering. We will now investigate how to deal efficiently with arbitrarily large amounts of voxel data, given a limited budget of video memory. To do so, we developed a new out-of-core data-management scheme based on an *application controlled demand paging* mechanism, that allows us to virtualize our voxel data structure. This allows for virtually infinite volume resolution and provides a full scalability of the amount of voxel data that can be manipulated. It is built around three main concepts: a ray-based visibility detection, a fast GPU cache mechanism, and a data parallel building and update of the data structure based on a GPU data producer.

Parts of the structure are incrementally loaded on-the-fly in video memory based on data request emitted directly during rendering. These parts are maintained in video memory using an efficient GPU-based caching scheme. This cache is in charge of maintaining the most recently used elements in video memory, while providing room for new elements by recycling the oldest ones. It is applied on both the octree structure, and the bricks (cf. Section 5). Contrary to previous approaches, this allows us to decouple the storage from the actual rendered data resolution. This results in a fully scalable solution that can deal with extremely large volumes. This cache was designed as a generic GPU cache mechanism that can be used in a wide range of ray-tracing applications, beyond our voxel-based rendering pipeline. In this thesis, we mainly target real-time applications, but our system also allows off-line usage. Indeed, even for off-line applications, rendering large datasets can be problematic, and off-line rendering engines increasingly rely on GPU acceleration to speed-up computations [ZHR+09].

7.1 Overview and contributions

We address the problem of dealing with very large amounts of voxel data, while the available amount of video memory is limited, as well as the bandwidth to access it from a larger mass storage. The amount of memory embedded on the video card is usually many times smaller than the amount of system memory (cf. Section 1.3). In addition, there are many situations in practice where only a small subset of this video memory is available for voxel rendering. This is the case for instance in a video game environment, where much of the video memory is already used by other parts of the game. This limitation makes it necessary to transfer data from the, much larger, system memory (or even the hard disk), or to generate them on-the-fly inside video memory (in case of procedural data or data generated from another representation). Both transfers between system memory and video memory (cf. Section 1.3) and procedural generation of data are slow. Thus, it is necessary to carefully control memory updates. Our goal is to provide a scheme that is able to scale to very large and detailed scenes. In this context, the whole scene and all its scales can not be permanently kept inside the limited amount of available video memory, even compacted inside the multiresolution spatial subdivision structure presented in Chapter 5. One insight is that, for a given point of view, the entire volume does not need to be in memory for rendering. Only visible voxels at the right scale need to be present, out-of-frustum as well as occluded data do not need to be loaded.

In such a context, we propose an out-of-core scheme to manage very large datasets that do not fit entirely into the limited amount of video memory available. As we have seen in Chapter 6, our multiresolution ray-casting scheme ensures that only the subset of visible voxels, at the exact necessary resolution (to project one voxel on one pixel of the screen), will be used for rendering. Thus, for a given frame, only a few voxels per-pixel of the screen are required to be present in video memory, which represents only a very small subset of the whole dataset. In addition, during rendering, when the point of view changes, only a small amount of new data is needed (e.g, because it was previously occluded, out-of-frustum, or at the wrong resolution), while most information can be reused. We propose a new data management strategy to maintain in video memory only subparts of the voxel structure needed when exploring a region of a large scene at a given scale, plus additional parts kept in memory for being reused. This strategy is illustrated in Figure 7.2. The maximum depth that can be explored is only limited by the number of nodes that can be kept loaded in order to connect to the root of the octree (Fig. 7.2).

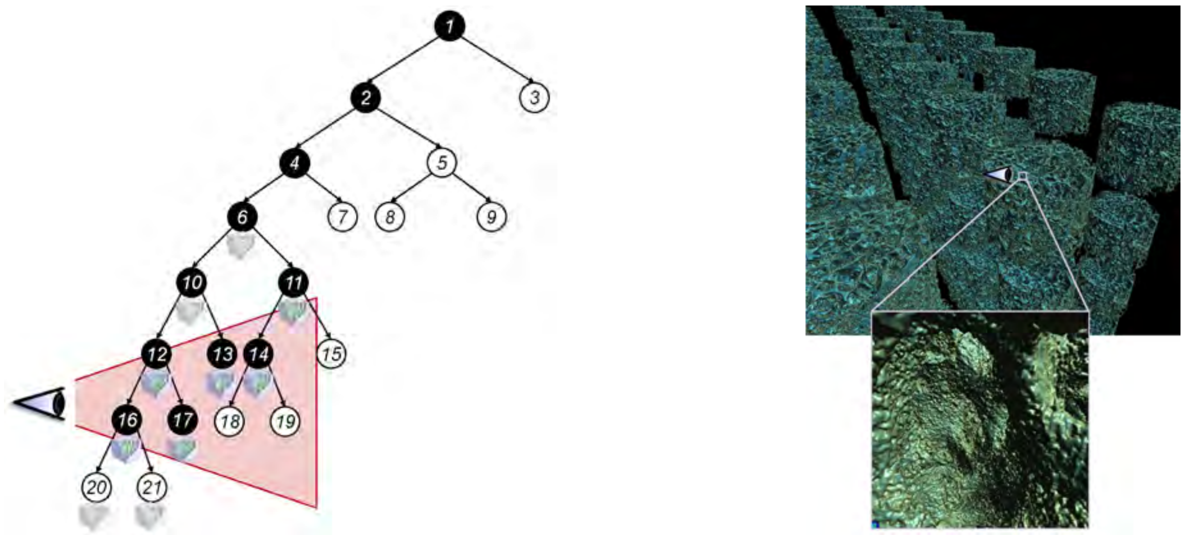


Figure 7.2. Illustration of the state of our voxel structure stored in GPU cache during the exploration of a very detailed scene. Current view frustum position is displayed in red. Displayed nodes and bricks are the ones actually stored in the caches. Black filled nodes and colored bricks are marked as used for the current frustum configuration. Others are not marked as used but kept in the cache for later usage.

7.1.1 Application controlled demand paging on the GPU

In order to allow only a subpart of our data structure presented in Section 5 to be physically present in video memory, we propose to rely on some sort of virtual memory (cf. Section 2.5.1). Virtual memory has been used for a long time inside operating systems, with *demand paging* (cf. Section 2.5.1) of data from the hard disk in order to provide a much larger address space than the available amount of system memory. Current generation GPUs do not expose such a mechanism to allow the manipulation and the rendering of datasets (generic data or textures) larger than the total amount of video memory.

In order to bring a virtual memory mechanism on the GPU, we propose an *application controlled demand paging* scheme (cf. Section 2.5.1) entirely controlled and managed on the GPU. It is implemented through a new GPU cache mechanism (detailed in Section 7.3) that allows us to keep only a subset of our whole data structure inside video memory, and to exploit the temporal coherence during the exploration of a scene by maximizing the reuse (from frame-to-frame) of data already loaded in video memory. Data requests on this cache (*cache miss*) are emitted directly by our ray-tracing algorithm that ensures a parsimonious traversal, and only visits the strictly required subset of the whole dataset. Data requests are forwarded by the cache mechanism to a GPU producer (detailed in Section 7.4) that is in charge of loading data inside the cache in video memory. Note that this demand-driven mechanism is independent of the usage: data visiting can be caused by primary rendering rays, but also secondary shadow rays, reflections or indirect lighting computations.

Figure 7.3 illustrates how our new GPU cache is inserted in the memory hierarchy (of current PCs, cf. Section 1.3) providing the data necessary for rendering the shader cores in charge of the ray-tracing computations.

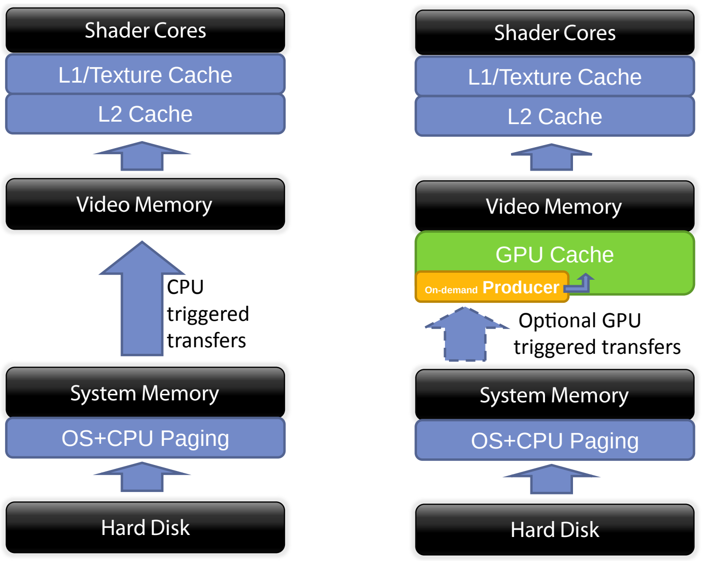


Figure 7.3. *Left:* Classical cache hierarchy used during GPU based rendering. No caching is available between the system memory and the video memory. *Right:* Our new GPU cache that allows on-demand paging of data inside video memory. Data is either loaded from system memory or generated on-the fly by a GPU producer.

Quick recall of the previous work

As detailed in Section 2.5.2, texture streaming approaches have been proposed for the GPU rendering of triangle meshes in order to solve the problem of out of core rendering [TMJ98, CE98, LDN04, GM05, LKS$^+$06]. The problem with these previous approaches is that they involve complex multi pass rendering schemes, with costly visibility determination in order to track usage of visible texture parts and detect missing data. The management of the replacement policy (that controls the strategy of eviction inside a cache, cf. Section 2.5.1) was done serially on the CPU, forcing the transfer of visibility information to the system memory. Due to the low bandwidth between video memory and system memory (Sec. 1.3) and the costly visibility determination, these schemes were restricted to a low number of managed elements, and thus to a very coarse-grained granularity, in order to allow interactive processing. In addition, precise LRU (Least Recently Used, cf. Section 2.5.1) replacement policy was usually too costly to implement. It requires tracking of all the elements used by each rendering pass, as well as a sorting operation based on the usage of each element. Thus, most of the previous schemes implement only a simple FIFO (First In, First Out) replacement policy, that does not ensure the most efficient reuse of data present in the cache. Goss et al. [GY98] proposed to do direct tracking of data used for rasterization in order to prevent using a complex visibility detection algorithm, but this approach required a deep modification of the texturing hardware. In addition, it still required a software (CPU) scanning and management (sorting) of the list of used elements in order to trigger loading. Such sequential streaming and data caching schemes for video memory buffers and implemented on the CPU do not scale to very large datasets, with real-time fine-grained management. The approach of Gobbetti et al. [GMAG08] developed in parallel to our work also suffers from the same problems.

The need for application control over paging

Demand paging for real-time rendering applications on the GPU is still a wide open problem. However, a fully automatic virtual memory and generic paging scheme on the GPU for the video memory, on the model of what is done on the CPU, would not fit the needs of high performance graphics applications. Indeed, in the context of software (CPU-based) rendering (cf. Section 2.5.3), multiple authors [CE97, WSBW01, DPH+03, WDS05] have shown that simple reliance on the virtual memory and generic automatic paging mechanism of the operating system leads to egregious performance. Thus, they have proposed specific application-controlled demand paging approaches in order to by-pass the inefficiencies of the automatic paging mechanism of the operating system. Especially, they have shown that application control over load/store operations (page-in/page-out), page size and address translation (to allow 2D and 3D storage) are essential to provide good performance and to adapt to the specificities of rendering applications.

In addition, the particularities of the data-parallel processing model of the GPU must also be taken into account in the design of a cache that will serve algorithms running on such architecture. Indeed, on the CPU, algorithms are usually tightly synchronized with the availability of the data, because only a few threads are running in parallel. On the GPU, there are thousands of threads running the same code in parallel, but there is no parallel multitasking and we do not want individual threads to proceed to the loading of their own data. This loading would not be efficient since it has to be done in parallel in order to fully exploit the hardware. Overall, we do not want individual rendering threads to be synchronised with the data they need. Thus, as we will see, we have to manually manage this multitasking between rendering tasks and loading tasks in order to provide an efficient data-parallel paging scheme. This multitasking totally desynchronizes rendering threads from the availability of the data they need, by collecting data requests in order to answer them all together in parallel, in a special phase separated from the rendering.

7.1.2 Details on our approach

As we have seen, no automatic paging mechanism is provided by today's GPUs, and anyway an application-specific approach is essential to provide high performance paging for rendering applications. Thus, we propose a very efficient *"software"* implementation of such a scheme on today's GPUs, dedicated to the real-time rendering of very large voxel datasets. This implementation is based on a new generic high performance GPU cache (Sec. 7.3) implementing a fine-grained LRU (Least Recently Used) page replacement policy. We make possible the tracking of fine-grained usage and data requests thanks to a data parallel implementation of the LRU replacement policy on the GPU. This cache is used to virtualize the access to the data structure in order to keep only a subpart of the entire dataset inside video memory at a given time, while allowing on-demand access to the whole dataset.

The most important characteristic of our approach compared to previous methods is that it places the control of the whole paging and caching scheme on the GPU, removing the need for costly communications and synchronization between the CPU and the GPU that were previously needed to manage the cache. It requires minimal interaction with the CPU, freeing it entirely for other tasks. It also removes the need to maintain a mirrored version in system memory of the GPU data structure. This was previously required in order to track element usage and to steer data structure updates [GM05, GMAG08]. This involved many supplementary CPU processes and the cumbersome maintenance of two data structures. This also involved a lot of small memory transfers from the CPU to the GPU memory resulting in a very low efficiency of the updates (cf. our comparison in Section 7.5.3).

Generic high-performance GPU cache

Our application controlled demand paging scheme is built around a new generic cache mechanism entirely implemented on the GPU. This cache mechanism has been designed to handle the memory management of any data buffer located in video memory. It basically relies on an LRU (Least Recently Used, cf. Section 2.5.1) recycling scheme in order to allocate slots for new elements (*pages*), by recycling slots occupied by those elements that have not been used for a long time. This generic cache is entirely implemented on the GPU as data-parallel processes and its implementation is detailed in Section 7.3.

As we will see in Section 7.3, our generic cache manipulates batches of requests (*"cache misses"*) and usage information generated directly by our ray-tracing (Sec. 6). Batches of requests are forwarded by the cache to a *GPU Provider* interface implemented by a *GPU Producer* mechanism (presented in Section 7.4) that can implement multiple Provider interfaces in order to fulfill data requests for multiple caches. The GPU Producer is in charge of actually loading new data inside the cache. Many different kinds of data producers can be implemented. They can either load requested data from the system memory, generate data dynamically or provide a mix of both, loading data and modifying them before writing them inside the cache. They are implemented on the GPU as data parallel tasks, dealing with batches of requests in parallel, and providing a parallel loading or generation of the voxel data and octree structure.

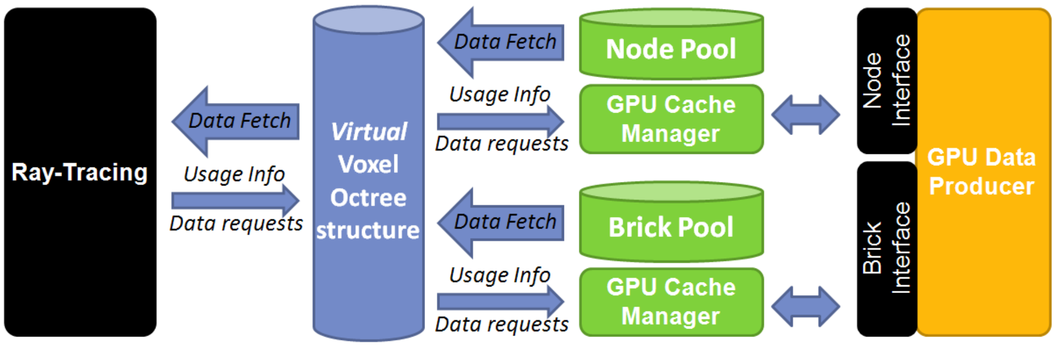


Figure 7.4. Illustration of the virtualization of our voxel data structure using two generic GPU cache mechanisms and a GPU Producer answering data requests and loading bricks and nodes inside the caches through two GPU Provider interfaces.

In practice, as illustrated in Figure 7.4, we use two instances of this cache. One to manage the *node pool* containing our octree data structure and one to manage the *brick pool* that contains its associated bricks (cf. Section 5.2.1). This leads to a unified management of the *brick pool* and the *node pool* as two LRU controlled caches. This allows us to virtualize our whole data structure, enabling deep zooms and exploration at arbitrary scales. In order to maintain these caches, rays used for rendering provide information about the nodes and bricks that were traversed during the ray-casting process, and ask for sub-nodes and for bricks when the ones they need are not present in the cache (cf. Section 7.1.2).

There are situations where large pre-computed datasets do not even fit inside the system memory. As we have seen, much previous researches focused on providing efficient paging schemes in system memory for data located on disk or accessed over a network. However, such scheme was not the target of our research, and we believe that today's real challenge lies more in the smart virtualization of the video memory. As we will see in Section 7.4, we still propose a loading producer able to load large datasets from disk.

Ray-based visibility and data requests

Our approach does not require a costly visibility determination as was the case for previous GPU rendering approaches in order to detect required missing data and to track the usage of data already loaded to maintain the LRU replacement policy. Other approaches like [GM05, GMAG08] rely on costly visibility determination passes interleaved with normal rendering passes. They use the CPU to issue visibility queries or to compute culling, and then to send drawing commands to the GPU. This requires a complex scheduling of visibility tests in order to be efficient. In addition, these methods also require a CPU traversal of a clone acceleration structure maintained in system memory. This clone structure is used to issue visibility queries, to load necessary data for a given point of view, and to manage the LRU mechanism. These approaches involve a lot of transfers and synchronizations between the CPU and the GPU, making them heavy and difficult to implement efficiently (in order to carefully overlap transfers and computations).

Our application-controlled demand paging mechanism provides us with what we call a ray-guided streaming approach. With this approach, visibility information and data requests are collected directly during the GPU traversal of the structure for rendering (cf. chapter 6), on a per-ray basis. The idea is to exploit the natural characteristics of ray-tracing within a structured space-subdivision structure[1] that ensures an ordered traversal. Indeed, traversing such a structure allows for visiting nodes in order of visibility along rays. Ray individuality combined with early ray termination (stopping when accumulated opacity gets saturated) during front-to-back traversal, ensures that only visible parts of the structure will be visited during rendering. In addition, as explained in Chapter 6, our rendering algorithm ensures the selection of the required level of detail on a per-ray basis, which makes it possible to request data to the producer at the exact necessary resolution. The result of this per-ray visibility scheme is illustrated in Figure 7.5.

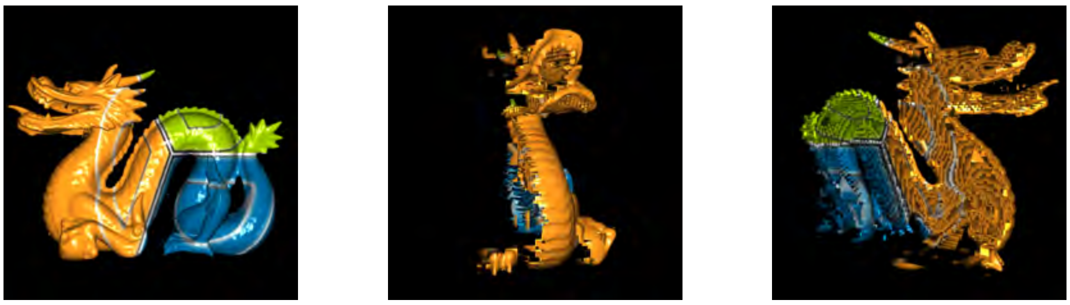


Figure 7.5. Demonstration of the result of the ray-based occlusion detection on a simple model. When the loading mechanism is disabled, one can see that occluded parts of the structure have not been loaded.

This approach leads to a very good solution to treat very complex multiscale scenes. In this way, it is possible only to trigger data load 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, as it was the case with previous methods. This also leads to minimal CPU intervention and no synchronization, since both LOD and visibility computations are performed per-ray on the GPU.

Another advantage of this ray-based visibility and data request scheme is that it allows totally arbitrary rendering schemes. Indeed, since each ray individually reports its own data usage and emits its data requests, rays can be launched in any direction, not necessarily in order to directly compute an image, as shown in our global illumination application presented in Chapter 9. This also makes it possible to support general ray-tracing schemes, with an arbitrary number of secondary rays as illustrated by our soft shadow application presented in Section 8.3. This new scheme provides a great flexibility compared to previous approaches (Sec. 2.5.3) that were final rendering centric and restricted to frustum-launched rays, with no support for secondary rays.

Structure updates and building

Since not only the bricks but also the octree structure are managed with our cache, GPU producers are actually in charge of providing data necessary to build the octree structure dynamically. It is important to note that in this context, because of the cache mechanism, updates of the octree structure are performed lazily and only when necessary, while other approaches need to update it entirely at

[1]Like an Octree, a Kd-tree or a regular grid, but not a BVH for instance whose nodes can overlap.

each frame [GMAG08]. The basic principle is to subdivide or fuse nodes in the octree in an LRU manner. When node tiles ar no longer used, they become more and more prone to be recycled inside the node cache. Thus, this also provides a progressive and lazy un-building of the structure during the exploration of a scene.

At a given point in time, the cached octree leaves do not necessarily correspond to the nodes used during the rendering (Sec. 6). 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, although currently useless, might need to be reused later. 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.

7.2 Multi-pass update scheme and progressive refinement

During the traversal of the hierarchical volume described in Section 6.1, 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 in Section 4.2) might be missing. Nodes might need to be subdivided, or nodes could be present at the right depth, but might miss the corresponding volume brick. In both cases, a ray will *ask* for missing data by issuing a *data request* to one of the two caches.

In classical CPU-based paging mechanisms, when a thread tries to access data that is not available in the cache, this thread gets suspended until the data have been loaded and are available in the cache. However, in the massively data-parallel environment of the GPU, we do not want threads to be automatically and individually suspended, and data loading to be processed immediately. Indeed, we want to process data requests all together and in parallel, in order to efficiently load or create required data. We want to maintain a coherency in the processed computations, in order to efficiently use the data-parallel architecture. Thus, we want to separate data loading operations from the rendering operation, and process them in a separate pass. Data requests on a cache will not be processed immediately, but instead will be added to a batch of requests to be processed in a separate *update phase* during the cache management presented in Section 7.3.

Thus, in order to manage the cache, our system operates in two steps, it interleaves a *ray-tracing phase* and an *update phase*. The first step traces rays until they need to render a part of the scene that is currently not present in the cache. In this case, the corresponding rays trigger a data request by adding it to the batch of requests as detailed in Section 7.3.3.

As we will see in Section 7.2.2, once a ray has issued a data request, two strategies can be employed. If we want to ensure a maximum rendering quality, the ray suspends its execution and saves all necessary data in a *state context buffer* to continue rendering in a successive rendering pass. Multiple rendering passes are issued per frame in order to provide all necessary data to get a complete image. On the contrary, if the priority is to ensure real-time interactivity, the ray can continue rendering using the lower resolution data present in the cache. In this case, only one rendering pass is used per frame. As we will see, these two strategies can be combined in order to provide a continuous range of degrees of interactivity.

7.2.1 Top-down refinement

The rendering process is organized into passes. Each pass interleaves a ray-casting phase, producing an image and 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.

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.

This multi-pass incremental update scheme is illustrated in Figure 7.6 with a simple case employing a real-time update strategy, giving the priority to the loading of the bricks over the subdivision of the octree:

- 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 the 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 the 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 with pass (4) the ray gets all data it needs and no more updates are necessary.

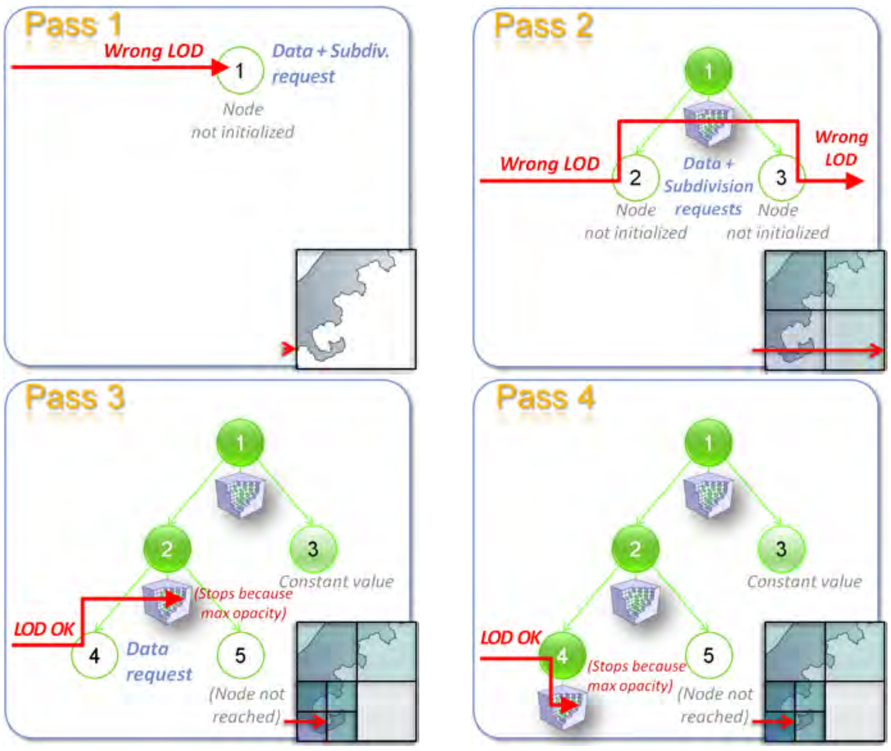


Figure 7.6. Illustration of ray-guided multi-pass update sequence for one ray.

7.2.2 Update strategies

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. We provide a control over the achieved image quality and interactivity by allowing the update scheme to be balanced between a *real-time* and a *"quality"* strategy.

Real-time strategy

In a full real-time strategy, only one rendering and one update pass are performed per frame. This can be achieved by producing an image even when data are missing. This is done by allowing rays to rely on a coarser brick that is actually present in upper levels of the octree when a brick is not present at the right resolution.

Another important parameter that controls the building of our structure is the respective priority between the subdivision of the octree and the loading of the bricks. Indeed, for a given node at the wrong LOD encountered by a ray, a choice has to be made between first requesting a brick for this node, and first requesting a subdivision. In the context of a real-time strategy, priority is given to the loading of the bricks. Giving priority to the brick requests allows to continuously get an increasing rendering precision among multiple frames. In addition, we ensure that lower resolution bricks will always be available in the octree, which allows us to always produce an image.

The disadvantage of this strategy is that for fast movements, due to the top-down building scheme, this leads to a lot of data loaded at lower resolution before the actual needed resolution is reached. Thus, this real-time strategy leads to higher convergence time before reaching the correct view resolution. With this approach, it takes more time to reach the correct required resolution, since time is taken to load bricks that are not needed at the correct LOD.

Quality-first strategy

In a quality-first strategy, the priority is not to quickly display a complete frame, but to quickly subdivide the octree structure in order to get a maximum quality as fast as possible. This is done by issuing multiple interleaved rendering and update passes per frame. Thus, priority is given to the subdivision of the octree. In this case, the required octree subdivision can be reached much more quickly, since no time is taken to load unnecessary bricks. However, in the context of a real-time strategy, such an approach would make the rendering quality converge less smoothly to the correct one.

In this mode, rays do not use lower resolution bricks to provide a complete image, but instead keep requesting subdivisions on the structure as long as the needed resolution is not reached. Once the needed subdivision is reached, rays ask for the required brick at this level of subdivision in order to produce an high-quality rendering.

Balanced strategy

In order to provide a control and a smooth transition between these real-time and quality strategies, we control the number of passes done per frame in order to get the higher rendering quality in a given time budget. To achieve this, quality-first passes are first computed with priority given to the subdivision of the octree, in order to quickly update the structure to the needed resolution. Then a final real-time pass ensures that a complete image will be displayed. The number of quality-first passes issued can be determined based on the remaining time budget for the frame, and an heuristic estimation of the time required for the final real-time pass.

Multi-pass ray-casting is achieved by providing rays the ability to suspend their execution whenever data are missing, and then to continue it from this point in the following rendering pass in a stop-and-go manner. The number of passes that is authorized per frame determines the degree of interactivity of the rendering.

The choice of the balance between real-time and quality strategies depends on the context of the application. For games for instance, it is acceptable that some frames show fewer 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 off-line rendering, physical simulations, or movie productions, for instance, it is important that each frame is accurately computed and, hence, the stop-and-go solution is most appropriate.

7.3 A High Performance GPU Cache for graphics applications

In this section, we detail the design of our generic GPU cache mechanism. While this cache has been primarily developed to manage our voxel octree data structure with bricks (Sec. 5), we demonstrate in Section 7.5.5 that it is generic enough to be used with any kind of data structure in the context of GPU ray-tracing.

Our cache automatically manages dedicated video memory regions we call pools (cf. Section 5.2.1). It can serve linear global memory sections as well as textures (cf. Section 1.3.4). As illustrated in figure 7.7, it is composed of a request interface (described in Section 7.3.3), that is used by ray-tracing kernels to emit data requests and usage information, a data loading interface allowing user defined *data Providers* (cf. Section 7.4) to load data inside the cache, and a management mechanism implemented as data parallel tasks (Sections 7.3.4 and 7.3.5).

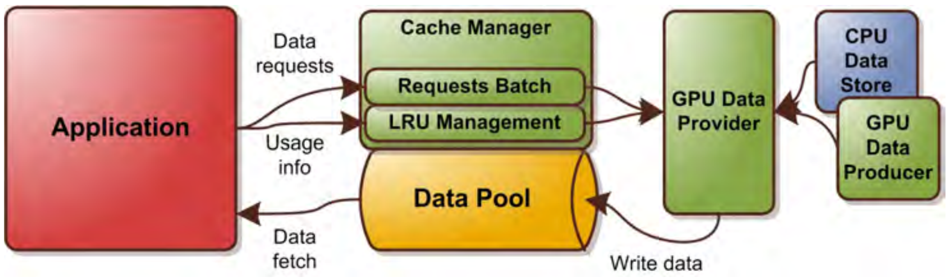


Figure 7.7. Global view of our generic GPU cache mechanism.

While previous caching methods for the GPU were CPU centric and relied on sequential computations, our method is completely run on the GPU. Having the cache managed on the GPU provides several advantages. The basic motivation for a parallel implementation of the cache mechanism is provided by the Amdahl's law [Wik11] that describes the maximum speedup that can be expected from a system given the ratio of codes running in parallel. Basically, this law shows that in order to get a maximum speedup from an increase of parallel processing units, it is critical to parallelize a maximum portion of a system. Thus, in order to offer a maximum scaling of our system with the increase of parallel processing power of the GPU, it is important to make a maximum portion of our system run in parallel. We demonstrate that managing a cache as a series of data-parallel tasks appears a lot more more efficient and greatly reduces management cost (Sec. 7.5.4). We also show how parallel streaming of the data directly from the GPU greatly reduces transfer overhead compared with CPU-triggered copies (Sec. 7.5.3). This scheme also allows efficient GPU data production that were not possible with previous approaches (Sec. 7.4). It also removes nearly all synchronizations between the GPU and the CPU, and totally frees the CPU from management computations.

Our system relies entirely on the GPU to manage the data structure, download necessary data from central memory (thanks to the latest CUDA features), and manage the cache mechanism. The CPU is used solely to make sure that the required data are available, making any supplementary work unnecessary. Our mechanism relies on the compute capabilities of modern GPUs (>= SM4.0, cf. Section 1.3). While we demonstrate it on NVIDIA hardware with a CUDA implementation, one can also easily implement it using a multi-vendor language like OpenCL or DirectX compute.

7.3.1 Cache structure and storage

As described in Section 5.2.1, we rely on video memory regions (that can be textures or linear memory) we call *pools* to store both the nodes of our octree structure and the voxel bricks. We generalize this concept of *data pools* in the context of our generic cache mechanism. These pools represent the physical memory regions managed by a cache mechanism. Following the formalism of virtual memory systems (cf. Section 2.5.1), a data pool is divided into fixed-length *pages* that are the atomic units managed inside a cache.

As for any virtual memory system, pages inside a pool are addressed through a *page table*. We define the page table as a simple array of references (pointers) into a pool, with no assumption on its actual structuring or spatial organization. Similarly, the actual encoding of a reference is defined depending on the client application, it is usually simple indexes inside the data pool.

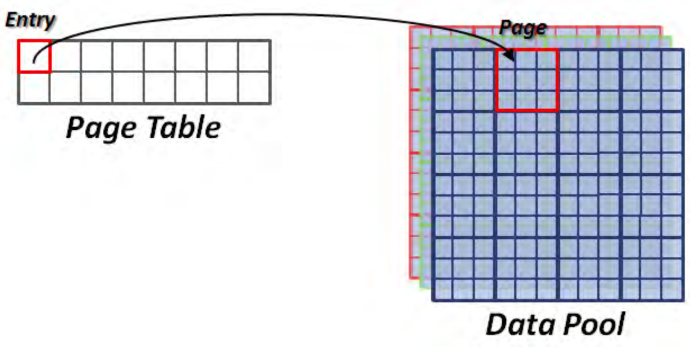


Figure 7.8. Illustration of the two components of our generic GPU cache structure: a page table with entries referencing pages inside a data pool. The data pool can contain an arbitrary number of layers

This pair of *page table* plus *data pool* defines our generic cache structure as illustrated in Figure 7.8. Since we target graphics applications, a data pool can have a 1D, 2D or 3D spatial dimensionality. Each data pool can be composed of multiple data layers (or fields) that are stored separately in memory. Each layer can contain a different set of data associated with each element of the pool. They can be implemented using different physical memory (texture or linear memory) and spatial organization (linear, Morton curve, texture blocks...). Elements can be of different sizes in each layer, only a one-to-one mapping between elements needs to be ensured (same number of elements in all layers). Typically, a pool containing voxels for instance could be composed of two layers: an "RGBA8" color layer stored in texture memory, and a "float" specular exponent layer stored in linear memory using a Morton curve.

Pages represent the basic element manipulated by the cache. They have the same 1D, 2D or 3D spatial dimensionality as the data pool they are stored-in. Their size can be freely configured depending on the application and the desired granularity of data management. The choice of this page size is one of the critical elements to ensure high performance of an application-controlled demand paging mechanism. It must be adapted to the data structures stored inside the cache.

A cache (meaning a data pool and its associated page table) can be used to store multiple instances of the same kind of data structure. As we will see in Section 7.4.1, in the context of real applications (in video games for instance), this allows us to store the structures of multiple different objects inside the same caches.

7.3.2 Building GigaVoxels data structures based on cached storages

In the context of GigaVoxels, we want to ensure a full scalability of our data structure with the increase of the resolution of the dataset. Thus, we propose to store both the octree structure and the associated bricks (cf. Chapter 5) in two separate caches, meaning that we manage both the node pool and the brick pool with two instances of our generic cache mechanism as illustrated in Figure 7.9. A page of the brick pool is sized to contain a single brick. Similarly, a page of the node pool is sized to contain a single *node tile*[2]. This allows us to virtualize our whole data structure, enabling deep zooms and exploration at arbitrary scales.

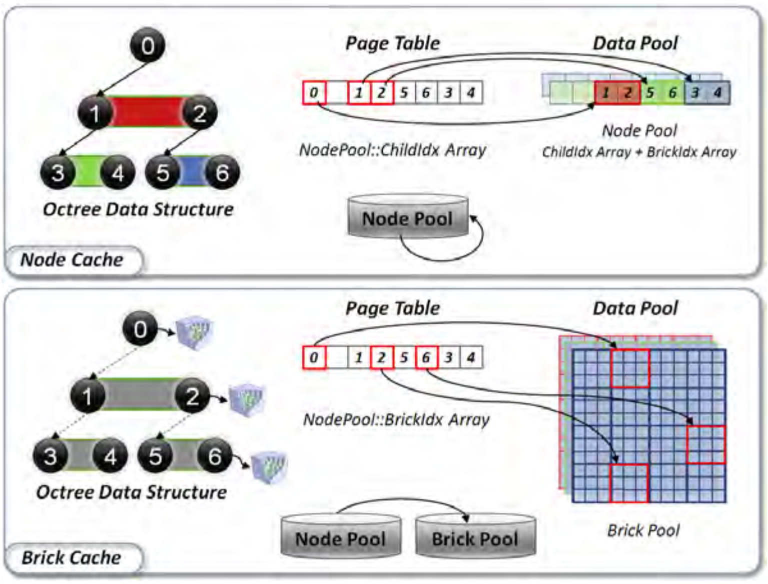


Figure 7.9. Illustration of the implementation as caches of the node pool and the brick pool storing the different elements of our voxel data structure.

Other approaches like [GMAG08] only manage the voxel brick dataset using a caching strategy, but limit the octree structure to a maximum resolution. In their case, the octree structure stored in video memory is updated every frame from a clone structure located in system memory. Such an approach induces a static transfer cost of the structure every frame, limiting its maximum extent. In our case, the rendering algorithm autonomously chooses the data resolution used for rendering (cf. chapter 6), which makes it possible to completely decouple the actual storage of the structure in the cache, from the state of the octree needed to render a given frame. In normal rendering situations, the stored structure is much larger than the one actually needed for a given frame that is only a subset of it. Thanks to that, many parts of the structure that are kept in the cache can be reused during the exploration of the scene.

Virtualized hierarchical page tables

The interesting point in our approach is that, instead of relying on dedicated page tables inside the two caches, whose sizes would limit the maximum number of different pages (node-tiles and bricks) that could be stored in each cache, we use the octree structure itself as the page table for the two caches. Indeed, both *bricks* of the brick pool and *node-tiles* of the node pool are already referenced by nodes of the octree in our data structure, as pointers located in different layers of the node pool. Thus, we can directly use these layers as page tables for the two caches. This scheme is illustrated in Figure 7.10.

[2]A set of $N \times N \times N$ nodes, with generally N=2 for an octree, cf. Section 5.2.1.

As described in Section 5.2.1, the octree structure is implemented in SOA (*Structure-Of-Arrays*) order using two separate arrays. The first array stores the `ChildIdx` field of the node structure (cf. Section 5.2.2, that is used to reference a node-tile that contains the set of sub-nodes (children) of a given node. Similarly, the second array of the octree structure stores the `BrickIdx` field used to reference the voxel brick associated to a node. Thus, the texttttChildIdx array is used as the page table for the brick cache, and the `BrickIdx` array is used as the page table for the node cache. This configuration is illustrated in both Figures 7.9 and 7.10 .

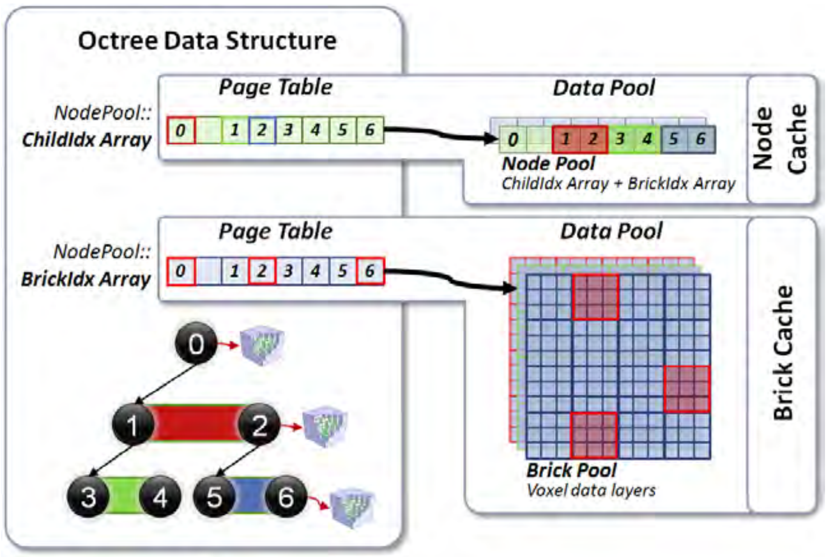


Figure 7.10. Illustration of the use of the octree structure itself as page tables for both the node cache and the brick cache.

As a result, the particularity of this system is that page tables themselves are hierarchical, virtualized and managed inside a cache, allowing for arbitrary large and detailed domains. In addition, since our original rendering scheme already relies on the same indirection to access data, no additional cost is induced by the translation from the virtual domain to the physical storage through the page table as is classically the case.

7.3.3 Interface with the cache

A client application using our generic cache mechanism (in the case of GigaVoxels, the ray-tracing algorithm) needs to interact in two ways with this cache. First, it needs to emit data requests (cache misses), when a page (not present in the cache) is required. Second, the application needs to provide the cache with the usage information required by the LRU replacement policy (presented in Section 7.3.4), and informs the cache each time it uses one of the page of the data pool.

Our cache is designed to be used in massively parallel environments, with thousands of threads running in parallel and interacting with it. This is the case for our ray-tracing rendering algorithm presented in Chapter 6, running one thread per ray launched on each pixel of the screen. In such environments, an important question is how to efficiently handle concurrent access on the cache from a high number of threads to provide usage information and emit data requests. The efficiency of the interface allowing this interaction is critical for the overall performance of our rendering algorithm, since it will be used each time a thread (a ray) accesses data present in the cache. Thus, we want a minimum impact on performance coming from this interface.

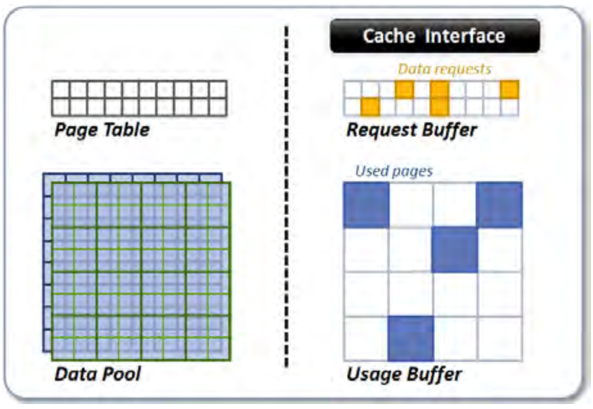


Figure 7.11. Illustration of our cache interface for data requests and tracking of used pages, based on a *request buffer* and a *usage buffer*.

Data requests on the page table

A data request is emitted by the client application each time an element of the page table is required but its associated page is not present in the cache. The goal is to build a batch of requests (a list containing all page requests) that we will be able to answer in parallel in a subsequent pass using our data producer mechanism described in Section 7.4. A naive approach would be to use a queue (FIFO) in which threads directly add their requests. However, in order to prevent race conditions, concurrent access to such a structure would require costly atomic operations [NVI11a] or complex parallel schemes [NVI09]. In addition, multiple threads can require the same data and thus ask for the same page. However, since we want to load this page only once, we must ensure the uniqueness of the request inside the queue, which represents a very costly operation.

Instead of directly building a list with unique elements, we rely on a special *request buffer* stored in the GPU global memory as illustrated in Figure 7.11. Its size is chosen to match exactly the number of elements in the page table. It is arranged in the same way in memory to ensure a one-to-one correspondence between entries in the page table and in the *request buffer*. Doing so, we can store one *request slot* for each entry of the page table. A data request is simply a boolean flag indicating that data are requested for a given entry in the page table. Each time a page is required for an entry of the page table, the slot in the request buffer associated with this entry is flagged. Then, as we will see in Section 7.3.5, a list of requests is built by collecting all the entries flagged in the request buffer.

This scheme has several advantages. First, it ensures the uniqueness of the requests in the request list. Second, there is no need to enforce any write ordering when a concurrent access is done since all threads write the same request flag value. This provides us with a fast request mechanism, and prevents us from relying on an "atomic" write operation that is a very costly operation on a parallel architecture (and especially on current generation GPUs) due to the fact that it forces the serialization of write operations.

Data usage information

As we have seen, our cache mechanism relies on an LRU (*Least Recently Used*) scheme. It allocates slots for new pages, by recycling those occupied by pages that have not been in use for the longest time. In order to reflect the actual usage of the pages in the cache, the application provides an information on what pages were used during a given rendering pass.

To collect this information, we use a similar approach than for data requests. We rely on a *usage buffer* that associates a usage flag with each page in the data pool, as illustrated in Figure 7.11. Whenever a page is used during a rendering pass, the application is in charge of writing the associated flag in the *usage buffer* to keep track of its usage. In the case of GigaVoxels, this information is provided by each ray during the ray-tracing of the data structure, for both the brick cache and the node cache.

Implementation details

In practice, both the *request buffer* and the *usage buffer* are implemented as an array in linear video memory so that it can be very quickly accessed from a CUDA kernel (or a fragment shader) during ray-tracing.

Instead of simply storing a boolean value for each slot of these buffers, we chose to store a 32 bit integer *timestamp*. This timestamp is affected with the time of the current pass (a global counter is maintained) by the threads of the client application in order to flag a slot. 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 and fast. Using such timestamps prevents us from clearing request and usage buffers at each frame, and entries containing the time of the current pass are known to be the flagged ones for the current pass.

Fine-grained usage and data request informations

In the scheme we just described in this section, there is no way to order and prioritize usage informations and data requests provided inside a given rendering pass. However, in a context of real-time rendering, it can happen that the available time budget per frame is not sufficient to allow processing all the data requests emitted in a given rendering pass. Thus, it becomes important to prioritize data requests in order to first proceed with the most important ones. We build upon the hypothesis that the most important requests are the ones emitted by the highest number of threads (rays). Indeed, in our GigaVoxels application, these are the ones that correspond to elements with the largest footprint on the screen. Also, in the context of a small cache with a lot of memory contention, it becomes useful to sort usage information inside a given rendering pass, in order to recycle in priority pages that were less visible in a previous pass.

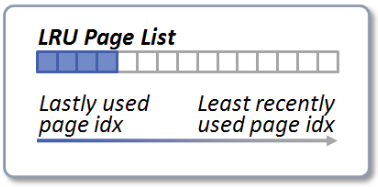
Thus, instead of simply storing a boolean flag, both our request buffer and our usage buffer can store counters incremented each time a rendering thread requests or uses a given page. During the processing of these buffers, counters will allow the sorting of the requests and usage information with a fine-grained intra-pass priority. Counters are incremented by rendering threads using atomic operations, in order to prevent race conditions on parallel read-modify-write operations. This scheme forces us to proceed with a cleaning of the two buffers at the beginning of each rendering pass, while it was not needed with the simpler approach.

7.3.4 LRU replacement policy

Now that we have seen how our cache interacts with a client application through its buffer-based interface, we will describe how we manage the LRU (Least Recently Used) replacement policy on the GPU. This LRU policy is used by our data request management scheme presented in Section 7.3.5 to determine *where* to store new data in the cache when a request for a page is emitted. It is based on the usage information provided by the usage buffer described in Section 7.3.3.

The principle of an LRU scheme is to replace the pages that have not been used for the longest period of time when new data need to be written into the cache. This scheme allows us to take advantage of the frame-to-frame coherence of the data needed to render a scene. In order to make sure the algorithm always discards the least recently used item, it is essential to keep track of what data was used, and when it was used.

LRU page list



We implement this LRU caching scheme on the GPU by maintaining a list of all the pages present in a cache, sorted by the time since they were last used (Fig. right). This list is called the *LRU page list*, it is stored in linear video memory and updated and maintained using data parallel operations on the GPU. This list contains as many entries as there are pages in the cache. Each entry stores the reference (or pointer) to a page in the cache. A reference is a 32bit value whose encoding depends on the application and the physical storage of the data pool (cf. Section 7.3.1).

In this *LRU page list*, references to the most recently used pages are kept at the beginning of the list, whereas those, not used for a long time, are at the end. When n new pages need to be inserted in the cache, replacing the n pages corresponding to the last n entries of the list perfectly respects the LRU scheme (cf. Figure 7.12). Using such a list gives us the nice property that the n elements addresses can be fetched in parallel by the threads in charge of loading data into the cache as described in Section 7.3.6.

Data-parallel LRU management

The LRU management is the first step of the cache management phase that is executed for each cache after each rendering pass (cf. Section 7.2). Usage information provided by the application through the *usage buffer* is used to maintain the *LRU page list* described in Section 7.3.4 in order to keep the most recently used elements at the beginning of the list, whereas those not used for a long time are at the end.

Sorting the *LRU page list* at each rendering pass on the GPU using a sorting algorithm would be prohibitive [SA08]. Instead, we rely on an incremental sorting scheme that makes use of two order-maintaining stream-compactions [BOA09]. This incremental sorting scheme, or maintenance procedure, is illustrated in Figure 7.12. Stream compaction, is a very important primitive building block for algorithms that exploit the massive data parallelism that is emerging in GPU hardware. It allows, in a parallel process, to compact elements of a list into a new list containing only the flagged elements [BOA09]. Basically, the idea is to grab in the *LRU page list* all the references to pages that were used in the current rendering pass, and to put them at the end of the list without modifying the order of the other references.

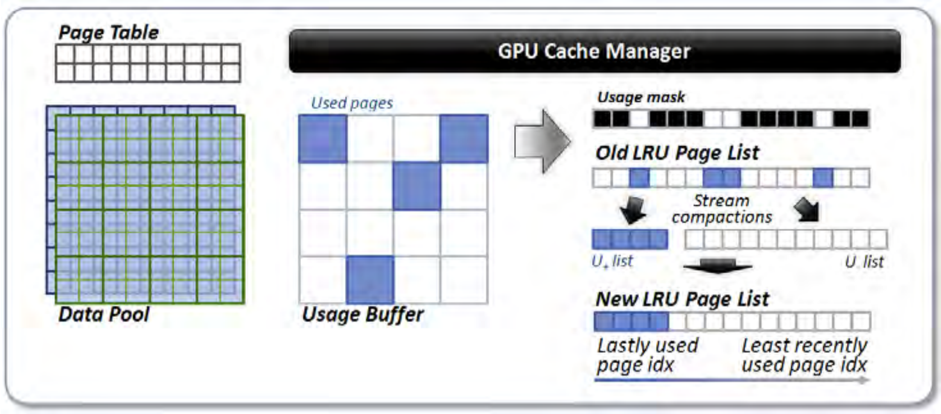


Figure 7.12. Incremental update of the *LRU page list* based on a per-page usage information provided inside a usage buffer by the application. Two stream compactions are used in order to keep references to lastly used pages at the beginning of the list.

As we have seen in Section 7.3.3, usage information in the cache is provided on a per-page granularity using a *usage buffer*. We use this *usage buffer* to generate a *usage mask* that indicates for each element of the *LRU page list* whether it was used in the current rendering pass. In practice, this is done in a data-parallel way with CUDA by launching a kernel with one thread per element of the *LRU page list*. Each thread checks in the *usage buffer* if the page was used in the current pass. If the stored timestamp matches the time of the current frame, the element is flagged.

The *LRU page list* then undergoes two stream compaction steps. The first stream compaction will create a list $\mathcal{U}_+$ that only contains the used elements, the second a list that contains all unused elements $\mathcal{U}_-$. Because inside each of these sublists, the order remains the same, the list of the unused elements will still have the oldest elements at the end and the most recently used in the beginning. Therefore, when concatenating $\mathcal{U}_+$ to the beginning of the $\mathcal{U}_-$ we inherently sort the usage list.

7.3.5 Managing data requests

In the previous section we saw how we efficiently maintain the LRU replacement policy on the GPU. This scheme provides us with information on *where* new data need to be loaded inside the cache through the ordered *LRU page list*. We will now look at how to handle data requests emitted on the cache interface (Sec. 7.3.3) during a rendering pass.

Computing the compacted request list

After the rendering pass, the *request buffer* presented in Section 7.3.3 is filled with the time of the current frame for each node that needs data to be loaded into the cache. In order to process these requests efficiently in parallel (Sec. 7.4), we want to collect the needed load operations in a compact *request list*. This compacted *request list* is computed by applying a stream compaction operation (cf. Section 7.3.4) on the *request buffer*, as illustrated in Figure 7.13.

We rely on a slightly modified stream compaction operation in order to interpret the timestamp information stored in the *request buffer* as a boolean flag, based on the comparison with the time of the current pass. We also tweaked the last step of scattering of the stream compaction (cf. [BOA09]) in order to directly write (without memory read) inside the request list the index inside the page table of selected elements.

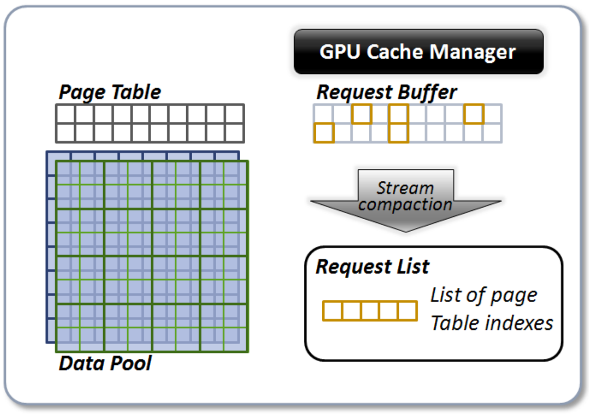


Figure 7.13. Generation of the compacted *request list* using a stream compaction operation on the *request buffer*.

At the end, this compact *request list* contains the index in the page table of all the entries that require a data load. In addition to a compact *request list* itself, the stream compaction operation provides us with another interesting information that is the length of this list, corresponding to the total number of flagged elements in the piece of *request buffer*. Indeed, this length also corresponds to number of pages that need to be loaded into the cache.

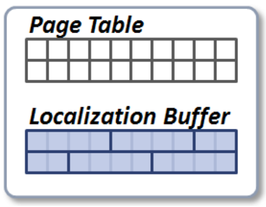
Prioritizing requests

As we have seen in Section 7.3.3, data requests in the request buffer can contain information on the number of threads that have issued the same request, instead of a simple timestamp. This scheme has a more important impact on the rendering performance than the simple boolean flag and is not used uppermost. However, this information on the number of threads issuing a data request can be used to parameterize the requests. In the context of an application with a strict real-time constraint, the number n of data requests handled per frame can be limited in order to bound the time taken by the loading of data inside the cache.

In our application, data requests that have been triggered by the highest number of threads are likely to be the most important ones, because they are the ones with the biggest impact in the image quality. In such a context, it is important to first serve data requests that have been triggered by the highest number of threads, used as a weighting factor. Thus, once compacted into the request list, data requests can be sorted based on this weighting factor, and only the first *n* requests of this list can be answered in a given frame.

Page table localization

As explained previously, the page table of our generic cache is manipulated as a simple list of references to pages stored into the cache. We do not want to make any assumptions about the actual data organization inside this page table. In the case of GigaVoxels, this page table actually stores nodes organized in an octree (or a set of octrees, cf. Section 7.4.1). In another graphic application, data could be organized into a Kd-tree or a regular grid for instance. In a non-graphic application, data could simply be a list of objects with no more structure. In order to stay generic, the cache management mechanism does not have to know anything about this data organization inside the page table.



However, the actual data organization inside a cache must be taken into account in order for the producer in charge of answering the data request to know *"what"* data it needs to load into the cache. As described in previous sections, a data request is only identified by the index into the page table of the entry data is requested for. Thus, there is no relationship between the index of a page table entry which is provided for the data requests, and the actual localization inside the underlining data structure.

The mapping between an index into the page table and the actual data provider and localization inside the underlining data structure is made through what we call *localization information*. Depending on the application, this localization information can be either implicit (in the case of a simple list or a grid for instance) or can have to be stored explicitly (e.g. for an octree or a Kd-tree) in a *localization buffer*. The actual format of localization information is also totally application dependent since it depends on the underlining data structure. Thus, it is manipulated as an opaque format by our generic cache management mechanism, with only the user defined data producer interpreting it.

Concretely, we abstract the management of this localization information inside the page table, which is able to provide a localization information for any entry of the table. In case of explicit localization information, the actual management of the *localization buffer* used to keep the mapping between an index into the page table and a localization information is done by application specific code. If the page table itself is "dynamic" and managed by a cache mechanism as it is in the case in GigaVoxels, the *localization buffer* is updated by the user defined data producer when generating a page.

The *localization buffer* must be kept as compact as possible, in order to limit the memory overhead it induces. Explicit localization information does not have to be specified in a per-entry basis for the page table. Indeed, in our application for instance, since the page table is in fact a layer of our node pool, it is enough to store one localization information per node-tile, since inside a node-tile localization can be implicitly reconstructed.

7.3.6 Parallel data load

Once the compacted *request list* has been computed with the data requests and the *LRU page list* has been updated to account for the data usage in the current pass, actual data loading into the cache can be achieved. We are now able to determine *what* data to load (the entry in the page table) and *where* to store it (through the LRU policy), we have everything needed to actually load data into the cache.

The loading of data into the cache is done through a generic mechanism we call a *GPU data provider*. GPU data providers are in charge of writing the data into the pages that are physically located into a data pool (cf. Section 7.3.1). As we will see in Section 7.4, data providers are implemented by *GPU producers* that are application dependent and can implement multiple data providers for different caches. Producers can be either procedural, generating new data on the fly inside the cache, or loading producers, loading data from the system memory. The data loading procedure is initiated through the launch of a generic CUDA loading kernel, in charge of calling the loading function of the user defined data provider.

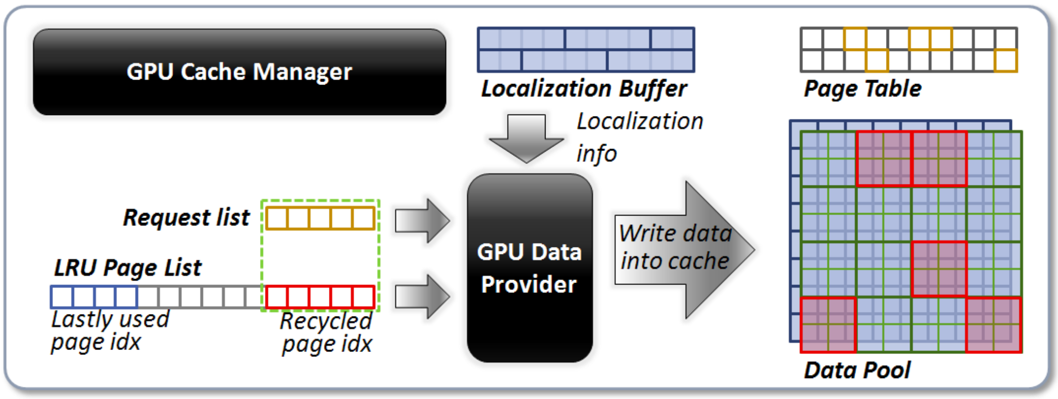


Figure 7.14. Illustration of the parallel loading procedure of data inside the cache.

Scheduling of data loading threads

The generic kernel responsible for loading into the cache is launched by associating one CUDA thread group [NVI11a] per page that needs to be filled. Indeed, since cache pages contain multiple elements, having one thread group per page allows the parallel processing of these elements by the producer, as well as the parallel processing of all the requested pages. In addition, thread groups provide the ability for the threads in a group to communicate through a fast shared memory [NVI11a]. This organization of parallel computation allows the threads in charge of a given page to collaborate together for an efficient loading or generation of the data.

Inside each group, the number of threads is chosen in order to maximize the occupancy of the GPU multiprocessors [NVI11a] and to be able to share computations (like destination addresses computation) between all data written into a given page. This scheduling of the working data-parallel threads is a critical point for the efficient execution of the production task. It totally depends on the size of the pages and the actual processing of the producer. Therefore this thread group organization is provided directly by the user-defined GPU producer.

Interface with user-defined producers

As illustrated in Figure 7.14, the interface between the generic data loading kernel and the user defined GPU producer is made by passing the index of the page table element that requests data (extracted from the *request list*) and the index of the page in the data pool to write data into (extracted from the *LRU page list*) to the data loading function of the producer, on a thread group granularity. These two informations provide the producer with both the *"what"* data to load and the *"where"* to load it into the data pool.

For each request, the producer will usually use the provided page to write data into, but there are cases where it will not write data or will not use the provided page:

- The first case is when, during the production of the data, it appears that the generated data are not really necessary. In the case of GigaVoxels for instance, this happens when during the production of a brick, the value of all voxels appears to be the same. In this case, we do not want to store an actual brick, but instead a constant value in the node. There are also situations where the data needed to fulfill a request are not available immediately to the producer. In this case, the producer may differ the loading of these data to a subsequent request.
- The second case is when we want to share pages between multiple entries of the page table. In GigaVoxels, this situation appears when we want to instantiate a brick inside multiple octree nodes, or we want to instantiate sub-parts of the octree (cf. Section 8.1). In this situation, the producer will not write data into a new page, but instead will provide the index of another page in the pool it wants to reference to.

In order to handle these cases, the GPU producer itself returns the index of the actual page to be written into the page table entry. The page table is automatically updated with this index by the generic loading kernel. In the usual case the returned index is actually the index of the pages that were reserved in the *LRU page list* to be recycled to handle this request. If it is not, it means that we are in one of the two situations we presented previously. In the first case, this index will be *null*, and in the second case it will be the index of the existing page the producer wants to reference to.

In the case where the page reserved in the *LRU page list* is actually used and recycled, an invalidation procedure described in Section 7.3.7 needs to be executed, in order to invalidate all references in the page table that were previously pointing to this recycled page. In order to prevent the reference we just created to be invalidated during this procedure, a flag indicating this case is put into the corresponding request element of the compacted *request list* (in practice a single most significant bit is set to one).

7.3.7 LRU invalidation procedure

Once new data have been loaded into the cache, some pages of the cache have been recycled to load new data, and the data previously stored in these pages have been overwritten. In order to keep the page table coherent with the new data, a special procedure needs to be applied in order to invalidate in the page table all previous references to the recycled pages. Indeed, without such a procedure, overwriting a page might lead to another one still pointing to this location assuming that the old content is still present. In other words, each time cache pages are recycled 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. This cleanup procedure is non-trivial because pages in the cache could be referenced by more than just a single page table entry (in the case of instantiated data).

This invalidation procedure must be done after the data production step, since the actual page usage is only known at this time, due to the ability of the producer not to produce data, or to use an existing page instead of loading a new one (cf. Section 7.3.6).

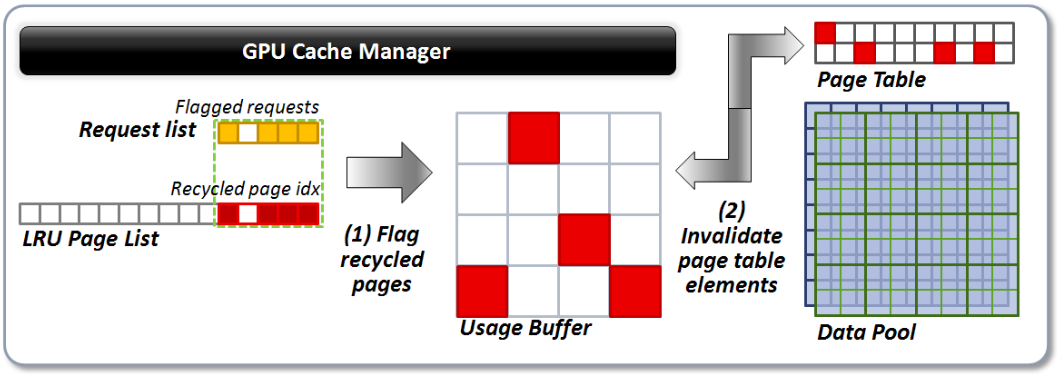


Figure 7.15. Illustration of our two steps invalidation procedure of recycled page table references.

Two steps procedure

Our solution to invalidate old page table references reliably is to use a two step procedure as illustrated in Figure 7.15.

First, for each of the requests of the compacted *request list* indicated as using a recycled page during the data load procedure (cf. Section 7.3.6), a flag is associated with the actual recycled page in the data pool (whose index is provided by the *LRU page list*). This flag is efficiently written in parallel for all entries of the compacted *request list* using a CUDA kernel with one thread per entry of the list.

Then in a second step, all entries of the page table are tested in order to detect if they reference a page that has been flagged as overwritten. If it is the case, and the corresponding entry in the *request list* is not flagged as being written in the current pass, the entry in the page table is set to *null* to indicate that the referred element is no longer present. Since all invalidation operations are independent, this step is also executed efficiently on the GPU as a data parallel kernel.

One of the priorities during the design of the whole cache mechanism was to keep its memory overhead as low as possible. Following that goal, we chose to rely on the *usage buffer* used for the LRU management procedure (cf. Section 7.3.4) in order to store the invalidation flag written in the first step. The invalidation flag is encoded as a special reserved value, not used for the LRU (in practice we use zero). This trick allows the invalidation procedure not to use any additional memory.

Producer specific invalidation processing

When a page table reference is invalidated, there are some GPU producers that need to perform a special processing. This is the case for instance for the data load producer described in Section 7.4.3, that needs to keep a system memory address in the data structure. To allow this scheme, an invalidation function is called by the invalidation kernel (step 2) on the GPU provider interface whenever a page table reference needs to be invalidated. This invalidation function is in charge of invalidating the reference in the page table entry and can perform whatever processing it needs. This function is not intended to perform complex computations that would require a parallel processing. It is intended to perform only a simple serial processing and it is triggered only using a single thread per invalidated entry.

This scheme also allows the producer to implement a write-back procedure for cached data. Such a procedure is not useful in our application, since cached data are simply used for rendering, and thus are not modified. However, one can imagine other usages of our cache where data stored in the cache could be modified on the GPU, and would need to be written-back to the system memory when their pages are recycled, in order to preserve their value for future use.

7.3.8 Application specific optimizations

As we have seen, in our GigaVoxels application both a node cache and a brick cache will always be instantiated in order to build and manage the data structure (Sec. 5). Thus, some memory buffers can be factored and shared between these two caches in order to save video memory. This is the case for the *request buffer* (Sec. 7.3.3) as we will see in the next section. It is also the case for the *localization buffer* (Sec. 7.3.3), since in our GigaVoxels application it describes the same octree structure in both the node cache and the brick cache. Also, all temporary buffers like the *compacted request list* that does not need to maintain data between passes can be factored between the two caches.

Factoring the request buffer storage

The *request buffer* presented in Section 7.3.3 is used to handle data requests on a cache. Both the node cache and the brick cache use the node pool as a page table. Thus, they both handle data requests on the same domain. Therefore, we chose to use the same video memory region to store a *request buffer* shared between both caches. Inside this buffer, the discrimination between requests emitted for one cache or the other is made through a special flag added to each request slot as a single bit.

This restricts the ability to emit requests for both caches at the same time for a given element, but this appears not to be a problem in our application. Depending on the streaming strategy (cf. Section 7.2), a single ray will generally issue either a node subdivision request on the node cache or a brick request on the brick cache, but not both at the same time. Nevertheless, with some streaming strategies, different rays can sometimes issue different kinds of requests. This is not the case for the *"real-time"* strategy presented in Section 7.2 since bricks are always requested before requesting a node subdivision, in order to always ensure the ability to render a complete image, even if it is not at correct resolution. However, this is the case for the *"quality first"* strategy presented in Section 7.2. In this case, some rays can ask for a brick because they detect that the octree is at the right resolution for their needs, while other rays can request a subdivision because they need higher resolution. Therefore in this case a conflict is possible but we do not consider this situation as a problem. Indeed, only one of the two requests will be handled in a given rendering pass. Fortunately, the latter will be in the next pass because, now that one request has been handled, subsequent requests for this node can only be of the other type.

This scheme does introduce a one-pass delay for the request to be handled in case a conflict actually occurs, but we do not consider it as a problem in the context of an off-line *"quality first"* strategy. The main benefit of such a scheme is that it keeps us from relying on an atomic operation on the *request buffer*, that would impact rendering performance. This would be necessary in case we would like to be able to handle both types of requests at the same time using two bits.

Using this kind of shared request buffer is handled by our generic cache by allowing the application to provide the test function used in the stream compaction operation described in Section 7.3.5. This test function is used to select the element of the request buffer that will be kept by the compaction operation. The behavior of the default function is to keep elements with a timestamp corresponding to the time of the current frame. To handle a shared *request buffer*, this function is just slightly modified in order to test the bit indicating the type of request.

With this approach, two stream compactions are still performed on the shared *request buffer*. One for the management of the node cache, and the other for the brick cache. In our tests, this stream compaction appeared to be one of the most costly operations done for the management of a cache. In order to reduce the cost of having to perform it twice, we rely on an additional optimization pass that performs a first stream compaction on the whole *request buffer* that extracts all valid requests. This compacted *request buffer* is a lot smaller than the original one. It is then passed to both caches and handled exactly in the same way as if it was the original *request buffer*. The difference is that the per-cache stream compaction is a lot faster thanks to the reduced size of the buffer.

Implementation summary

Figure 7.16 presents a summary of all permanent GPU memory regions used by the two instances of our cache mechanism managing the node pool and the brick pool in the context of GigaVoxels. In this section, we present a typical usage scenario of our cache mechanism inside GigaVoxels, in order to analyze the impact our caches in terms of memory occupancy.

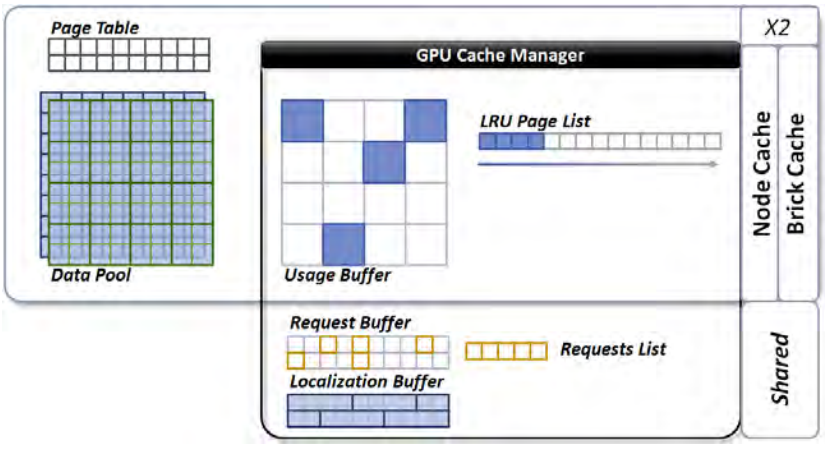


Figure 7.16. Summary of every GPU memory regions allocated for a typical usage of our cache mechanisms in GigaVoxels.

In our typical scenario, the brick pool is allocated with $48^3 = 110592$ bricks, each containing 8^3 voxels. This represents 384^3 voxels containing two 32bit values (a 4 byte color and a 4 byte normal distribution as described in Section 4.5), for a total of 432MB. The node pool is allocated with 262144 node-tiles, representing 2097152 nodes each containing 8 bytes, for a total of 16MB.

- **Node Pool**: 262144 node-tiles ($2 \times 2 \times 2$ nodes), 2097152 nodes $\times$ 8Bytes = 16MB
- **Brick Pool**: 110592 bricks (8^3 voxels), 56623104 voxels $\times$ 8 bytes = 432MB

The node pool is managed with a node cache containing:

- **Usage Buffer**: 262144 $\times$ 4 bytes = 1MB
- **LRU Page List**: 262144 $\times$ 4 bytes = 1MB

And the brick pool is managed with a brick cache containing:

- **Usage Buffer**: 110592 $\times$ 4 bytes = 432KB
- **LRU Page List**: 110592 $\times$ 4 bytes = 432KB

These two caches share the following memory regions:

- **Request Buffer**: 2097152 $\times$ 4Bytes = 8MB
- **Request List**: 262144 $\times$ 4Bytes = 1MB
- **Localization Buffer**: 262144 $\times$ (4 + 4) = 8Bytes = 2MB

Thus, total video memory used by the two caches for their operation is 13.84MB, compared to the 448MB of data actually stored and managed by the caches. Thus, the memory required for the management of our caches represents only a 3.08% overhead in this typical situation.

7.4 Handling data requests : Voxel producers

Requests made on our generic cache mechanism are handled through a *data provider interface*. A data provider interface is actually implemented by a GPU data Producer, that can actually implement multiple provider interfaces, in order to answer multiple caches. Data producers are in charge of actually fulfilling data requests by generating or loading all data necessary for a given voxel structure (Sec. 5). By implementing multiple provider interfaces, a single data Producer can manage the multiple types of data that are necessary to build a given data structure.

In the case of GigaVoxels, as illustrated in Figure 7.17, producers implement two provider interfaces, one for the node cache and one for the brick cache. Thus, a producer provides the data needed for both building the octree structure and filling bricks.

As we will see in Section 7.4.1, each GPU cache can rely on multiple data Providers to get data from. This makes it possible to connect multiple Producers to the same cache, and thus to store data for multiple different structures associated with different objects of a scene in the same cache.

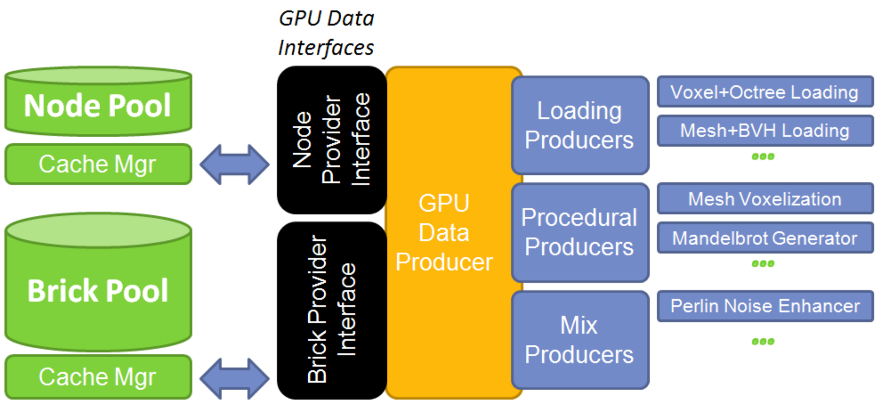


Figure 7.17. Illustration of a GigaVoxels data producer implementing a data provider interface for both the node cache and the brick cache, thus providing both octree nodes and brick data. Multiple kinds of data producers can be used, a few examples are shown here.

GPU producers directly write into the video memory from data-parallel production threads generated by a kernel launch as described in Section 7.3.6. In GigaVoxels, data can come from any kind of source. We identified three main classes of data producers than can be used within our voxel based rendering pipeline:

- The first type is loading producers. Loading producers load voxels and octree data precomputed and stored in system memory. The download of data in system memory from CUDA threads is made efficient by relying on a direct mapping of the system memory inside the address space of the GPU [NVI11b]. Whereas previous solutions had to trigger many independent updates, our solution transfers data more efficiently in parallel from multiple threads.
- The second type is procedural producers. Procedural producers generate data entirely on the fly on the GPU to answer data requests. This generation can be done based on a mathematical function (such as a fractal), or from the transformation of another representation stored on the GPU. For instance, we demonstrate in Section 7.4.6 a mesh-voxelisation producer that generates data (to build the structure and fill the voxel bricks) by voxelizing a triangle mesh stored inside the video memory (in a grid acceleration structure). We also show how procedural details can be added in order to amplify the voxelized mesh.
- The third type is mixed producers. Mixed producers both load data from system memory and add procedural details in order to amplify them.

In practice, our GPU producers are composed of both a CPU and a GPU function. The CPU function is mainly in charge of passing configuration parameters to the GPU production function in charge of actually loading data in the cache. In the case of producers loading data from the system memory, the CPU function is also in charge of preparing data and making them available in system memory so that the GPU loading function can access them.

7.4.1 Managing multiple objects

We call objects individual voxel-based models that are used in a given scene and that can be rendered separately using our ray-casting algorithm (Sec. 6). Each object has its own data structure stored in video memory and used for rendering. Objects that share the same kind of data structure (in our case the octree-based structure with associated bricks presented in Section 5) can be stored in the same caches, since the type of data to store is the same and they require the same page size (in our case page sizes correspond to the size of a node-tile and the size of a brick).

In the cache management scheme described in Section 7.3, we assumed that data stored in a cache came from the structure of a single object, thus from a single Producer. In practice, we want to be able to use a cache to store data coming from multiple objects. These objects must be instances of the same type, since a given cache is dedicated to the storage of a specific set of types organized in the data layers of a data pool. Data for a given object are provided by an instance of a GPU producer dedicated to provide these data.

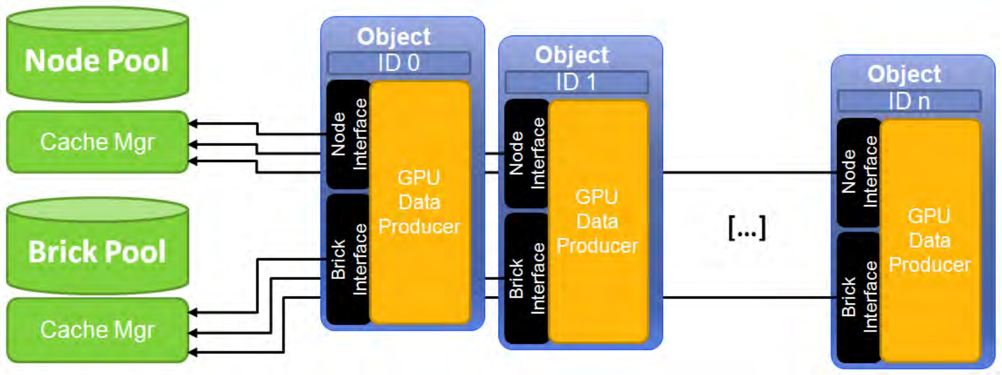


Figure 7.18. Illustration of multiple objects binding to the cache mechanisms in GigaVoxels.

To allow this usage, a GPU cache accepts the connection of a set of producers implementing the provider interface accepted by the cache. Each connected producer is identified by an instance or *object id*. This *object id* information is associated to each entry of the page table. It is used by the cache manager to route data requests to the right data provider interface. In practice, this information is added to the localization information described in Section 7.3.5.

In order to ensure an efficient data generation, a segmentation pass is performed on the compacted request list (cf. Section 7.3.5) in order to group together all requests intended for a given producer. These requests can then be handled together by the same kernel call using the dedicated GPU producer.

Also, our cache can serve multiple rendering client applications totally transparently through our cache interface described in Section 7.3.3.

7.4.2 Writing into texture pools

GPU producers are in charge of directly writing data inside pages located into the data pools. As we have seen in Section 7.3.6, this writing is done in parallel by multiple GPU threads executing a *data production function* of the producer. The GPU pools used to store the data maintained in caches can be located inside two kinds of video memories: the linear memory and the texture memory (cf. Section 1.3). Thus, a GPU Producer must be able to write directly into such memories.

The problem of texture memory

Writing into a linear video memory from a CUDA thread is a straightforward operation. It is done, as on the CPU, through a classical pointer dereferencing. However, writing into a texture memory is not so straightforward. Indeed, texture memory can not be addressed directly in CUDA. It is encoded in a proprietary format dedicated to providing a fast texture sampling. For a long time, this memory was not even writable from a CUDA thread. With the lastest generation of NVIDIA hardware (based on the Fermi architecture [fer10]), a Surface API was introduced in CUDA, providing random read/write access into textures. The problem is that this API only supports 1D, 2D and Layered 2D[3] textures, and not 3D textures.

In GigaVoxels, our brick pool is stored inside a 3D texture, to be able to take advantage of the hardware trilinear filtering, the 3D locality cache organization and the 3D addressing. In order to motivate the usage of a 3D texture, we compared the performance of texture sampling when using a Layered 2D texture and a 3D texture for ray-casting on last generation hardware. The results of these experiments are presented in appendix A.1 and clearly show the advantage of 3D textures in our application.

Reverse-engineering texture format

In order to be able to use 3D textures together with GPU based producers, and to keep both fast sampling access during rendering, and fast write access during data loading, we designed an alternative write access to 3D textures. To do so, we reverse-engineered the proprietary 3D texture format as well as the CUDA API in order to get a direct access to the texture memory from a CUDA thread through standard video memory pointers.

7.4.3 GPU load producer

The first kind of producer that we developed for GigaVoxels is the loading producer that loads pre-computed octree nodes and voxel bricks (generated following our pre-integration scheme described in Chapter 4) and stored inside the system memory. It is used to implement an actual caching scheme between a full pre-computed structure stored in system memory and the working subset of this structure maintained in video memory for rendering. Of course, this GPU producer assumes that the whole dataset fits inside the system memory, that is usually many times larger than the video memory. We will see in the next section how we can manage larger datasets that do not even fit inside the system memory.

Other approaches like [GMAG08] transfer individual data elements (voxel bricks) from the system memory to the video memory using copy operations triggered from the CPU. Each individual element has to be copied to a different position in the data pool that is specified by the LRU scheme. The problem with such an approach is that the cpu-to-gpu copy operation used to transfer each element has a long latency of execution, probably due to driver software overhead and synchronizations between

[3] Layered 2D textures are composed of a stack of 2D textures that do not support interpolation between layers

successive calls. This overhead makes a series of such copies very inefficient when a small amount of memory is transfered per copy operation. In this case, as shown by our experiments presented in Section 7.5.3, the bandwidth available on the bus connecting the graphics card to the main memory is not exploited fully and the hardware is underused. This problem appears to be even worse when doing transfers inside a texture (as it is the case for our voxel bricks). In this case, in addition to the CPU software overhead, we observed an additional cost per copy operation that is probably due to the format conversion operation needed to transform the linear memory provided by the application into a hardware specific texture format. In addition, this problem prevents transfers to texture memory to be proceed asynchronously, with computations done on the GPU at the same time.

This problem appears very important in our case, since we specifically designes our cache mechanism to be able to operate on small units of cached data (pages), in order to provide a very fine caching granularity and reduce over-storage of non-useful data. This is true for the bricks that we try to keep small (8^3 or 3^3 voxels) and for the nodes that we group inside $2 \times 2 \times 2$ node tiles (however, note that in practice we store nodes inside a linear memory region).

GPU based data download

In order to overcome this problem, we propose to do this memory transfer manually from CUDA threads directly into our GPU data production function. By doing so, we are able to reach performances close to the theoretical bandwidth of the bus as shown in Section 7.5.3. To perform this loading, we rely on a feature of CUDA named zero-copy [NVI11a] that allows us to access a subpart of the system memory mapped inside the GPU address space directly from a CUDA thread. To be efficient and use the full available bandwidth with the system memory, such an access must be made on adjacent words for each thread of the same CUDA *warp* [NVI11a] (this property is called coalescing). We ensure a maximum efficiency of such a GPU producer by carefully choosing the CUDA thread blocks configuration used by the GPU data production function (Sec. 7.3.6). In addition, this scheme also ensures a best usage of the available bandwidth with the video memory, thanks to the very large amount of memory accesses initiated in parallel from multiple warps processed inside different *Multi Processors* [NVI11a] of the GPU.

Compressed transmission

In addition to the immediate performance gain, this GPU-based loading approach allows us to perform other interesting optimizations. Data loaded from system memory can be transfered in a compressed form, and decompressed on the fly by the GPU producer before being written into the cache in video memory. With such a scheme, the GPU cache acts as a temporary cache for uncompressed data used for rendering. The interesting point in this scheme is that all computations that are done during the data transfer are virtually free to process. Indeed, the latency of memory access done from a thread into the system memory is hundreds of time higher than the latency of an arithmetic operation, thus even complex computations done in parallel with the transfer inside loading threads happen to be totally masked by the transfer.

Maintaining system memory source data location

Now that we have seen how we efficiently transfer data from the system memory, one remaining question is how the GPU producer gets the information on *where* given data (needed to fill a page) are located in system memory. The simplest approach would be to keep this information as an explicit localization information that would be different for both the node cache and the brick cache. It would be stored inside *localization arrays* associated with the page tables (cf. Section7.3.5).

But this would require a lot of additional storage since one unique system memory location would have to be associated with each entry of the two page tables. Such a scheme would be like duplicating the octree structure with system memory locations for the sub-nodes and the bricks, and would require twice the video memory storage for it.

Instead of relying on such additional *localization arrays*, we rely on the observation that a system memory location is only needed for entries of the page table that do not reference a page already loaded in the cache. Thus, for entries of the page table that contain a *null* pointer. In this case, instead of storing a *null* value inside the page table, we store the address in system memory where the page can be loaded from. Thus, when the page is requested, this address can be used by the Producer to actually load the data into the GPU cache. In order to differentiate valid references to pages actually loaded in the cache, from references in the system memory of pages not loaded in the cache, we add a special flag to each entry of the page table. This flag indicates if the reference is actually a valid reference to a page in the GPU cache, or not. Thanks to this scheme, we rely directly on the GPU data structure to keep references to the data located in system memory, with absolutely no memory overhead.

7.4.4 Loading from disk and system memory caching

Even if it is very fast, the problem with the loading producer we described in the previous section is that it restricts the total amount of voxel data that can be manipulated to what can fit inside the system memory. Allowing out-of-core management of data located on disk was not the priority of our work, and we focused on the caching inside the video memory. However, we will quickly show that our scheme can be extended to manipulate data larger than the system memory and stored on disk, with a special *disk loading producer*. The overall idea is to add a second level of caching inside the system memory. This caching is totally demand-driven exactly as our caching inside the video memory is. In order to keep it simple, and since it was not our primary focus, we implemented a simple FIFO replacement policy (Sec. 2.5.1).

Instead of directly storing flagged references to pages kept stored in system memory (as it is the case for our simple scheme explained previously), the node-tiles loaded from the system memory contain null pointers for all their nodes to both associated bricks and children node-tiles. However, we rely on the localization information associated with a node-tile (cf. Section 7.3.5) to store the address in system memory of this tile. Thus, when a sub-tile or a brick is required for one of the nodes of a tile loaded in the cache, the GPU producer checks in system memory (based on the address stored in the localization information) if the address of this associated element is present. If it is, the producer can directly load it from system memory.

If it is not, the producer simply does nothing (and alerts the cache manager as explained in Section 7.3.6). However, at each rendering pass, the batch of requests of both the node cache and the brick cache are downloaded to the CPU[4]. This allows it to detect from the CPU when given data are required and needs to be loaded from disk. This loading can be done totally asynchronously from the rest of the application. At each successive rendering pass, the same data will be requested. If they have been loaded in the meantime, their addresses will be available in system memory and the GPU producer will be able to load it. If they have not been loaded yet, the GPU producer still does nothing, and the request will be fulfilled in a successive rendering pass.

Depending on the memory available in system memory, pre-fetching schemes can be employed in order to load more data than required from disk. In case these data are requested later, they would be already present in system memory and ready to be downloaded by the GPU producer.

[4]This represents a small amount of data per pass and does not entail a significant overhead.

7.4.5 Examples of dynamically loaded datasets

Figure 7.19 shows the result of interactive rendering with GigaVoxels of two datasets loaded dynamically and on-demand using our GPU loading producer. The first dataset in 7.19(a) is a lion model generated using 3D-Coat [Shp11], a voxel sculpting software. The full dataset has a resolution of 2048^3 voxels and represents 64GB on disk, with 8 bytes per voxel (a RGBA8 color channel and a vec4 normal distribution). It is rendered with our approach at 60-80 FPS depending on the exploration speed and distance to the object, on an NVIDIA GTX480 and at 512×512 rendering resolution. The second dataset in 7.19(b) has been generated by tomographic reconstruction [CCF94] from the X-ray scan of a lizard. Its resolution is 2048^3 and it represents 32GB of data on disk.

Also, the scene presented in Figure 7.1 (this chapter's teaser) has been rendered at 20-40FPS using GigaVoxels on an NVIDIA GTX 280. Its full resolution is 8192^3 voxels and it has been generated by cloning a single 1024^3 medical dataset 8 times on each axis. It is stored actually cloned on disk and loaded dynamically using our GPU disk loading producer.

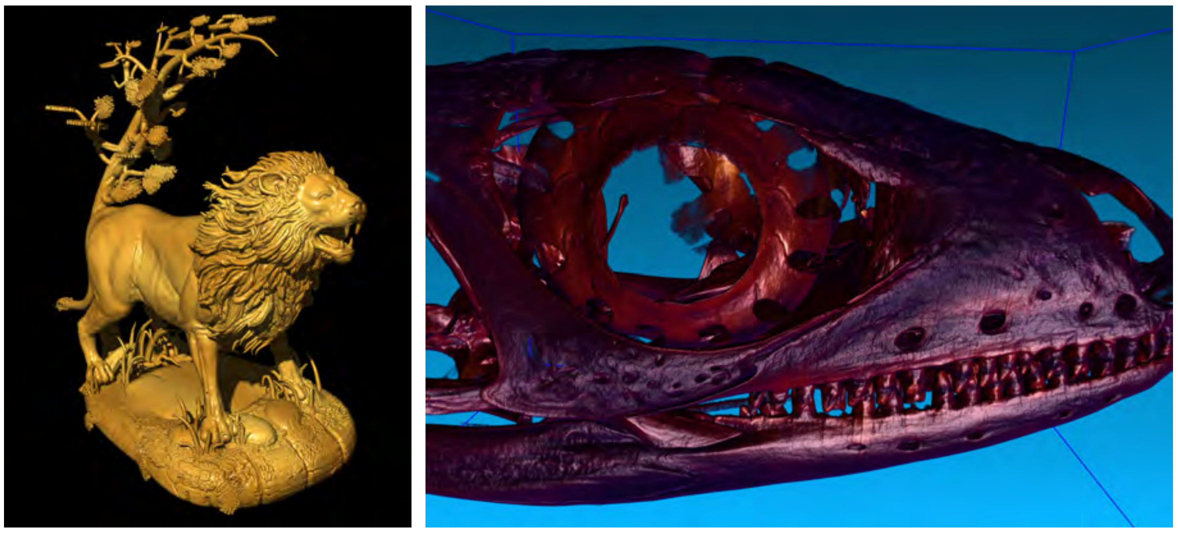

(a) (b)

Figure 7.19. Two examples of high resolution datasets rendered in real-time using GigaVoxels and loaded from system memory using our data loading producer.

7.4.6 GPU procedural and mixed producers

GPU procedural producers directly generate voxel data and the octree structure, either from a mathematical function, or from another representation. Mixed producers can also be implemented, loading data from the system memory, then amplifying them using a procedural function. At the end of this section, we present results obtained with fully procedural producers generating 3D Mandelbrot and Julia fractals, as well as a producer voxelizing a triangle mesh on-the-fly on the GPU and amplifying it by adding details using a Perlin noise function.

Compact localization information

In the case of a loading producer as presented in the previous section, no special localization information (Sec. 7.3.5) is required on the page tables of the two caches since the only information we need is the address in system memory where to load data from.

The situation is a little bit more complicated with procedural Producers. Indeed, there is no relationship between the address of an entry of the page table, which is contained in the data request, and the spatial extent it represents as a node of the octree (position and size). In fact, due to the cache mechanism, the organization of the nodes in the node pool can be arbitrary and has nothing to do with the actual scene. However, for procedural producers, a spatial localization information is required in order to know what portion of space (position and size) needs to be generated. What we need is information about the spatial extent (position and size) of the node that requests data, in order to generate it. This information is stored inside the *localization array* presented in Section 7.3.5.

To be able to provide a compact information about the spatial organization, we rely on a localization array composed of two layers :

- 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 32bit integer. Each 10bits 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 10bit 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 encoding restricts the maximum octree depth to 10 levels. In case more resolution is needed, we rely on three 32bit values.
- Each node also stores a *localization depth value* in form of an 8bit 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 our octree subdivision structure. This allows us to derive exactly what information needs to be loaded or produced by the GPU producer, both in terms of octree nodes and in terms of bricks. Of course, values in these two arrays need to be updated or created when nodes are written in the node cache by the GPU producer.

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. 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.

Examples of procedural and mixed producers

Figure 7.20 shows two images rendered using GigaVoxels and a fractal-based procedural GPU producer. The first image 7.20(a) has been rendered using a 3D Julia procedural function amplified using a Perlin noise that adds holes and detailed textures. The second image 7.20(a) uses a 3D Mandelbulb function [Whi09] that is a special 3D Mandelbrot set. Both are rendered at 70-90FPS on an NVIDIA GTX480, they provide "unlimited" resolution, with only the localization information and floating point precision limiting the maximum depth of the octree.

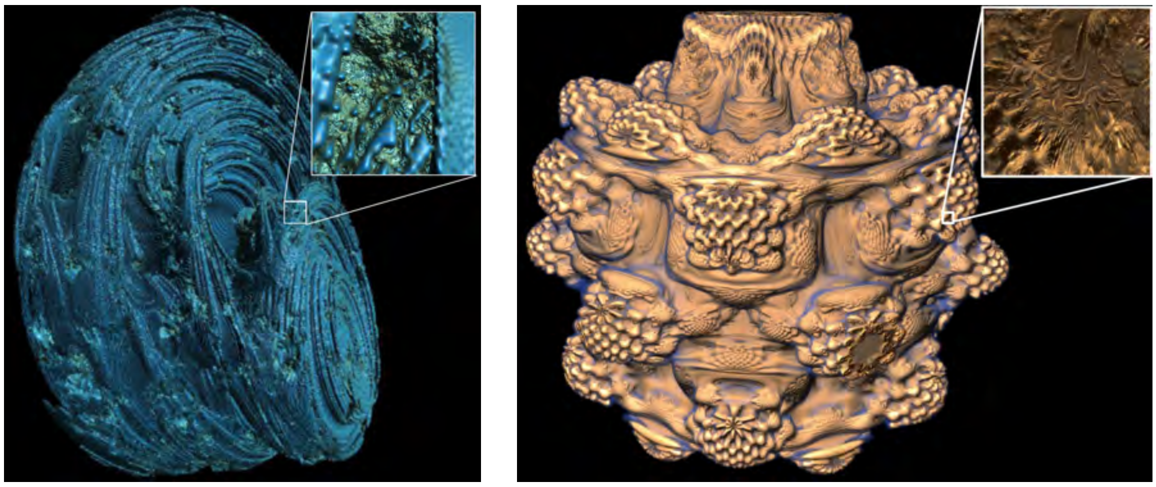

(a) (b)

Figure 7.20. Two examples of highly detailed datasets generated on-the-fly by a fractal-based procedural GPU producer and rendered at 70-90FPS using our approach on a GTX480.

Figure 7.20 shows two images rendered in real-time using mixed GPU producers. Image 7.21(a) has been computed using a voxelization GPU producer that computes a distance field on-demand from of a triangle mesh stored in video memory. It adds a Perlin noise on the voxelized data, with a lookup table used to generate a lava-like material. Image 7.21(b) presents a rendering with GigaVoxels of the 4096^3 Visible Human dataset, enhanced with a Perlin noise providing a Mummy-like look. This dataset represents 256GB on disk (one RGBA8 value per voxel) and is loaded dynamically using our disk loading GPU producer 7.4.4. It is rendered at around 15-20FPS on an NVIDIA GTX 280.

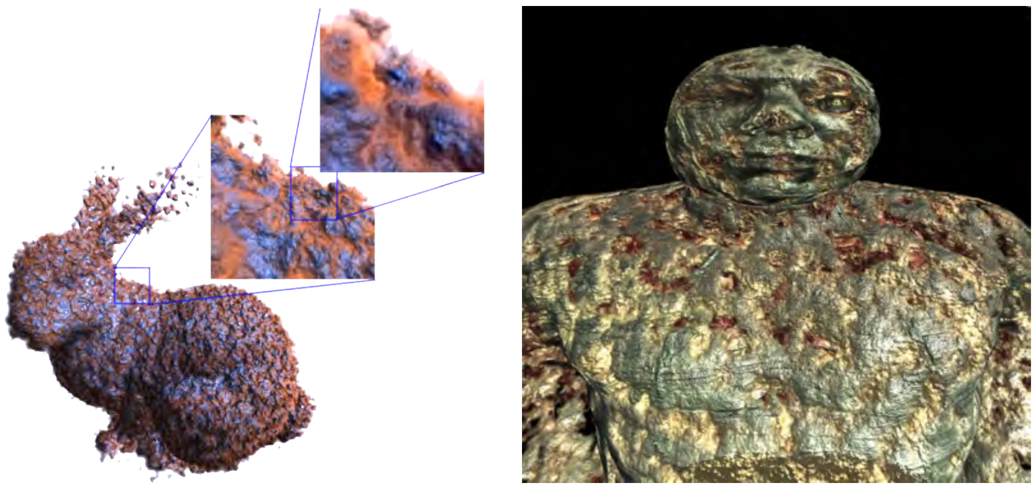

(a) (b)

Figure 7.21. (a) : Triangle mesh voxelized on-the-fly and amplified using a Perlin noise, rendered at 70FPS on a GTX280. (b) : 4096^3 voxels Visible Male dataset, amplified using a Perlin noise and rendered at 15-20FPS.

7.5 Results and performance analysis

In order to evaluate the performance of our out-of-core caching and streaming mechanisms, we studied their behaviors in typical rendering scenarios. All these experiments have been made using an NVIDIA GTX480 GPU and an Intel Core 2 Duo E6850 CPU @3GHz.

7.5.1 Repartition of the costs per frame

The actual performance of our paging system needs to be evaluated on a sequence. Thus, in our first case study presented in Figure 7.22, we analyze the repartition of the costs per frame of the different processing phases, on a typical sequence of exploration (presented in Figure 7.22 top) inside the Mandelbulb scene. This scene is generated by the fully procedural fractal Producer presented in Section 7.4.6. We use the real-time update strategy presented in Section 7.2.2, with only one rendering pass and one update pass per frame. We present the repartition of the costs (in ms), for each frame of the sequence, between the rendering phase (our ray-casting presented in Chapter 6), the data management of the two caches (node and bricks), the loading of the nodes and the loading of the bricks inside the caches. The top graph shows the repartition of the timings when the exploration is done at "normal" speed, during 64 seconds. The bottom graph shows the results for the same sequence explored at 4× the normal speed, during 16 seconds. The first observation is that the rendering time is quite constant all along the sequence. It slightly increases as the camera zooms inside the dataset, most probably due to the deeper traversal of the octree that is induced. On average, the rendering phase represents 56% of the total time of a frame at 1x speed, and 34% at 4x speed. Inside the ray-casting pass, we also measured the cost of interacting with the two cache interfaces (Sec. 7.3.3). On average, emitting data requests and usage information on the two caches only increases the total rendering time by 5%.

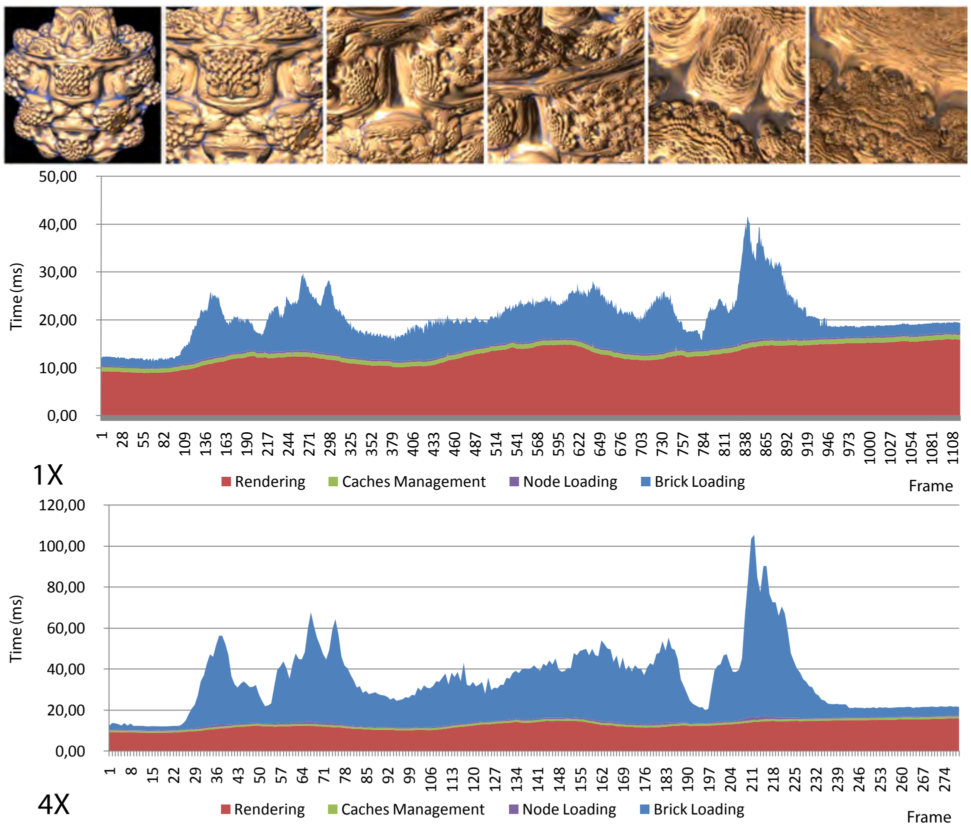


Figure 7.22. Repartition of the costs (in ms) per frame of the main operations computed in our approach, on a 64 second exploration sequence inside the Mandelbulb scene. The top image shows 6 snapshots of the whole sequence, first graph times with normal speed exploration, second graph times with 4x exploration speed.

The cost of the management of the two caches (LRU scheme Sec. 7.3.4 + data requests management Sec. 7.3.5) is totally constant and represents on average 4% of the total frame time at 1x speed and 2.5% at 4x. As we will see in Section 7.5.4, this cost is very low compared to the cost of the CPU management schemes employed in other approaches. The cost of loading nodes inside the node cache is very low, and represents on average 1.5% of the total frame time. However, as expected, the cost of generating bricks that contain actual voxel data and loading them inside the brick cache is very high. It represents 30% of the total frame time on average at 1x speed, and on average dominates the cost of a frame at 4x with 57% of the frame time.

7.5.2 Cache efficiency

In our second case study presented in Figure 7.23, we analyze the efficiency of our cache mechanism in the management of the brick pool. On the two graphs, we show the percentage of data that have been loaded and used inside the brick cache for each frame of the same sequence than the one used in the previous section, with 4x exploration speed. The top graph shows results when using a 512MB cache, while the bottom graph shows the results with a smaller 128MB cache. This allows us to compute the ratio of data used in the cache per frame over the data loaded. This ratio appears to be very good, with only 2.6% of cache misses on average with the 512MB cache, and 5.9% with the 128MB cache.

Since we load data entirely on-demand, without using any pre-loading of data (based on statistical prediction for instance), almost all loaded data are actually used in the subsequent rendering pass and this does not generate trashing. Trashing can only appears when the total amount of data required for a given frame is larger than the size of the cache. In this case, quality reduction strategies have to be employed to ensure real-time rendering.

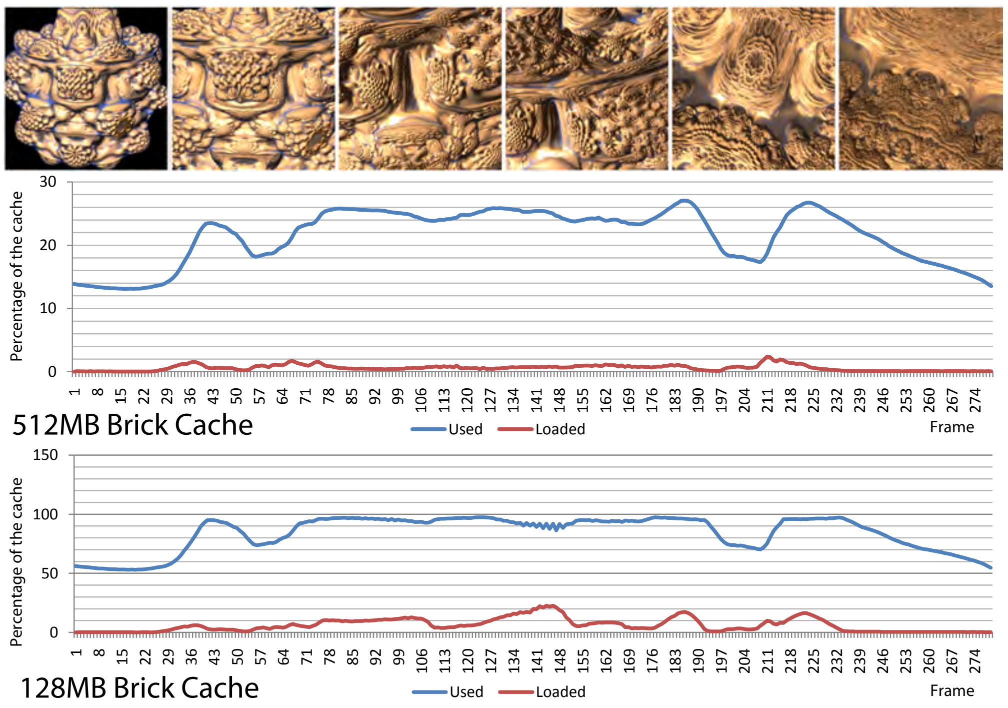
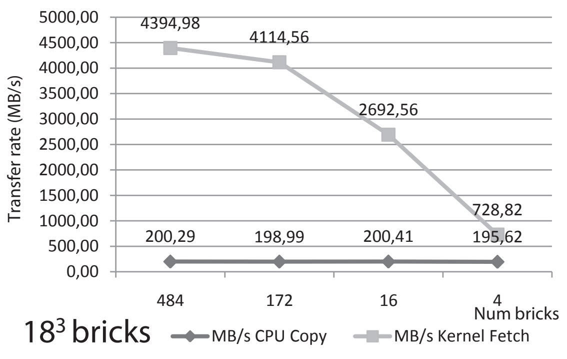


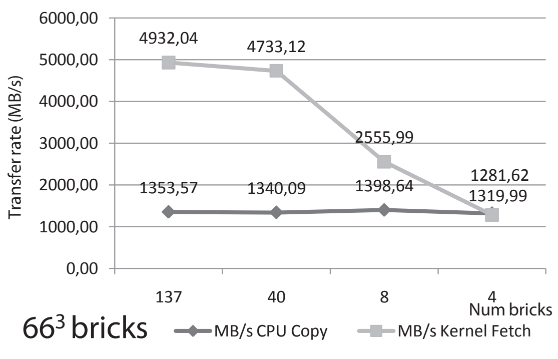
Figure 7.23. Percentages of pages loaded and used in the brick cache for each frame of a 16 second sequence of exploration of the Mandelbulb scene on a GTX480.

7.5.3 Comparison with CPU-based transfers

One of the major differences of our approach compared with previous work is the streaming of the data located in system memory in parallel directly from the GPU using kernel fetches (cf. GPU load Producers presented in Section 7.4.3). Figure 7.24 compares the transfer speed we have been able to achieve using our GPU-based method, with the speed obtained using the CPU-based transfer approach used for instance in [GMAG08]. The CPU-based approach issues multiple copy-to-texture instructions (CUDA *cudaMemcpyToArray* instruction [NVI11a]), one for each brick to transfer. Figure 7.24(a) shows the transfer rate in MB/s of the two approaches, depending on the number of bricks and with 18^3 voxel brick resolution, and Figure 7.24(a) shows the same comparison with 66^3 voxel brick resolution. We clearly see that our approach is increasingly faster than the CPU approach as much as the number of transfered bricks increases. With around 480 bricks of 18^3 voxels and 137 bricks of 66^3 voxels, we reach a little bit more than half the theoretical bandwidth of $8GB/s$ of the PCIExpress bus (cf. Section 1.3), which is a very good performance. However, the CPU approach can not do better than $\frac{1}{40}$ of the theoretical bus bandwidth for 18^3 bricks, and $\frac{1}{5}$ of this bandwidth with 66^3 bricks.



(a)



(b)

Figure 7.24. Comparison of the transfer rates (in MB/s) achieved with CPU-based copies and our GPU-based streaming approach (*kernel fetch*), for 18^3 voxels bricks (a) and 66^3 voxels bricks (b).

7.5.4 Comparison with CPU-based LRU management

Another major difference of our approach compared with previous work is the management of the cache done entirely on the GPU, and in particular the management of the LRU replacement policy. In order to show the advantage of our method in terms of execution speed, we compared the cost of our GPU-based LRU management with a CPU-based management similar to the one used in [GMAG08]. Figure 7.25 shows the average time per frame in milliseconds taken by the two approaches, depending on the size of the managed pool in MB, with 64B pages. We see that our approach performs increasingly better than the CPU-based approach as the size of the cache, and thus the number of pages to manage, increases. In this test we limited the maximum cache size to 64MB and our approach is 1.7x to 27.5x faster than the CPU approach.

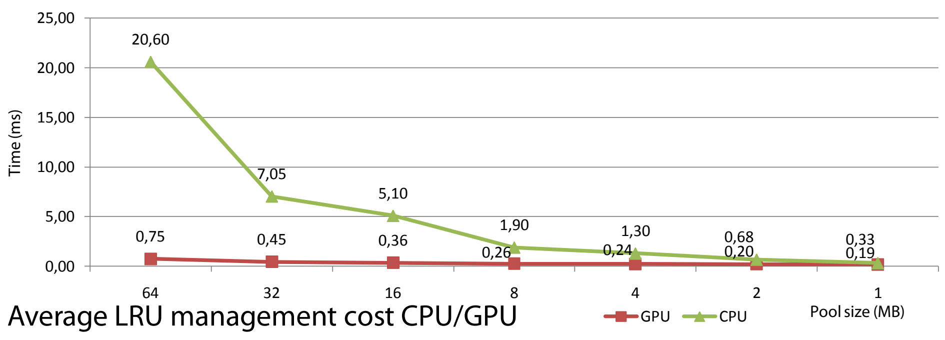


Figure 7.25. Comparison of the average time (in ms) for the management of the LRU replacement policy between our GPU approach and a CPU approach, depending on the size of the cache (in MB) and with 64B pages.

7.5.5 Out-of-core ray-tracing of triangle scenes

In order to demonstrate the genericity of our cache mechanism (Sec. 7.3), we used it to implement an out-of-core renderer for large triangle scenes. This renderer is based on a BVH (Bounding Volume Hierarchy, [LGS+09]) structure, instead of an octree, that stores sets of triangles in its leaves. It is rendered on the GPU using a dedicated ray-tracer that traverses the BVH structure and renders stored triangles. As for GigaVoxels, both the BVH space subdivision structure and the triangle data are managed with our generic cache, inside a *BVH node cache* and a *triangle cache*. Triangles are grouped in pages inside the triangle cache, and each leaf of the BVH can only reference one page of triangles. Both BVH nodes and triangles are streamed using a dedicated GPU Producer from a precomputed structure stored in system memory. This rendering scheme was not particularly optimized and our goal was not to contribute to the domain of out-of-core real-time ray-tracing of large triangle meshes, but simply to demonstrate how our cache mechanism can adapt to different data structures and representations.

Figure 7.26 shows screen-shots taken during the exploration of the 13M triangles Power Plant scene rendered using our dedicated ray-tracer, from the BVH-based representation stored and managed in video memory using our GPU cache. This dataset takes approximatively 512MB stored in system memory and we allocated only 50MB of triangle cache in video memory in order to simulate a constrained memory environment. We get 15-30FPS on an NVIDIA GTX 280 during the exploration of the scene, with the framerate varying per-frame in function of the amount of triangle data loaded inside the cache.

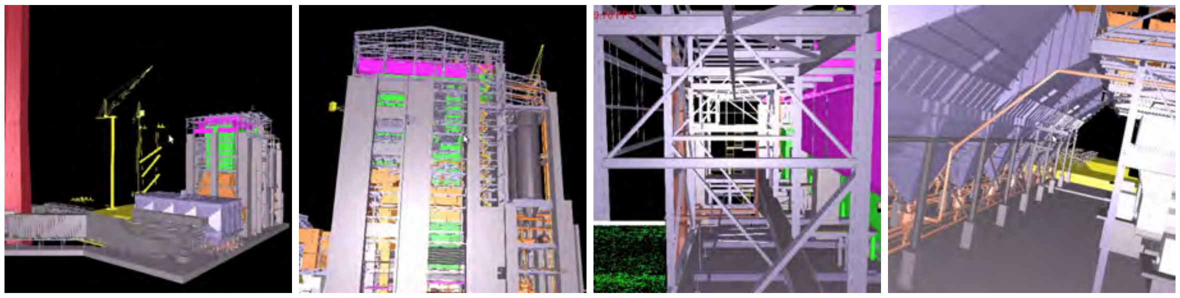


Figure 7.26. Images of the Power Plant 13M triangles model rendered at 15-30FPS using a dedicated ray-tracer and our GPU-based cache mechanism on a GTX280.

Part II

Contributions: Applications of the model

Chapter 8

Direct applications

In this chapter, we present some interesting applications of our method to render specific effects that are difficult to achieve with the classical triangle-based representation. In particular, we show how our hierarchical data structure can be used to synthesize fractal geometry and create virtually infinite resolution procedural scenes in Section 8.1. In Section 8.2, we demonstrate how our caching scheme and ray-based on-demand paging allows a very easy instancing of voxel objects in space, which was difficult to achieve with previous approaches. Finally, in Section 8.3, we demonstrate how our voxel-based pre-integrated cone tracing can be used to very efficiently render blurry effects such as soft-shadows and depth-of-field.

8.1 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 sub-nodes. 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 in Figure 8.1.

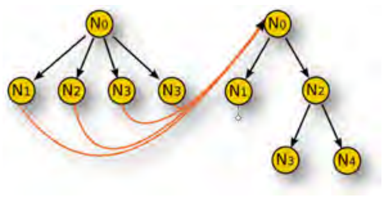


Figure 8.1. Instantiated octree.

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 8.2 shows an example of a Menger sponge fractal. It is implemented using the generalization of our octree, an N^3-tree node with $N = 3$ (see Section 5.2). These two usages, instancing and recursivity, are also made available by our GPU cache mechanism that correctly manages such cases.

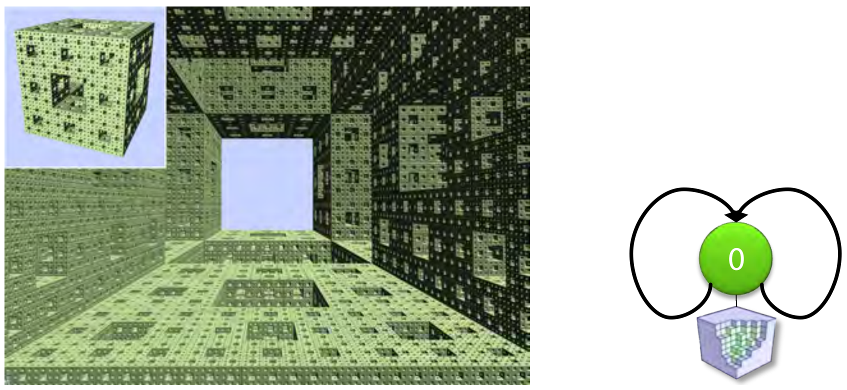


Figure 8.2. Example of a Sierpinski sponge fractal fully implemented with recursivity in the octree. This example is running around 70FPS. The bottom graph shows the only node recursively linked used in this case.

8.2 Voxel object instancing

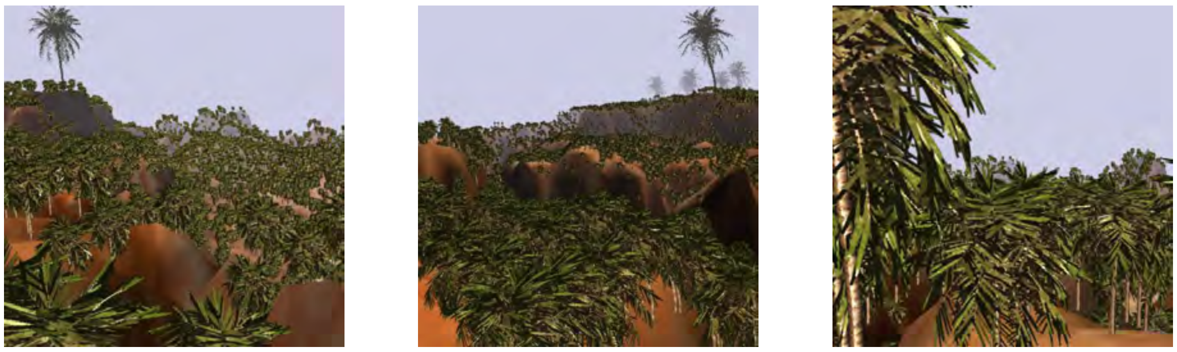


Figure 8.3. Example of instancing of thousands of voxel trees in a forest scene rendered at 20-25FPS on a GTX280.

Our ray-tracing framework is compatible with several voxel entities, or objects, present in the scene. This can be either multiple different objects, each with its own octree structure and data producer and stored in the same caches in video memory as we have seen in Section 7.4.1, or this can be the same octree structure "instantiated" (ie. *rendered*) multiple times at different positions in the scene.

Such instantiation of the same octree structure in a scene is made possible by the autonomous traversal of the structure by our ray-tracing algorithm (Sec. 6). During the rendering of each instance, the required parts of the structure will be requested to the caches, depending on the distance to the viewer and potential occlusions. Scaling, rotation and LODs are automatically handled and allow us to represent scenes with many complex objects at high framerates. Figures 8.3 and 8.4 show two examples of such scenes.

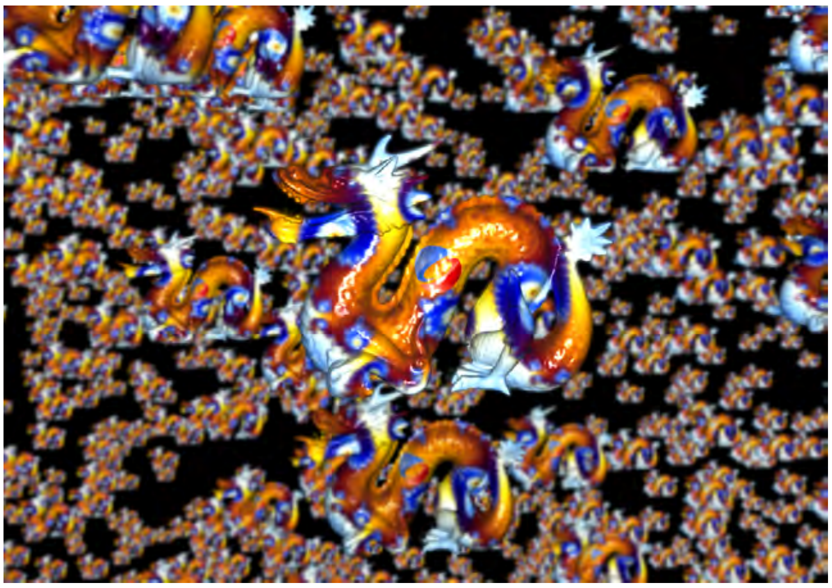


Figure 8.4. Examples of free instancing of multiple GigaVoxels objects in space.

8.3 MipMap-based blur effects

Throughout this thesis we have shown that our hierarchical structure as well as our GPU paging mechanism are key elements to enable the processing of large data volumes, and thus to efficiently render voxel-based pre-filtered geometrical representations.

As we have seen in Chapter 4, this pre-filtered (and pre-integrated) geometry representation allows us to efficiently trace approximate cones using only one single ray in order to produce anti-aliased rendering. Beyond simple anti-aliased rendering, cone tracing has many interesting applications. Even though our solution is approximate and is based on some hypotheses on our scenes (cf. Section 4.4), it provides a very fast way to estimate the total incoming energy scattered by the objects of a scene and coming towards an arbitrary cone. While this approximation is not always precise, especially with large cones, it has the great advantage of always providing smooth results, while multisampling schemes are knows to generate noisy results.

In this section, we detail how our voxel cone tracing can be used to quickly estimate soft shadows as well as depth-of-field effects. Rendering such effects is very challenging with triangular models. More generally with B-reps, rendering blurry effects happens to be more costly in terms of computation than doing sharp rendering, due to the multisampling scheme that needs to be employed. Interestingly with our approach, rendering blurry effects uses lower resolution volume data (thanks to the LOD), and thus happens to be faster than sharp rendering.

8.3.1 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.

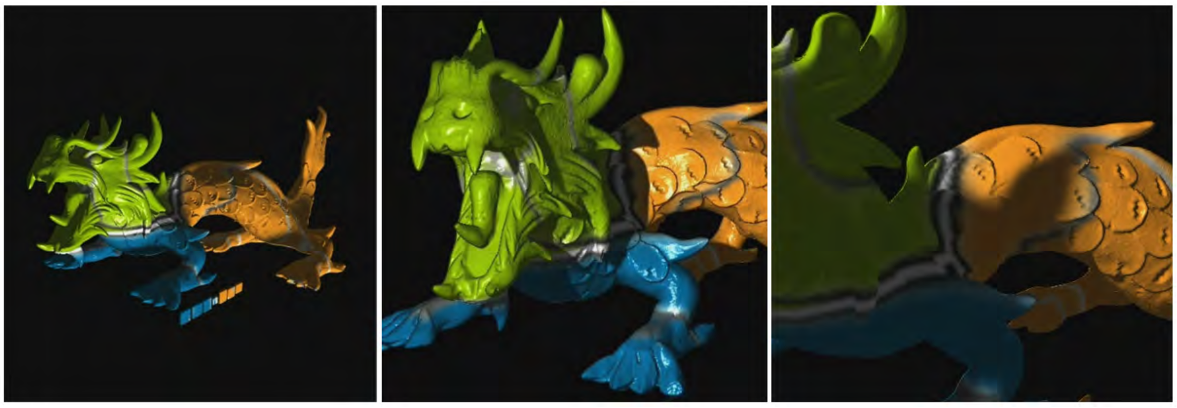


Figure 8.5. Example of soft shadows rendered by launching secondary rays and using the volume MIP-mapping mechanism to approximate integration over the light source surface. Interestingly, the blurrier the shadows, the cheaper they are to compute.

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 amounts to shooting a ray from P towards the light source. If the opacity value of the ray saturates on the way to the light, 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 for primary-rays inspired by cone tracing, we obtain an *impact volume* at the intersection between the cone emitted from the eye and the object. To take this into account, we should not shoot a simple ray toward the light, but again an entire cone. This *light cone*'s apex will lie on the point light source itself and its radius is defined by the size of the impact volume (see Figure 8.6).

To sample the light cone, we perform a similar traversal as for the view, following our voxel cone tracing model presented in Section 4.2. Only this time, the LOD 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.

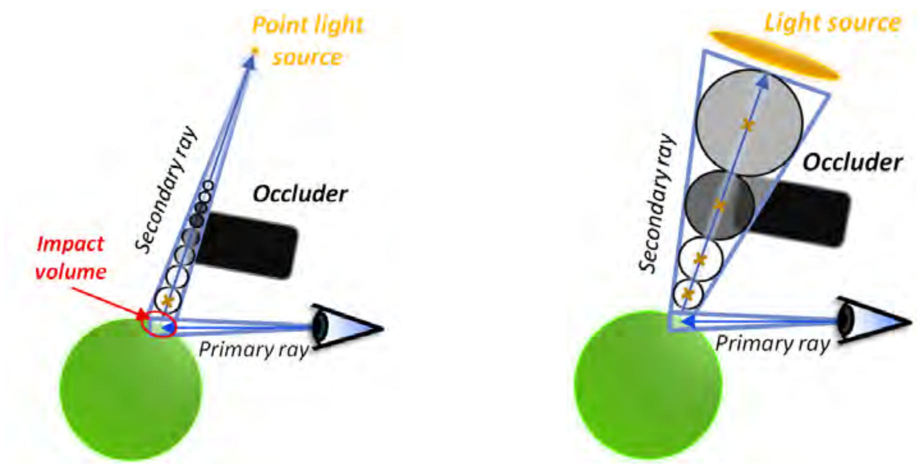


Figure 8.6. Left: illustration of shadow computation for a point light source, taking impact volume into account. Right: soft shadows computation for a surface light source.

To approximate soft shadows, we compute a *filtered* 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 8.6). Again, we accumulate the volume values. The resulting value reflects how much the light reaching V is occluded.

This coarse approximation is extremely efficient, delivers pretty-good shadows and is fully compatible with our cache mechanism presented in Section 7.

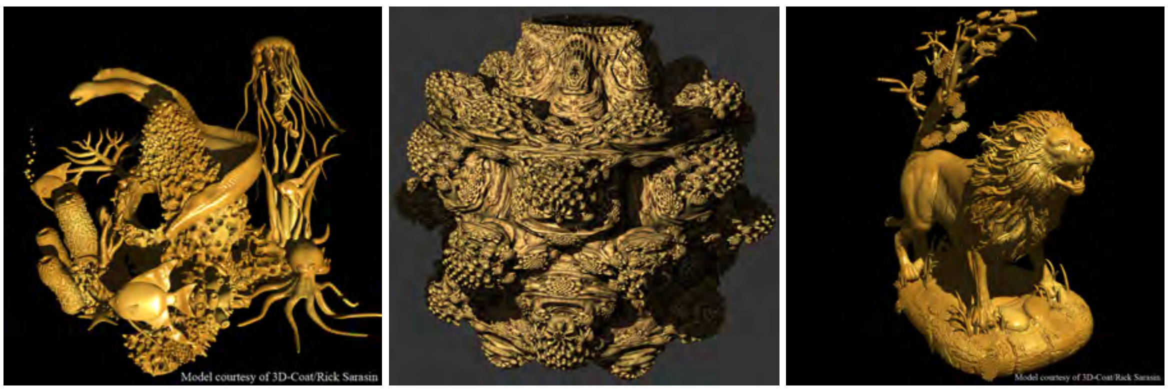

Model courtesy of 3D-Coat/Rick Sarasin
Model courtesy of 3D-Coat/Rick Sarasin

Figure 8.7. Examples of high resolution voxel objects efficiently rendered in real-time with soft shadows using our approach.

8.3.2 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 the 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 8.9 this set of rays can again be grouped in some form of double cone, the *lens cone*. This double-cone defines the LOD that should be used along the rays launched for all pixels of the screen, in order to approximate this integral over the camera lens.

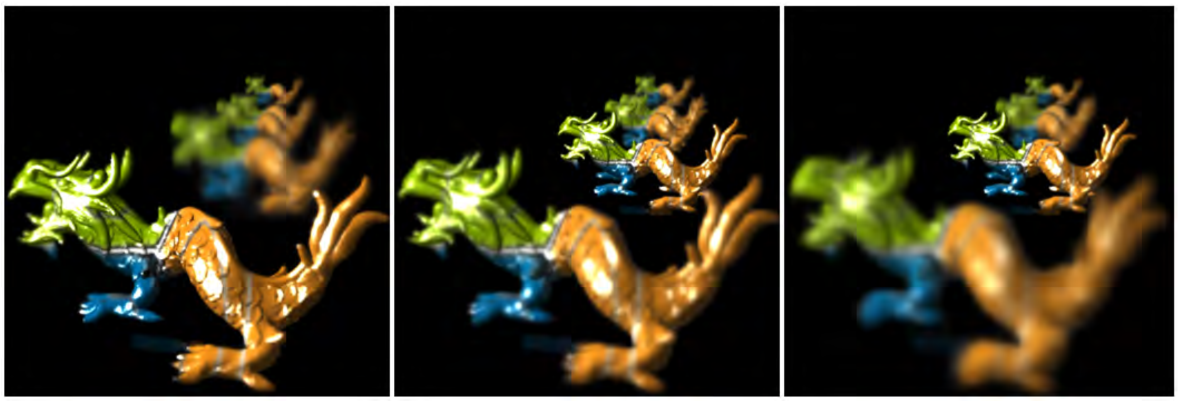


Figure 8.8. Example of Depth-Of-Field rendering with GigaVoxels thanks to the volume MIP-mapping. Once again, the blurrier are the objects and the cheaper it is to render.

Paradoxically with our approach, 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 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 necessary to produce the final image. 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. In addition, these secondary rays are perfectly handled by our caching mechanism, thanks to our per-ray data request mechanism.

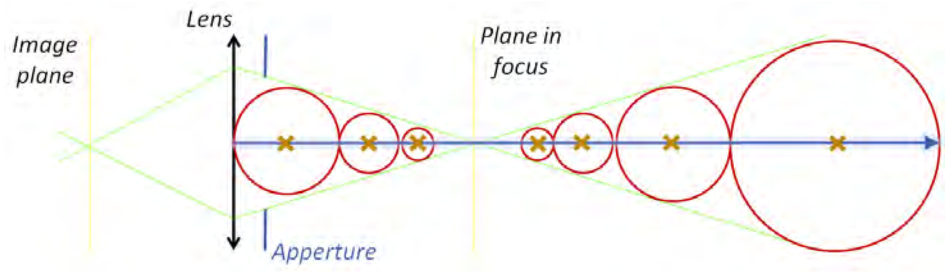


Figure 8.9. Illustration of the cone tracing used to approximate Depth-Of-Field effect with a single ray.

Chapter 9

Interactive Indirect Illumination Using Voxel Cone Tracing

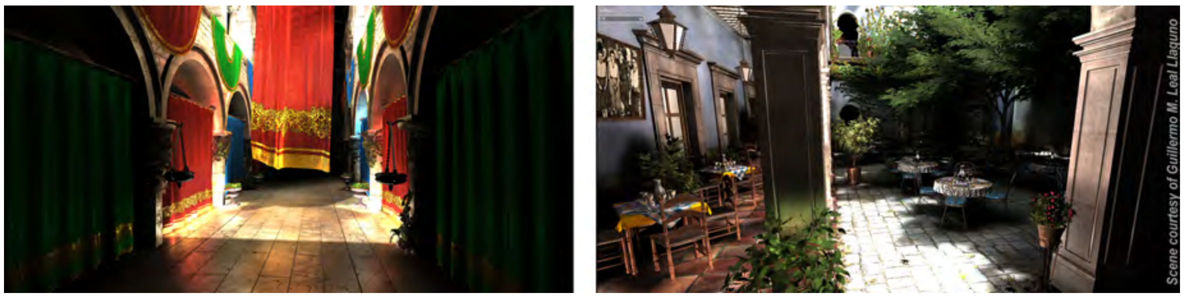


Figure 9.1. Real-time indirect illumination (25-70 fps on a GTX480): Our approach supports diffuse and glossy light bounces on complex scenes. We rely on a voxel-based hierarchical structure to ensure efficient integration of 2-bounce illumination. (Right scene courtesy of G. M. Leal Llaguno).

Indirect illumination is an important element of realistic image synthesis, but its computation is expensive and highly dependent on the complexity of the scene and of the BRDF of the surfaces involved. While off-line computation and pre-baking can be acceptable for some cases, many applications (games, simulators, etc.) require real-time or interactive approaches to evaluate indirect illumination.

We present a novel algorithm to compute indirect lighting in real-time that avoids costly precomputation steps and is not restricted to low frequency illumination. It is based on our hierarchical voxel octree representation generated and updated on-the-fly from a regular scene mesh coupled with our approximate voxel cone tracing (Chap. 4) that allows a fast estimation of the visibility and incoming energy. Our approach can manage two light bounces for both Lambertian and Glossy materials at interactive frame rates (25-70FPS). It exhibits an almost scene-independent performance thanks to an interactive octree voxelization scheme, hereby allowing for complex scenes and dynamic content.

In contrast to the rendering usages we presented previously in this thesis, in this application our voxel representation is not used to render primary rays, but instead is employed as a proxy to compute indirect illumination. The approach presented in this chapter does not rely on our out-of-core scheme presented in Chapter 7, but we see its integration as an interesting direction for future work, as it could highly benefit from our ray-based demand paging scheme.

9.1 Introduction

There is no doubt that indirect illumination drastically improves the realism of a rendered scene, but generally comes at a significant cost because complex scenes are challenging to illuminate, especially in the presence of glossy reflections. Global illumination is computationally expensive for several reasons. It requires computing visibility between arbitrary points in the 3D scene, which is difficult with rasterization based rendering. Second, it requires integrating lighting information over a large number of directions for each shaded point. Nowadays, with complexity of the rendering content approaching millions of triangles, even in games, computing indirect illumination in real-time on such scenes is a major challenge with high industrial impact. Due to real-time constraints, off-line algorithms used by the special-effect industry are not suitable, and dedicated fast approximate and adaptive solutions are required. Relying on precomputed illumination is very limiting because common effects such as dynamic light sources and glossy materials are rarely handled.

In this chapter, we present a novel algorithm that avoids costly precomputation steps, and is not restricted to low frequency illumination. It exhibits an almost scene-independent performance and is suitable to be extended for out-of-core rendering, therefore allowing for arbitrarily complex scenes. We avoid using the actual scene geometric mesh and can achieve indirect illumination in arbitrary scenes at an almost geometry-independent cost. We reach real-time frame rates even for highly detailed environments and produce plausible indirect illumination (see Teaser).

The core of our approach is built upon our pre-filtered hierarchical voxel representation of the scene geometry presented in chapter 4. For efficiency, this representation is stored on the GPU in the form of a dynamic sparse voxel octree [CNLE09, LK10] (Chap. 5) generated from the triangle meshes. We handle fully dynamic scenes, thanks to a new real-time mesh voxelization and octree building and filtering algorithm that efficiently exploits the GPU rasterization pipeline (Sec. 9.4.2). This octree representation is built once for the static part of the scene, and is then updated interactively with moving objects or dynamic modifications on the environment (like breaking a wall or opening a door).

We rely on this pre-filtered representation to quickly estimate visibility and integrate incoming indirect energy splatted in the structure from the light sources using a new approximate voxel cone tracing. The main contributions of our work are the following:

- A real-time algorithm for indirect illumination;
- An adaptive scene representation independent of the mesh complexity;
- An efficient splatting scheme to inject and filter incoming radiance information (energy + direction) into our voxel structure;
- A fast GPU-based mesh voxelisation and octree building algorithm.

9.2 Previous Work

There are well established off-line solutions for accurate global-illumination computation such as path tracing [Kaj86], or photon mapping [Jen01]. These have been extended with optimizations that often exploit geometric simplifications [TL04, CB04b], or hierarchical scene structures, but do not achieve real-time performance. Fast, but memory-intensive relighting for static scenes is possible [LZT+08], but involves a slow preprocessing step. Anti-radiance [DSDD07, DKTS07] allows us to deal with visibility indirectly, by shooting negative light, and reaches interactive rates for a few thousand triangles.

To achieve higher frame rates, the light transport is often discretized. Particularly, the concept of VPLs [Kel97] is interesting, where the bounced direct light is computed via a set of *virtual point* *lights*. For each such VPL, a shadow map is computed, which is often costly. Laine et al. [LSK$^+$07] proposed to reuse the shadow maps in static scenes. While this approach is very elegant, fast light movement and complex scene geometry can affect the reuse ratio. Walter et al. [WFA$^+$05] use lightcuts to cluster VPLs hierarchically for each pixel, while Hasan et al. [HPB07] push this idea further to include coarsely sampled visibility relationships. In both cases, a costly computation of shadow maps cannot be avoided and does not result in real-time performance. ISM [RGK$^+$08] and Microrendering [REG$^+$09] reach real-time performance by using a point-based scene approximation to accelerate the rendering into the VPL frusta, but cannot easily ensure sufficient precision for nearby geometry. Our approach represents direct illumination hierarchically and uses cone tracing that replaces shadow map computations to accelerate the image generation. The most efficient real-time solutions available today work in the image-space of the current view, but ignore off-screen information [NSW09]. Our approach is less efficient than such solutions, but does not require similarly strong approximations. In particular, we achieve high precision near the viewer which is important for good surface perception [AFO05].

Our work derives a hierarchical representation of the scene that produces a regular structure to facilitate light transfer and achieve real-time performance, similar in spirit to Kaplanyan et al. [KD10], who implement diffuse indirect illumination using a diffusion process in a set of nested regular voxel grids. While relatively fast, this approach suffers from a lack of precision coming from the relatively low resolution of the voxel grids that can be used. This resolution is limited by the cost of the diffusion process as well as the memory occupancy. Instead of relying on a diffusion process, our approach relies on a ray-tracing approach to collect the radiance stored in the structure. This approach allows us to use a sparse storage of the incoming radiance and scene occlusion information, while maintaining high precision. This lack of precision limits Kaplanyan's approach to diffuse indirect illumination, while our approach can manage both diffuse and specular indirect lighting.

9.3 Algorithm overview

Our approach is based on a three-step algorithm as detailed in Figure 9.2. We first inject incoming radiance (energy and direction) from dynamic light sources into the leaves of the sparse voxel octree hierarchy. This is done by rasterizing the scene from all light sources and splatting a photon for each visible surface fragment. In a second step, we filter the incoming radiance values in the higher levels of the octree (mipmap). We rely on a compact Gaussian-Lobe representation to store the filtered distribution of incoming light directions. This is done efficiently in parallel by relying on screen-space quad-tree analysis. Our voxel filtering scheme also treats the NDF and the BRDF in a view-dependent way. Finally, we render the scene from the camera. For each visible surface fragment, we combine the direct and indirect illumination. We employ an approximate cone tracing to perform a final gathering [Jen96], sending out a few cones over the hemisphere to collect illumination distributed in the octree. Typically for Phong-like BRDF, a few large cones (~5) estimate the diffuse energy coming from the scene, while a tight cone in the reflected direction with respect to the viewpoint captures the specular component. The aperture of the specular cone is derived from the specular exponent of the material, allowing us to efficiently compute glossy reflections.

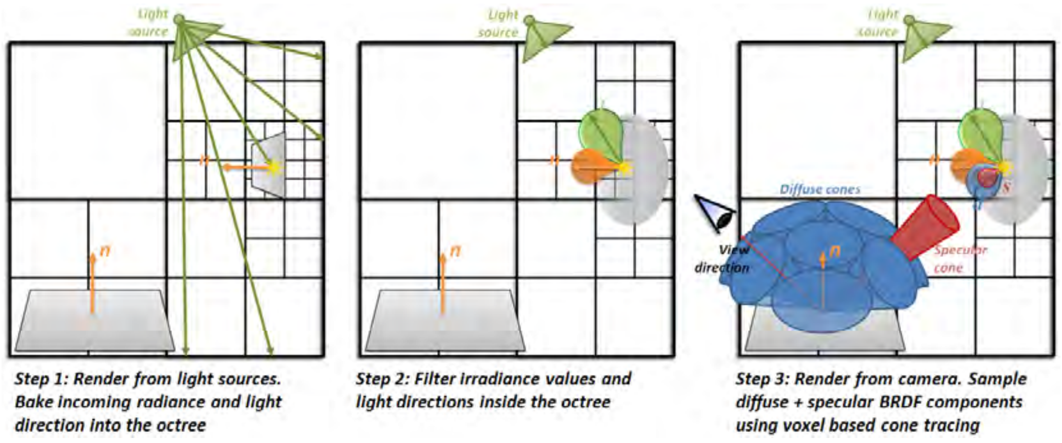

Light source
n
Step 1: Render from light sources. Bake incoming radiance and light direction into the octree
Light source
n
Step 2: Filter irradiance values and light directions inside the octree
Light source
Diffuse cones
View direction
n
Specular cone
Step 3: Render from camera. Sample diffuse + specular BRDF components using voxel based cone tracing

Figure 9.2. *Left:* Illustration of the three steps of our real-time indirect lighting algorithm. *Right:* Display of the sparse voxel octree structure storing geometry and direct lighting information.

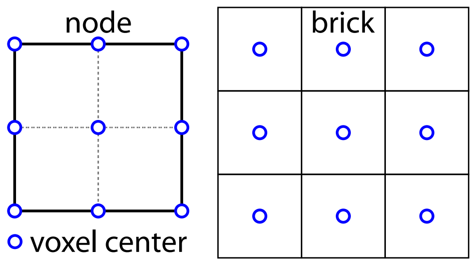
9.4 Our hierarchical voxel structure

The core of our approach is built around our pre-filtered hierarchical voxel version of the scene geometry we described in Chapter 4. For efficiency, this representation is stored in the form of a sparse voxel octree as described in Chapter 5.

Having a hierarchical structure, allows us to avoid using the actual scene geometric mesh and can achieve indirect illumination in arbitrary scenes at an almost geometry-independent cost. It is possible to improve precision near the observer and to abstract energy and occupancy information farther away. We reach real-time frame rates even for highly detailed environments and produce plausible indirect illumination. We can choose a scene resolution suitable to the viewing and lighting configuration, without missing information like light undersampling or geometric LOD would. Thus, our approach always ensure smooth result, in contrary to path-tracing or photon mapping approaches [Jen96].

9.4.1 Structure description

Our sparse voxel octree is a very compact pointer-based structure with assciated bricks described in Chapter 5. Octree nodes are stored in linear GPU memory and nodes are grouped into $2 \times 2 \times 2$ tiles. Since we use a brick instead of a single value per node we can use hardware texture trilinear interpolation to interpolate values. This structure allows us to query filtered scene information (energy intensity and direction, occlusion, local normal distribution functions - NDFs) with increasing precision by traversing the tree hierarchy. This property will allow us to achieve adaptivity and handle large and complex scenes.



As we have seen in Chapters 5 and 6, using small bricks in the sparse octree is more efficient both in terms of storage and in terms of rendering speed. Thus, we rely on the corner-centered voxel localization configuration detailed in Section 5.1.4 with $3 \times 3 \times 3$ voxel bricks. We assume that the voxel centers are located at the node corners and not at the node centers (Fig. left). This ensure that interpolated values can always be computed inside a brick covering a set of $2 \times 2 \times 2$ nodes.

The only difference with the structure presented in Chapter 5 is that we add neighbor pointers to the nodes of our structure. These will enable us to quickly visit spatially neighboring nodes during the interactive voxelization of dynamic objects. We will see that these links are particularly important to efficiently distribute the direct illumination over all levels of the tree.

9.4.2 Interactive voxel hierarchy construction

Our sparse hierarchical voxel structure will replace the actual scene in our light transport computations. The voxel data representation (see above) allows interpolation and filtering computations to be simple.

In order to quickly voxelize an arbitrary triangle-based scene, we propose a new real-time voxelization approach that efficiently exploits the GPU rasterization pipeline in order to build our sparse octree structure and filter geometrical information inside it. This voxelization must be fast enough to be performed at each rendering frame, allowing for fully dynamic scenes. In order to scale to very-large scenes, our approach avoids relying on an intermediate full regular grid to build the structure and instead directly constructs the octree. To speed-up this process, we observed that in real video-game situations, large parts of the environment are usually static or updated punctually on user-interaction. This allows us to voxelize these parts once in the octree, and to update them only when necessary, while full dynamic objects are re-voxelized at each frame. Both semi-static and fully dynamic objects are stored in the same octree structure, allowing an easy traversal and a correct filtering of both types of objects together. A timestamp mechanism is used to differentiate both types, in order to prevent semi-static parts of the scene to get destructed at each frame. Our structure construction algorithm performs in two steps: octree building and MIP-mapping of the values.

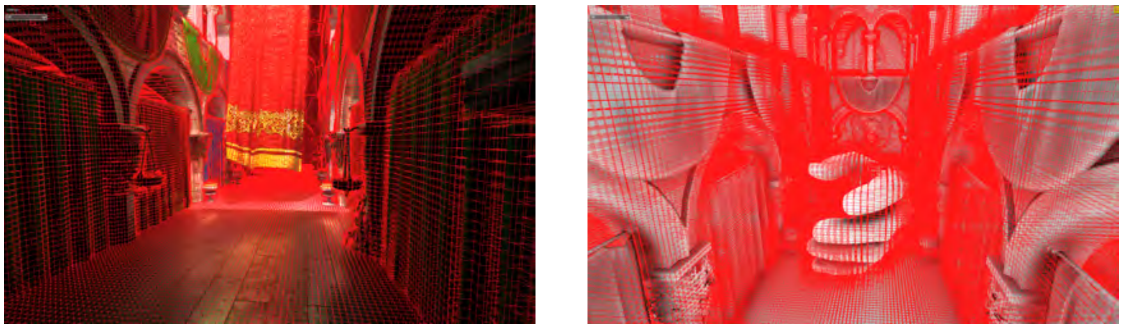


Figure 9.3. *Left:* Display of the sparse voxel octree structure storing filtered geometry and direct lighting information for a static environment. *Right:* Display of the octree with both the static environment and a dynamic object with the structure updated in real-time.

Octree building

We first create the octree structure itself by using the GPU rasterization pipeline. To do so, we rasterize the mesh three times, along the three main axes of the scene, with a viewport resolution corresponding to the resolution of the maximum level of subdivision of the octree (typically 512×512 pixels for a 512^3 octree). By disabling the depth test to prevent-early culling, this allows us to generate at least one fragment shader thread for each part of the surface that will fall into a given leaf of the tree (and needs to write surface attributes, typically texture color, normal and material information). These threads traverse the octree from top-to-bottom and directly subdivide it when needed during the traversal, in order to reach the correct leaf where they can write their values.

Whenever a node needs to be subdivided, a set of $2 \times 2 \times 2$ sub-nodes is "allocated" inside a global shared *node buffer* pre-allocated in video memory. The address of this set of new sub-nodes is written inside the "child" pointer of the subdivided node and the thread continue its descent down to the leaf, creating each successive new required set of nodes. These allocations are made through the atomic increment of a global shared counter indicating the next available page of nodes in the shared node buffer. Since we are in a massively parallel environment, multiple threads can request the subdivision of the same node at the same time, and so could generate an incoherent structure. To prevent such a conflict, we rely on a per-node mutex that makes it possible to perform the subdivision only in the first thread getting it. Unfortunately, it is not possible to put the other threads to sleep while waiting for the first thread to finish the subdivision.

In order to avoid an active waiting loop that would be too expensive, we implemented a *global thread list* where interrupted threads put themselves for a deferred execution. At the end of the rasterization pass, deferred threads are re-run (in a vertex shader) to allow them to write their values in the tree. Such a deferred pass can possibly generate new deferred threads and is re-executed as long as the global thread list is not empty. Values in the leaves are written directly inside the brick associated with the nodes, and bricks are allocated similarly to the nodes inside a *shared brick buffer*.

In our OpenGL implementation this scheme is made possible by the new `NV_shader_buffer_load` and `NV_shader_buffer_store` extensions that provide CUDA-like video memory pointers as well as atomic operations directly inside OpenGL shaders.

MIP-mapping

Once the octree structure is built and the surface values written inside the leaves, these values must be MIP-mapped and filtered in the inner nodes of the tree (following our approach described in chapter 4). This is simply done in $n - 1$ steps for an octree of n levels. At each step, threads compute the filtered values coming from the bricks in the sub nodes of each node of the current level. To compute a new filtered octree level, the algorithm averages values from the previous level. Since we rely on vertex-centered voxels as described in Section 9.4.1, each node contains a 3^3-voxel brick, whose boundaries reappear in neighboring bricks. Consequently, when computing the filtered data, one has to weight each voxel with the inverse of its multiplicity (Fig. left). In practice, this results in a 3^3-Gaussian weighting kernel which, for our case, is an optimal reconstruction filter [FP02a].

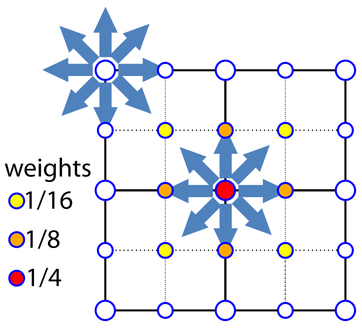


Figure 9.4. Lower-level voxels surround higher-level voxels. During filtering, shared voxels are "evenly distributed" resulting in Gaussian weights.

9.4.3 Voxel representation

Each voxel at a given LOD must represent the light behavior of the lower levels - and thus, of the whole scene span it represents. For this, we rely entirely on our pre-filtered geometry model described in Chapter 4 to model the directional information with distributions that describe the underlying geometry. We add the filtered incoming radiance to this model, which will be injected into the representation by all direct light sources. We also store and filter the distribution of all incoming light directions, in order to allow the computation of indirect specularities.

Since storing arbitrary distributions would be too memory-intensive, we choose to store only isotropic Gaussian lobes characterized by an average vector D and a standard deviation σ (as described in Section 4.5). Following [Tok05], to ease the interpolation, the variance is encoded via the norm $|D|$ such that $\sigma^2 = \frac{1-|D|}{|D|}$. In Section 9.6, we will detail how to calculate light interaction with such a data representation.

9.5 Ambient Occlusion

To illustrate the use of our approximate cone tracing (Sec. 4.2 in this context and to facilitate the understanding of our indirect illumination algorithm, we will first present a simpler case: an ambient occlusion estimation (AO), which can be seen as an accessibility value [Mil94]. The ambient occlusion $A(p)$ at a surface point p is defined as the visibility integral over the hemisphere Ω (above the surface) with respect to the projected solid angle. Precisely, $A(p) = \frac{1}{\pi}\int_{\Omega} V(p,\omega)(cos\omega)d\omega$, where $V(p,\omega)$ is the visibility function that is zero if the ray originating at s in direction ω intersects the scene, otherwise it is one. For practical uses (typically, indoor scenes which have no open sky) the visibility is limited to a distance since the environment walls play the role of ambient diffusors. Hence, we weight occlusion α by a function $f(r)$ which decays with the distance (in our implementation we use $\frac{1}{(1+\lambda r)}$). The modified occlusion is $\alpha_f(p + r\omega) := f(r)\alpha(p + r\vec{\omega})$.

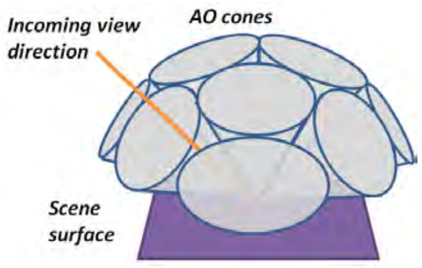


Figure 9.5. Ambient Occlusion is computed via a set of cones launched over the hemisphere associated with a given surface.

To compute the integral $A(p)$ efficiently, we observe that the hemisphere can be partitioned into a sum of integrals: $A(p) = \frac{1}{N}\sum_{i=1}^{N} Vc(p,\Omega_i)$, where $Vc(p,\Omega_i) = \int_{\Omega_i} V_{p,\theta}(cos\theta)d\theta$. For a regular partition, each $Vc(p,\Omega_i)$ resembles a cone. If we factor the cosine out of the Vc integral (the approximation is coarse mainly for large or grazing cones). We can then approximate their contribution with our voxel-based cone tracing, as illustrated in Figure 9.5. The weighted visibility integral $V(p,\omega)$ is obtained by accumulating the occlusion information only, accounting for the weight $f(r)$. Summing up the contributions of all cones results in our approximation of the AO term.

Final Rendering

To perform the rendering of a scene mesh with AO effects, we evaluate the cone tracing approximation in the fragment shader. For efficiency, we make use of deferred shading [ST90] to avoid evaluating the computation for hidden geometry. *I.e.,* we render the world position and surface normal pixels of an image from the current point of view. The AO computation is then executed on each pixel, using the underlying normal and position.

9.6 Voxel Shading

For indirect illumination, we will be interested not only in occlusion, but need to compute the shading of a voxel. For this shading, we rely on the pre-filtered representation we described in Section 4.5. The difference in our case is that we need to add an incoming radiance information to this pre-filtered representation. This incoming radiance will be injected in the structure from the light sources, and collected during rendering using our approximate cone tracing (cf. Section 9.7).

As shown in [Fou92b, HSRG07], shading computations can conveniently be translated into convolutions, provided that the elements are decomposed into lobe shapes. In our case, we have to convolve the BRDF, the NDF, the span of the view cone as well as the incoming light directions, all except the BRDF already being represented as Gaussian lobes in our structure. We consider the Phong BRDF, *i.e.,* a large diffuse lobe and a specular lobe which can be expressed as Gaussian lobes. Nonetheless, our lighting scheme could be easily extended to any lobe-mixture BRDF.

9.7 Indirect Illumination

To compute indirect illumination in the presence of a point light is more involved than AO. We use a two-step approach.

First, we capture the incoming radiance from a light source in the leaves of our scene representation. Storing incoming radiance, not outgoing, will allow us to simulate glossy surfaces. We filter and distribute the incoming radiance over all levels of our octree.

Finally, we perform approximate cone tracing to simulate the light transport. Writing the incoming radiance in the octree structure is complex, therefore we will, for the moment, assume that it is already present in our octree structure, before detailing this process.

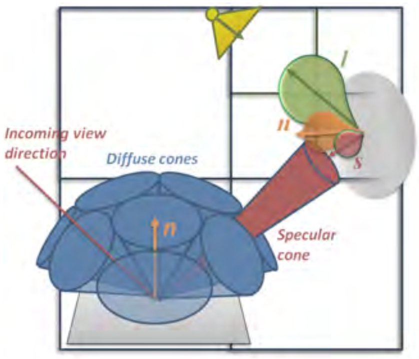


Figure 9.6. We determine indirect lighting via a set of cones, with shading computed with our distribution model.

9.7.1 Two-bounce indirect illumination

Our solution works for low energy - low frequency and high energy - high frequency components of arbitrary material BRDFs, although we will focus our description on a Phong BRDF. The algorithm is similar to the one described in Section 9.5 for AO. We use deferred shading to determine for which surface points we need to compute the indirect illumination. At each such location, we perform a final gathering by sending out several cones to query the illumination that is distributed in the octree. Typically, for a Phong material (Fig. 9.6, right), a few large cones (typically five) estimate the diffuse energy coming from the scene, while a tight cone in the reflected direction with respect to the viewpoint captures the specular component. The aperture of the specular cone is derived from the specular exponent of the material, allowing us to compute glossy reflections.

9.7.2 Capturing direct Illumination

To complete our indirect-illumination algorithm, we finally need to describe how to store incoming radiance. Our approach is inspired by reflective shadow maps [DS05]. We render the scene from the light's view using standard rasterization, but output a world position. Basically, each pixel represents a photon that we want to bounce in the scene. We call this map the *light-view map*. In the following, we want to store these photons in the octree representation. Precisely, we want to store them as a direction distribution and an energy proportional to the subtended solid angle of the pixel as seen from the light. Because the light-view map's resolution is usually higher than the lowest level of the voxel grid, we can assume that we can splat photons directly into leaf nodes of our octree without introducing gaps. Furthermore, photons can always be placed at the finest level of our voxel structure because they are stored at the surface, and we only collapsed empty voxels to produce our sparse representation. To splat the photons, we basically use a fragment shader with one thread per light-view-map pixel. Because several photons might end up in the same voxel, we need to rely on an atomic add.

Although the process sounds simple, it is more involved than one might think and there are two hurdles to overcome. The first problem is that atomic add operations are currently only available for integer textures. We can easily address this issue by using a 16bit normalized texture format, that is denormalized when accessing the value later. The second difficulty is that voxels are repeated for adjacent bricks. We mentioned in Section 9.4 that this redundancy is necessary for fast hardware-supported filtering. While thread collisions are rare for the initial splat, we found that copying the photon directly to all required locations in adjacent bricks leads to many collisions that affect performance significantly. This parallel random scattering, further, results in bandwidth issues. A more efficient transfer scheme is needed.

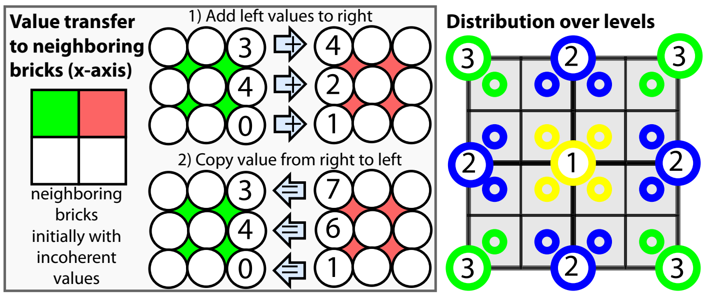

Value transfer to neighboring bricks (x-axis)
1) Add left values to right
3
4
0
4
2
1
2) Copy value from right to left
3
4
0
7
6
1
neighboring bricks initially with incoherent values
Distribution over levels
3
2
3
2
1
2
3
2
3

Figure 9.7. Left: During photon splatting, each photon is only stored in one voxel, therefore, inconsistencies appear for duplicated voxels of neighboring nodes. An addition and copy along each axis corrects this issue. Right: To filter values from a lower to a higher level, three passes are applied (numbers). The threads sum up lower-level voxels (all around the indicated octants) and store them in the higher level.

Value transfer to neighboring bricks In order to simplify the explanation, let's consider that our octree is complete, so that we can then launch one thread per leaf node.

We will perform six passes, two for each axis (x, y, z). In the first x-axis pass (Fig. 9.7, only left), each thread will add voxel data from the current node to to the corresponding voxels of the brick to its *right*. In practice, this means that three values per thread are added. The next pass for the x-axis will transfers data from the right (where we now have the sum) to the left by copying the values. After this step, values along the x-axis are coherent and correctly distributed. Repeating the same process for the y and z-axis ensures that all voxels have been correctly updated. The approach is very efficient because the neighbor pointers allow us to quickly access neighboring nodes and thread collisions are avoided. In fact, not even atomic operations are needed.

Distribution over levels At this point we have coherent information on the lowest level of the octree and the next step is to filter the values and store the result in the higher levels. A simple solution would be to launch one thread on each voxel of the higher level and fetch data from the lower level. Nonetheless, this has an important disadvantage: For shared voxels, the same computations are performed many (up to eight) times. Also, the computation cost of the threads differs depending on the processed voxel leading to an unbalanced scheduling.

Our solution is to perform three separate passes in which all threads have roughly the same cost (Fig. 9.7, right). The idea is to only partially compute the filtered results and use the previously-presented transfer between bricks to complete the result.

The first pass computes the center voxel using the involved 27 voxel values on the lower level (indicated by the yellow octants in Fig. 9.7). The second pass computes *half* of the filtered response for the voxels situated in the center of the node's faces (blue). Because only half the value is computed, only 18 voxel values are involved. Finally, the third pass launches threads for the corner voxels (green) that compute a partial filtering of voxels from a single octant.

After these passes, the higher-level voxels are in a similar situation as were the leaves after the initial photon splatting: octree vertices might only contain a part of the result, but summing values across bricks, gives the desired result. Consequently, it is enough to apply the previously-presented transfer step to finalize the filtering.

Sparse octree So far we assumed that the octree is complete, but in reality, our structure is sparse. Furthermore, the previously-described solution consumes many resources: During the filtering, we launched threads for all nodes, even those that did not contain any photon. Consequently, often filtering was just applied on zero values. This is crucial, as direct light often only affects a part of the scene. Here, we will propose a better control of the thread calls and at the same time deal with an incomplete/sparse octree.

To avoid filtering zero values, one could launch a thread per light-view-map pixel, find the corresponding leaf node and then walk up in the structure to the level on which the filtering should be applied and execute it. The result would be correct because our filtering was carefully designed to not introduce read-write conflicts (one thread simply overwrites the other's result). Unfortunately, whenever threads end up in the same node, work is performed multiple times. We want to reduce this overhead, but detecting an optimal thread set is very costly in practice. Our solution is approximate, but very efficient and delivers a good tradeoff.

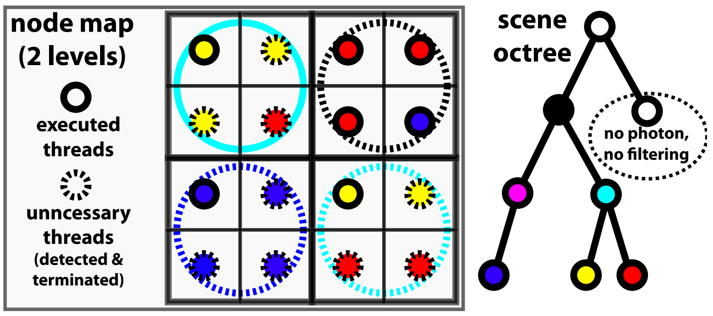


Figure 9.8. The node map (left) is constructed from the light-view map. It is hierarchical (like a MipMap) and has several levels (large circles represent the next level). On the lowest level, a pixel contains the index of the node in which the corresponding photon is located. Higher levels contain the common lowest ancestor of all underlying nodes. This structure allows us to avoid launching too many threads during the filtering of the photons (dashed threads are stopped). Further, by building upon the light-view map, nodes that did not receive photons will also not receive threads.

Our idea is to rely on a 2D *node map* derived from the light-view map. It resembles a Mipmap and reduces its resolution at each level. The pixels of the lowest node-map level store the indices of the 3D leaf nodes containing the corresponding photon of the light-view map. Higher-level node-map pixels store the index of the lowest common ancestor node for the preceding nodes of the previous level (Fig.9.8).

We still launch one thread per pixel of the lowest node-map. But when a thread moves up from level $i-1$ to the next, it first looks up its corresponding ancestor node index in the i^{th} level of the node map that is stored in some pixel p. Let $p_0, \ldots, p_3$ be the pixels in the $(i-1)^{th}$ level of the node map, that were fused into p. If the ancestor node in p is below or on the i^{th} level of the octree, we can deduce that all threads that passed through $p_0, \ldots, p_3$ will afterwards end up in the same node. Therefore, we would like to terminate all threads except one, but this is hard to achieve. To stop at least many unnecessary threads, we will employ a heuristic process and stop all those threads that did not traverse p_0 (in practice, this is the top-left pixel). Fig.9.8 illustrates with dashed lines the node where threads are stopped when they pass.

This strategy is quite successful and we obtain a > 2 speedup compared to not using our heuristic to terminate threads. Further, our filtering performs generally much faster than the naive implementation that filters all nodes. Exact timings for the latter are difficult to provide because we only filter where we find photons and so it depends highly on the scene configuration.

9.8 Results and discussion

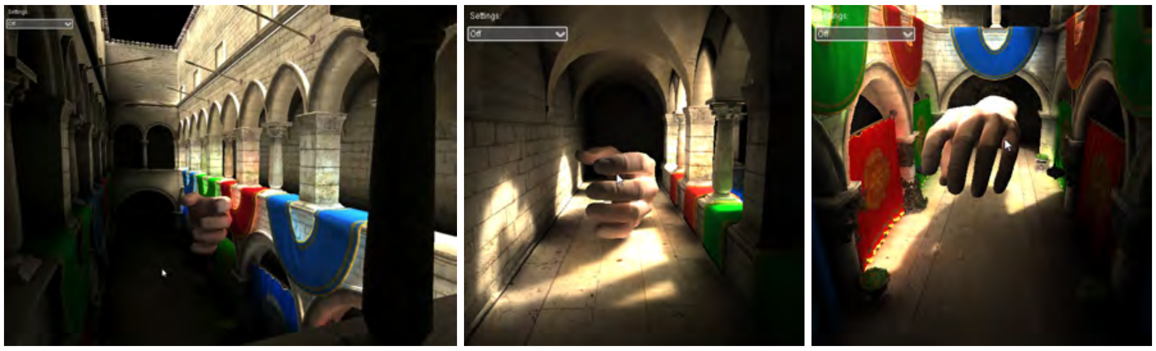


Figure 9.9. Examples of real-time rendering (25-30FPS) of two bounds indirect illumination with a fully animated object (hand) voxelized interactively inside our octree structure.

We implemented our approach on a GTX 480 system with an Intel Core 2 Duo E6850 CPU. Our solution delivers interactive to real-time frame rates on various complex scenes with both indirect diffuse and specular effects, as shown in Figures 9.9 and 9.13, and as can be seen in the online video (`http://artis.imag.fr/Membres/Cyril.Crassin/PG2011/PG2011.mov`).

Table 9.1 Timings in ms (for a 512^2 image) of each individual effect (Mesh rasterisation, Direct lighting, Indirect diffuse, Total direct+indirect diffuse, indirect specular, direct+indirect diffuse+indirect specular) on the Sponza scene alone using 3 diffuse cones and a 10°specular cone aperture

Steps	Rast.	Dir.	Dif.	Dir+dif.	Spec.	Total
Times	1.0	2.0	7.8	14.2	8.0	33.0

Table 9.1 shows the timing of each individual effect that can achieve our approach for the Sponza scene (Fig. 9.13, 280K triangles) with a 9 level octree (512^3 virtual resolution). In addition to these costs, the interactive update of the sparse octree structure for dynamic objects (in our case the Wald's hand 16K triangles mesh) takes approximatively 5.5ms per frame. On the Sponza scene, that gives an average frame rate of 30FPS with no specular cone traced, and 20FPS with one specular cone. The pre-process for the creation of the octree for the static environment takes approximatively 280ms.

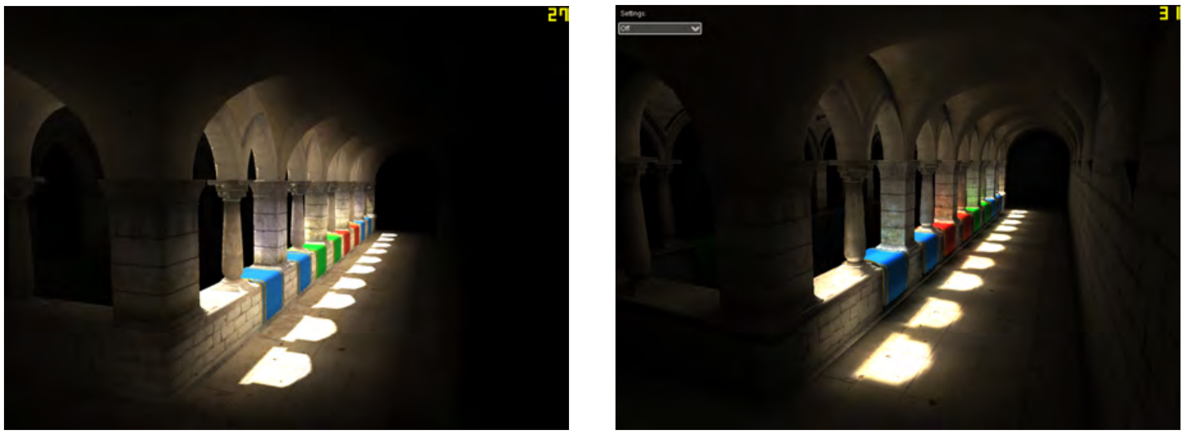


Figure 9.10. Image quality and performance comparison between [KD10] (left, 27FPS) and our approach (right, 31FPS).

We compared our approach to one of the most closely related real-time solution that is [KD10]. In all situations, our solution achieves higher visual quality (Fig.9.10) and maintains better performance. Our approach can capture much more precise indirect illumination even far from the observer, while [KD10] nested grids representation prevents it.

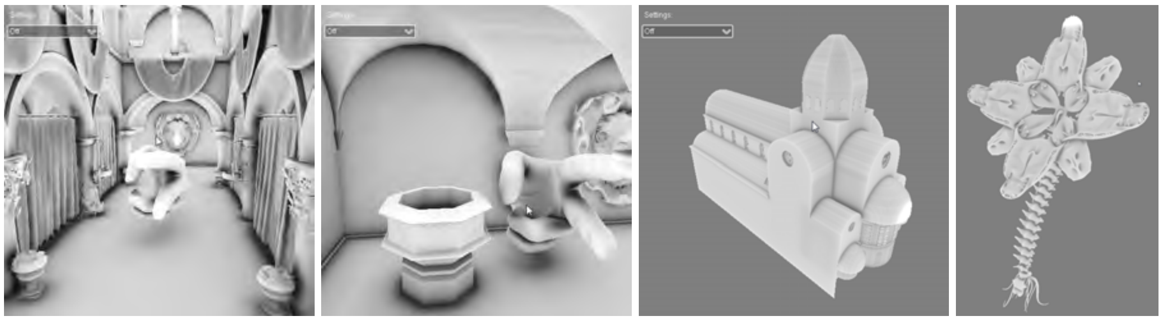


Figure 9.11. Real-time ambient occlusion rendered in real-time (70-150FPS) with our approach.

Figure 9.11 shows results we got with the rendering of ambient occlusion with our approach. We capture a lot more detail than SSAO (Screen Space Ambient Occlusion [Mit07]) approaches, and our approach performs especially better when some regions occluded on the screen produce high amounts of occlusion. Timings for ambient occlusion computations are shown in Table 9.2 and a comparison with a ground truth ray-traced with OptiX [NVI11c] is presented in Figure 9.8.

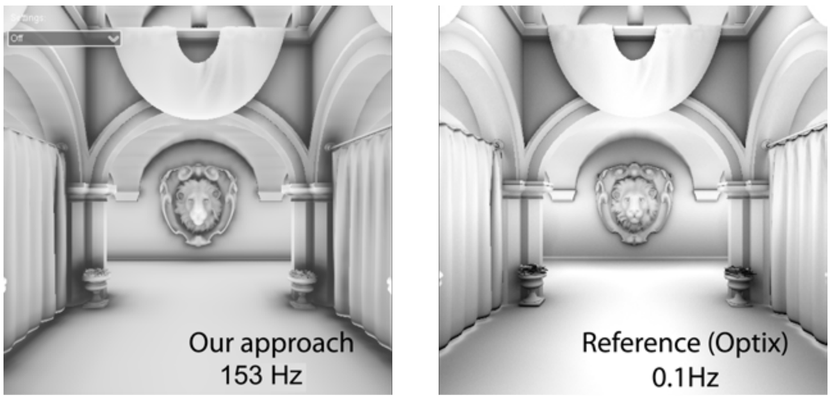


Figure 9.12. Comparison with ground through for ambient occlusion computation. Our (153FPS) vs. reference (OptiX, 0.1FPS)

Table 9.2 Timings (Full rendering 512^2) in ms for the Sponza scene of the ambient occlusion computation for 3 cones and various cone apertures.

Cone aperture (deg)	10	20	30	60
AO	16.6	9.0	6.5	3.9

One critical point of our solution is the memory consumption that can be very high even with our sparse structure, especially due to the support for indirect specularity (that requires the storage of more information per-voxel). In practice, we allocate roughly 512MB on the GPU, however, this is lower than in [KD10] when trying to achieve comparable visual quality. As for many real-time approaches, our solution exhibits differences in comparison with a reference solution, especially on high-frequency details. Our view-dependent refinement and approximate cone-tracing distributes more precision near to the observer, but this can be a disadvantage for particular configurations in glossy scenes. For instance, precise caustic effects would be difficult to capture. However, this limitation applies to any existing real-time solution to a similar extent and we propose one of the only solutions that is able to capture indirect specularities in real-time.

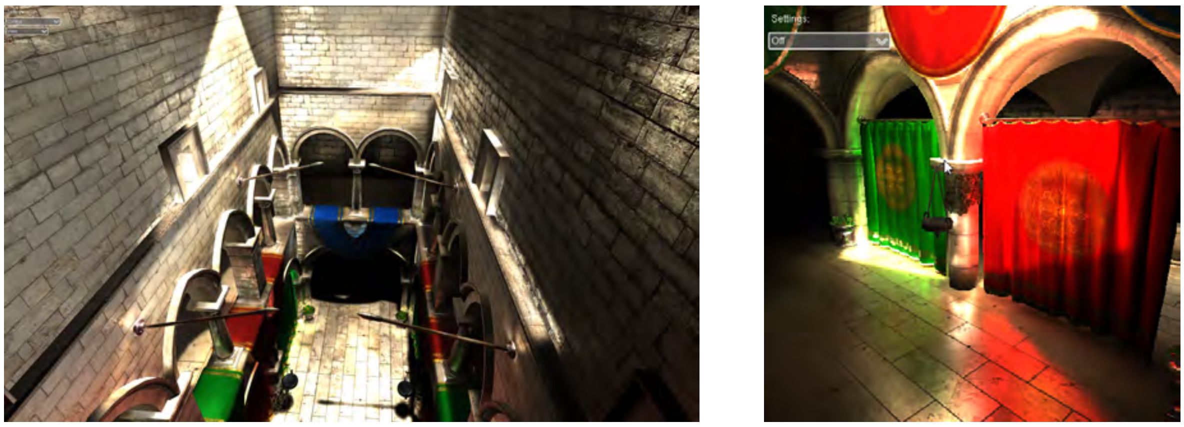


Figure 9.13. Our method supports diffuse indirect lighting as well as glossy reflections in real-time.

9.9 Conclusion

We presented a novel real-time global illumination algorithm. Using adaptive representations, we are able to compute approximate two-bound indirect lighting in complex dynamic scenes. We also proposed a new real-time approach for the voxelization and the pre-filtering of dynamic objects, as well as the interactive update of an octree structure. Our solution supports indirect diffuse as well specular illumination. Quality-wise, it outperforms existing competitors due to our approximate cone tracing. It scales well when trading off quality against performance and could integrate seamlessly into out-of-core rendering systems as the one presented Chapter 7. In the future, we want to work on the integration of our application controlled paging system in order to improve the achievable precision and to allow indirect illumination computation on very large scenes. One of the difficulties will be to mix on-demand loading and caching of static parts of the scene, with the interactive voxelization of dynamic objects.

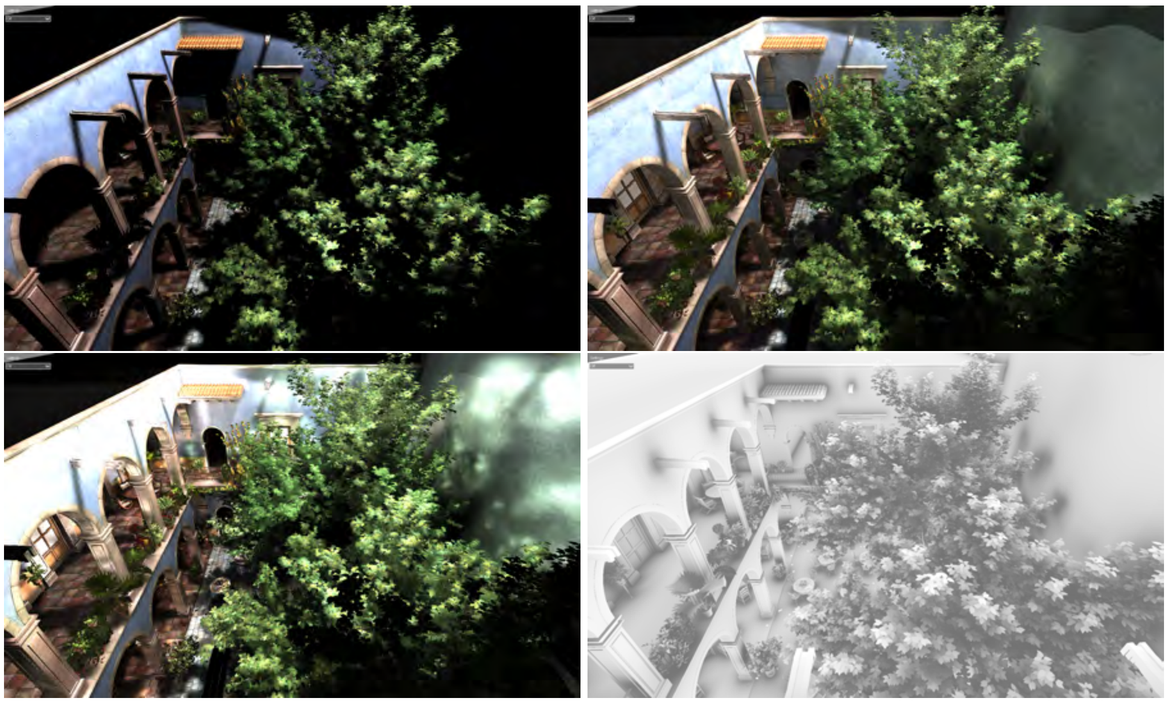


Figure 9.14. The SanMiguel test scene with (1)Direct lighting only (2)Indirect diffuse only (3)Indirect diffuse and specular (4)Ambient occlusion only

Part III

Conclusion

Chapter 10

Conclusions and perspectives

Summary of contributions

In this thesis, we have proposed several solutions in order to unlock the usage of very large voxel representations in real-time applications as a way to render large and detailed scenes, by relying on an appearance preserving pre-filtered geometry representation.

We proposed a model for representing and pre-filtering geometry inside a voxel-based 3D MIP-map pyramid. From this, we built a pre-integrated cone tracing algorithm allowing very fast alias-free rendering. This voxel-based cone tracing allows us to efficiently approximate visibility and lighting integration inside a cone footprint. Moreover, we have demonstrated how to deal efficiently with the main problem of voxel representations: the huge memory consumption. In order to bring memory-intensive voxel representations as a standard GPU primitive, we proposed a new rendering pipeline for high-performance rendering of large and detailed volumetric objects and scenes on the GPU.

Our pipeline is centered around a new sparse octree data structure providing a compact storage and an efficient access and update to the pre-filtered voxel representation. This structure is used by a fast GPU rendering algorithm based on ray-casting, that provides an adaptive multiresolution rendering approach. This makes the rendering of voxel-based scenes independent of the complexity of the underlining geometry, providing a fully scalable way to render very complex scenes.

However, even with a compact data structure, voxel data representations usually exceed the memory of current GPUs by large amounts. In addition, since we wanted our approach to scale to arbitrarily large scenes, restricting the storage of a scene to the amount of data that fits inside the video memory would not have been sufficient. Thus, we built an efficient GPU-based caching and on-demand loading mechanism that allows us to virtualize totally our voxel data structure, and to keep only a small subset of the whole dataset inside the video memory. This caching scheme maximizes the reuse of data loading inside the video memory during the exploration of a scene, thus minimizing the streaming of data.

Data can be either streamed from the much larger system memory, or generated directly on the GPU either procedurally or from another representation (such as a triangle mesh voxelized on the fly). This mechanism is entirely triggered by requests emitted per-ray directly during rendering, providing exact visibility determination and minimal data production or loading. These efficient strategies adapt the volume resolution according to the point of view and enable us to completely overcome the memory limitation of the GPU, allowing fast rendering and exploration of very complex scenes and objects. 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 are all integrated in the same framework: our per-ray queries.

Thanks to these strategies, we demonstrated how the rendering based on our voxel representation can provide high quality and can be more efficient than the standard GPU rasterization for very complex meshes. We validated our approach with several example scenes presented throughout this thesis, as well as performance analysis associated with each main parts of this work.

Based on these main contributions, we demonstrated how our new voxel-based geometry representation and pre-integrated cone-tracing can be used to render efficiently blurry effects such as soft shadows and depth-of-field. Finally, we introduced a new real-time approach to estimate two bounds of indirect lighting as well as ambient occlusion, relying on our pre-filtered geometry representation and voxel-based cone tracing. For this application, we have also shown how our representation can handle animated objects through dynamic updates of the data structure and real-time voxelization of triangle meshes.

We have demonstrated how voxel-based representations can be a valuable solution to represent very complex scenes and objects. All their advantages hint at a more extensive future use of such representations in real-time applications, in particular in video games. The whole approach we proposed in this thesis provides an efficient solution in this context, and we believe it will pave the way to many new interesting effects and rendering paradigms.

Perspectives and future work

In this thesis, we tackled the problem of memory consumption of voxel representations, and we strongly believe that we built the foundations required for future work and research based on massive voxel scenes. In order to make this work fully reusable by other researchers, or to be easily integrated into an industrial project, we are currently transforming our reference implementation into an open source project that will be entirely freely available to the community.

We see several interesting future research directions based on our work. The first main area we would like to explore is the animation of voxel representations. Being able to render efficiently animated scenes is now the major block for the usage of voxel representations in video games. We started to address this problem in our global illumination application, by proposing a real-time voxelization and pre-filtering scheme of animated triangle-based geometry. However, this scheme is not optimal. First it is not compatible with our on-demand loading scheme, and thus does not scale with a large number of animated objects. The voxelization of an animated object has to be done at each frame, whenever it is visible or not. In addition, this voxelization has to be done from bottom to top in order to compute pre-filtering, which means that a high resolution voxelization is always needed, whatever the resolution actually required for rendering.

This represents a major problem for the scalability of the approach and leads to another major area of future work: the efficient and scalable pre-filtering of complex representations. Providing solutions to the top-to-bottom pre-filtering problem based on input surface geometry is challenging, however we believe that approaches based on procedural generation are a very promising direction to explore. Furthermore, correlation aware filtering schemes (cf. our decorrelation hypotheses, in Section 4.4) are required in order to enhance the quality and the accuracy of the cone-tracing approach, especially when using large cone apertures.

For the animation of voxel representations, we strongly believe in animation through deformation of the volumetric representation using a shell-map approach [PBFJ05, JMW07]. This would make it possible to animate a low resolution triangle mesh, like a character, and to add thin volumetric details that would be deformed on its surface. Such deformation poses several major problems. Among them, the efficient tracing of curved rays and cones in the voxel domain, as well as the need for pre-filtering with anisotropic volume dimensions, in order to handle geometry compressions. We also see a great interest of such shell-maps using very high resolution voxel data for authoring tools. Indeed, voxel sculpting would greatly benefit from the ability to directly add high resolution voxel details inside a thin layer at the surface of triangle mesh. The mesh could be moved and deformed freely during the editing to add details where needed, at any resolution.

More generally, in the context of high performance parallel rendering, we strongly believe that we are going toward the use of more and more structured representations to render synthetic scenes. Such structuring is critical to allow image-order rendering approaches, that allow minimizing data access as well as framebuffer bandwidth. Data access becomes the major bottleneck in parallel computing architectures, both in terms of latency and in terms of power consumption. In addition, such scene structuring that provides fast global random access to any geometry of the scene, allows for an easier implementation of global effects which are difficult to achieve with rasterization-based approaches. Consequently, we believe in the convergence between the object-order rasterization approach and the image-order ray-tracing approaches, with increasing needs for screen-space locality and thus increasing structuring of the input geometric data.

Part IV

Appendix

Chapter A

Preliminary GPU performance characterization

A.1 Characterizing texture cache behavior

In order to make an optimal volume ray-casting implementation on a given hardware, one of the most important points is to characterize the behavior of the texture cache with the sampling scheme appearing in ray-casting typical usage. The texture cache (cf. Section 1.3) is an on-chip very fast memory that is used by the texture sampling hardware of the GPU. It allows us to maximize the reuse of data read from the video memory among multiple threads and different samples. This lowers the amount of data transfered between the GPU and the video memory, and improves performance of texture access. The performance of this cache is critical for our ray-casting application that heavily relies on texture sampling operations (cf. Chapter 6).

In our ray-casting usage, the rendering is done by assigning one thread per screen pixel, each thread in charge of computing the volume rendering integral along a single ray by sequentially sampling inside bricks stored in a 3D texture (Chap. 6). The performance behavior of the texture cache in such a context will impact the technological choices we will make in terms of the scheduling of the rendering threads, and the type of texture to use.

Especially, the main questions we wanted to answer were:

- Is the cache behavior isotropic ? Is the cache efficiency the same whatever the viewing direction ? If not, how does it impact performance ?
- How does the caching performance vary in function of the type of texture used ? In our case, a 3D texture that provides fully accelerated trilinear access to the data can be used, or a Layered 2D texture that only provides bilinear interpolation, but may provide better access performance. In the case of the Layered 2D texture, the third interpolation has to be computed inside the shaders/CUDA kernel, increasing the bandwidth used between the shader units and the texture units and the charge of the stream processors.

To answer these questions, we built a series of tests using a simple ray-casting code implemented in CUDA, sampling volume data inside a texture. Rays are launched using orthographic projection, with one ray (thread) per pixel. Each thread does a fixed number of texture access (1000 texture access/thread). We compared the performance of different types of textures and data formats depending on the traversal direction and the screen orientation, and we tested them with and without interpolation. These tests were done on an NVIDIA GTX580 GPU.

The results of these experiments are presented in Table A.1 and Graph A.1. They present the time spent in texture reads depending on the traversal direction (in the form $ScreenAxis_x$ $ScreenAxis_y$ $DepthAxis$) and the texture type, for RGBA8 texel format. We compared access to 3D and Layered 2D textures (2D texture arrays, `EXT_texture_array`) with the nearest interpolation. We also compared the access time to these two texture types with linear interpolation. We tested 3D texture with full trilinear interpolation, Layered 2D texture with bilinear interpolation and no interpolation between layers, and Layered 2D with interpolation between layers done in the kernel.

The first interesting thing to note is that the texture read performance is not the same whatever the traversal direction. This means that the texture cache does not have an isotropic behavior on successive reads, and thus that geometry of the cache lines is not cubical. The second interesting information is that with a full trilinear interpolation, 3D textures appear on average 1.74x faster than Layered 2D textures (when averaging the 6 tested screen orientations and directions).

Table A.1 Total times of texture reads for ray-casting inside a 512^3 RGBA8 texture, with orthographic projection and step size $\frac{1}{512}$, depending on the view direction and orientation (and in average), and the type of texture employed.

Filter Mode	Texture type	XY Z	YX Z	XZ Y	ZX Y	YZ X	ZY X	Average
Nearest	3D	11,90	11,76	11,76	14,08	19,23	18,52	14,54
	Layered 2D	9,43	9,52	11,90	31,25	22,22	27,78	18,69
Linear	3D	12,20	13,16	26,32	37,04	43,48	50,00	30,36
	Layered 2D No Z interp.	10,53	13,89	23,26	76,92	43,48	66,67	39,12
	Layered 2D Man. Z interp.	21,28	45,45	28,57	100,00	50,00	71,43	52,79

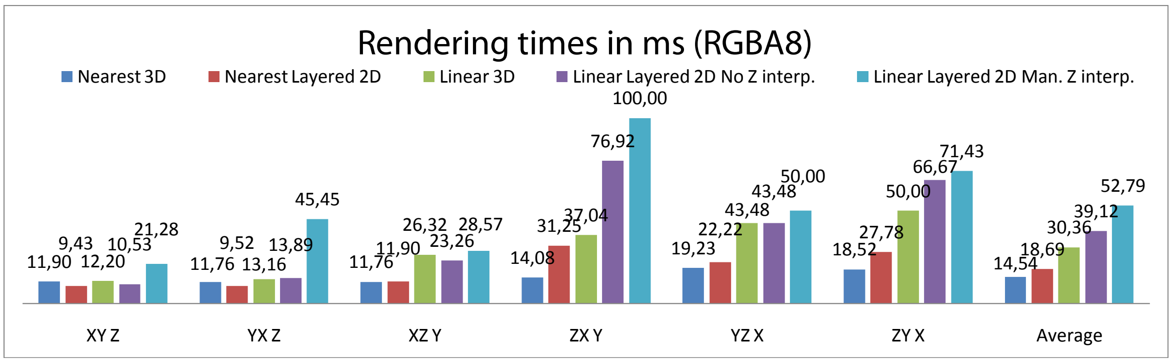


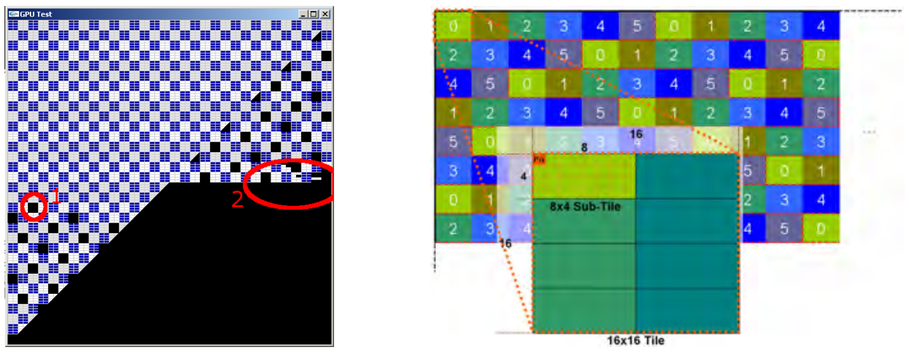
Figure A.1. Graph view of the data presented table A.1.

A.2 Rasterization and scheduling of fragment shading

A.2.1 Motivations

Many hardware details are still hidden from the programmer, in particular mechanisms used for primitives rasterisation and fragments shading. Understanding how fragments are scheduled among the GPU processing units is a critical points for our research, and we investigated the behaviour of this part of the GPU pipeline using a series of directed tests.

Criticality of optimization on GPU is very different than for CPU, due to the extremely high performance contrasts. GPUs have "fast paths": Unintuitive depressing 1000x slowdowns as compared to expectations are often met when programming. Conversely, knowing these fast paths can lead to more than 1000x speedups. To find and remain in these fast paths, it is important to understand how the GPU works and behaves. In particular, we investigated how fragment shader threads are scheduled among the GPU "stream processors" in charge of executing them (Sec. 1.3). These low level experiments were done on the G80 NVIDIA GPU in 2007.



(a) (b)

Figure A.2. *(a):* A tile containing a very slow fragment (1) makes all next tiles scheduled on the same TP wait. Others TPs continue their processing until their fragments FIFO are empty (2). *(b):* Screen subdivision in tiles spread among G80 MPs. Tiles subdivision into 8x4 pixels sub-tiles.

A.2.2 Methodology

We wrote a small probe program "fragSniffer" allowing us to trigger various configuration tests providing 2 kinds of outputs:

- Showing fragment writing order into the front-buffer, comprising locks, stalls, synchronizations.
- Measuring performance changes when changing configuration parameters.

The principle is to use very slow fragment shaders (doing simple additions into a user controlled loop, typically 10^6 iterations). In particular, some of our fragment shaders slow down only for one pixel on screen, or for a couple of pixels. In case of several slow pixels, these can run either the same shader instructions or 2 different branches of a conditional statement. Speed, number of slow pixels and their location, as well as speed of "background fragments" can all be controlled manually.

Tested configurations concern the pattern and relative location of slow pixels, and also the type of primitive drawn on screen: large or small, stripped or not, 2D,1D or 0D, traced in smart, raster or shuffle order, tiling the screen or overlapping. Note that due to the use of extremely slow shaders, the cost of CPU, bus transfer and vertex transform is negligible.

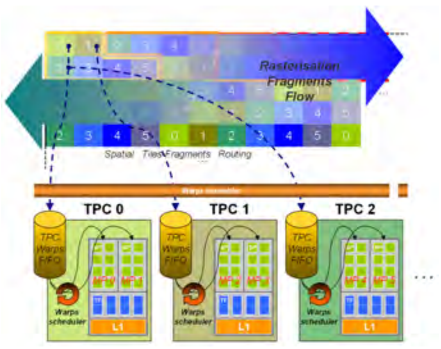
Our probe tool fragSniffer is freely available here: `http://www.icare3d.org/FragSniffer/FragSniffer_0.2.zip`

The board used for our experiments was a 8800 GTS: it has 96 Stream processors (SP), grouped by 8 working in SIMD into 12 Multi Processors (MP), which are paired as 6 Texture Processors (TP). Threads are scheduled on Multi Processors into Warps of 32 threads executed in SIMD (within 4 cycles) on the 8 SPs of the MP (see [NVI11a] for more details).

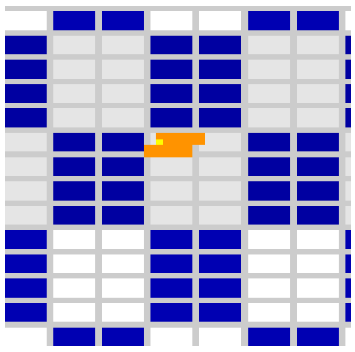
We also ran our probe on other boards, e.g. a 8600M GT having 16 SP grouped into 4 MP =2 TP just to verify that our results were consistent.

Disclaimer

The G80 is a quite complex ecosystem we tried to locally understand by running these experiments. We might have misinterpreted some behaviors, conducted some inappropriate experiments, or even incorrectly designed or run some of them. We provide our observations and conclusions so that you can trace our reasoning, and provide our probing tool so that our data can be verified.



(a)



(b)

Figure A.3. *(a):* Screen subdivision in tiles spread among G80 MPs. Tiles subdivision into 8x4 pixels sub-tiles. *(b)*: Display of the footprint of a 32 threads warp.

A.2.3 Summary of our "interpretations and discoveries"

Here is a quick summary of our initial conclusions; all our experiments and deductions can be found in our online article `http://www-evasion.imag.fr/GPU/CN08`.

- The rasterizer allocates fragments to texture processors (pairs of multiprocessors) based on the location on screen:the screen is subdivided in tiles of size 16x16 which are bounds to TPs according to a fixed pattern (see Figure A.2(b)).
- For one given TP, the flow of fragments is assembled in warps of 32 threads then stored in a FIFO (Fig. A.3(a)). Warps of one FIFO are executed by any of the two MPs of the TP.
- Threading can use shader wait-states (texture access, transcendent maths, pipeline dependencies) to run some warps partly in parallel on the same MP.
- In the general case, warp fragments are not geometrically ordered and can correspond to any location within the screen footprint of a TP (see Figure A.3(b)).
- In fact, fragments are managed in groups of 2x2 (4x2?) "superfragments". In particular, points, lines and triangle borders yield some waste since "ghost fragments" are generated to fill superfragments and are treated as regular threads (with no output). For an unknown reason, only 4 isolated points or primitives of size 1 can fit a warp (8 were expected).
- If one FIFO is full the rasterizer has to wait, which might starve the other TPs (see Figure A.2(a)).

Bibliography

[ABA02] Carlos Andújar, Pere Brunet, and Dolors Ayala. Topology-reducing surface simplification using a discrete solid representation. *ACM Transactions on Graphics*, 21:88–105, April 2002.

[AFO05] Okan Arikan, David A. Forsyth, and James F. O'Brien. Fast and detailed approximate global illumination by irradiance decomposition. In *ACM Transactions on Graphics (Proc. SIGGRAPH)*, pages 1108–1114, 2005.

[AH05] A. Asirvatham and H. Hoppe. *GPU Gems 2*, chapter "Terrain rendering using GPU-based geometry clipmaps", pages 109–122. 2005.

[Ake93] Kurt Akeley. Reality engine graphics. In *Proceedings of SIGGRAPH '93*, pages 109–116, 1993.

[AL04] Timo AIla and Samuli Laine. Alias-free shadow maps. In *Proceedings of EGSR*, pages 161–166, June 2004.

[AM00] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5:9–22, January 2000.

[AM04] T. Aila and V. Miettinen. dpvs: an occlusion culling system for massive dynamic environments. *Computer Graphics and Applications, IEEE*, 24(2):86 – 97, march-april 2004.

[Ama84] John Amanatides. Ray tracing with cones. In *Proceedings of SIGGRAPH '84*, pages 129–135, 1984.

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., 2008.

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, pages 3–10. 1987.

[BD02] David Benson and Joel Davis. Octree textures. In *SIGGRAPH*, pages 785–790, 2002.

[BD06a] Lionel Baboud and Xavier Décoret. Realistic water volumes in real-time. In *EG Workshop on Natural Phenomena*. Eurographics, 2006.

[BD06b] Lionel Baboud and Xavier Décoret. Rendering geometry with relief textures. In *Graphics Interface '06*, 2006.

[BEW$^+$98] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a PC Game Engine. *IEEE Comput. Graph. Appl.*, 18:46–53, January 1998.

[BHGS06] Tamy Boubekeur, Wolfgang Heidrich, Xavier Granier, and Christophe Schlick. Volume-surface trees. *Computer Graphics Forum*, 25(3):399–409, 2006. Proceedings of EUROGRAPHICS 2006.

[BHMF08] Johanna Beyer, Markus Hadwiger, Torsten Möller, and Laura Fritz. Smooth Mixed-Resolution GPU Volume Rendering. In *IEEE/EG International Symposium on Volume and Point-Based Graphics*, pages 163 – 170, 2008.

[Bik07] J. Bikker. Real-time ray tracing through the eyes of a game developer. In *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, pages 1–1, Sept. 2007.

[Bli82] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. In *SIGGRAPH : Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, 1982.

[Bly06] David Blythe. The direct3d 10 system. *ACM Transactions on Graphics*, 25:724–734, July 2006.

[BMW$^+$09] Jiri Bittner, Oliver Mattausch, Peter Wonka, Vlastimil Havran, and Michael Wimmer. Adaptive global visibility sampling. *ACM Transactions on Graphics*, 28(3):94:1–94:10, August 2009. Proceedings of ACM SIGGRAPH 2009.

[BN11] Eric Bruneton and Fabrice Neyret. A survey of non-linear pre-filtering methods for efficient and accurate surface shading. *IEEE Transactions on Visualization and Computer Graphics*, 2011.

[BNL06] Antoine Bouthors, Fabrice Neyret, and Sylvain Lefebvre. Real-time realistic illumination and shading of stratiform clouds. In *Eurographics Workshop on Natural Phenomena*, sep 2006.

[BNM$^+$08] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *ACM Symposium on Interactive 3D Graphics and Games (I3D)*, 2008.

[BNS01] Imma Boada, Isabel Navazo, and Roberto Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 13(3), 2001.

[BOA09] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 159–166, 2009.

[BT04] Zoe Brawley and Natalya Tatarchuk. Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. In *ShaderX3: Advanced Rendering Techniques in DirectX and OpenGL*. 2004.

[BTG03] Bill La Barge, Jerry Tessendorf, and Vijoy Gaddipati. Tetrad volume and particle rendering in X2. In *SIGGRAPH Sketch*, 2003. http://portal.acm.org/ft_gateway.cfm?id=965491.

[BW03] Jiri Bittner and Peter Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 30(5):729–755, September 2003.

[BWPP04] Jiří Bittner, Michael Wimmer, Harald Piringer, and Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624, September 2004. Proceedings of EUROGRAPHICS 2004.

[Car03] Christian Carvajal. Shaken and stirred, *XXX* visual effects. *CINEFEX*, 92, 2003.

[Cat74] Edwin Earl Catmull. *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, 1974. AAI7504786.

[CB04a] Per H. Christensen and Dana Batali. An irradiance atlas for global illumination in complex production scenes. In *Rendering Techniques (EGSR)*, pages 133–142, 2004.

[CB04b] Per H. Christensen and Dana Batali. An irradiance atlas for global illumination in complex production scenes. In *Proceedings of EGSR*, pages 133–141, June 2004.

[CBWR07] Jean Pierre Charalambos, Jiří Bittner, Michael Wimmer, and Eduardo Romero. Optimized hlod refinement driven by hardware occlusion queries. In *Advances in Visual Computing (Third International Symposium on Visual Computing – ISVC 2007)*, pages 106–117. Springer, November 2007.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. In *Proceedings of SIGGRAPH '87*, pages 95–102, 1987.

[CCF94] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS: Proceedings of the 1994 symposium on Volume visualization*, 1994.

[CDP95] Frédéric Cazals, George Drettakis, and Claude Puech. Filtering, clustering and hierarchy construction: a new solution for ray tracing very complex environments. In *Eurographics*, September 1995. Maastricht.

[CE97] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 235–., 1997.

[CE98] David Cline and Parris K. Egbert. Interactive display of very large textures. In *Proceedings of the conference on Visualization '98*, pages 343–350, 1998.

[CGP04] Sharat Chandran, Ajay K. Gupta, and Ashwini Patgawkar. A fast algorithm to display octrees, September 18 2004.

[CHCH06] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast GPU ray tracing of dynamic meshes using geometry images. In *GI : Proceedings of the 2006 conference on Graphics interface*, pages 203–209, 2006.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19:547–554, October 1976.

[CN94] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3D texture hardware. Technical report, 1994.

[CN07] Cyril Crassin and Fabrice Neyret. Représentation et algorithmes pour l'exploration interactive de volumes procéduraux étendus et détaillés. Master's thesis, UJF, INPG, june 2007.

[CNLE09] 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)*, feb 2009.

[CNSE10] Cyril Crassin, Fabrice Neyret, Miguel Sainz, and Elmar Eisemann. *Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In book: GPU Pro*, chapter X.3, pages 643–676. A K Peters, 2010.

[COCSD03a] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Claudio T. Silva, and Fredo Durand. A Survey of Visibility for Walkthrough Applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[COCSD03b] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. In *IEEE Transactions on Visualization and Computer Graphics*, volume 9, pages 412–431, July 2003.

[COM98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *Proceedings of SIGGRAPH '98*, pages 115–122, 1998.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of SIGGRAPH '84*, pages 137–145, 1984.

[Cro77] Franklin C. Crow. The aliasing problem in computer-generated shaded images. *Communications of the ACM*, 20:799–805, November 1977.

[Cry10] Crytek. Crytek terrains authoring, 2010.

[CS94] Daniel Cohen and Zvi Sheffer. Proximity clouds - an acceleration technique for 3D grid traversal. *The Visual Computer*, 11(1):27–38, 1994.

[CZP68] Kenneth M. Case, Paul F. Zweifel, and G. C. Pomraning. Linear transport theory. *Physics Today*, 21(10):72–73, 1968.

[DB89] Jevans D. and Wyvill B. Adaptive voxel subdivision for ray tracing. In *Graphics Interface*, pages 164–172, June 1989.

[DDSD03] Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *ACM SIGGRAPH 2003 Papers*, pages 689–696, 2003.

[DGPR02] David DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. In *SIGGRAPH*, pages 763–768, 2002.

[DGR$^+$09] Z. Dong, T. Grosch, T. Ritschel, J. Kautz, and H.-P. Seidel. Real-time indirect illumination with clustered visibility. In *Proceedings of VMV*, 2009.

[DH92] John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of the 1992 workshop on Volume visualization*, pages 91–98, 1992.

[DKTS07] Zhao Dong, Jan Kautz, Christian Theobalt, and Hans-Peter Seidel. Interactive global illumination using implicit visibility. In *Proceedings of Pacific Graphics*, 2007.

[DN04] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. In *Rendering Techniques (EGSR)*, pages 93–102, june 2004.

[DN09] Philippe Decaudin and Fabrice Neyret. Volumetric billboards. *Computer Graphics Forum*, 28(8):2079–2089, 2009.

[Dom] Digital Domain. Digital domain web site. http://www.digitaldomain.com.

[DPH$^+$03] David E. DeMarle, Steven Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed interactive ray tracing for large volume visualization. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 12–, 2003.

[DS05] C. Dachsbacher and M. Stamminger. Reflective shadow maps. In *Proceedings of I3D*, pages 203–213, 2005.

[DSDD07] Carsten Dachsbacher, Marc Stamminger, George Drettakis, and Frédo Durand. Implicit visibility and antiradiance for interactive global illumination. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), 2007.

[DSSC08] Joel Daniels, Cláudio T. Silva, Jason Shepherd, and Elaine Cohen. Quadrilateral mesh simplification. In *ACM SIGGRAPH Asia 2008 papers*, pages 148:1–148:9, 2008.

[Dun04] Jody Duncan. Freeze frames, *The Day After Tomorrow* visual effects. *CINEFEX*, 98, 2004.

[ED08] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization and applications. In *GI '08: Proceedings of Graphics Interface 2008*, volume 322 of *ACM International Conference Proceeding Series*, pages 73–80. Canadian Information Processing Society, 2008.

[EHK$^+$04] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama, and Daniel Weiskopf. Real-time volume graphics. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 29, 2004.

[EHK$^+$06] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, and Daniel Weiskopf. *Real-time Volume Graphics*. A. K. Peters, Ltd., 2006.

[EKE01] 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)*, pages 9–16, 2001.

[EVG04] Manfred Ernst, Christian Vogelgsang, and Günther Greiner. Stack implementation on programmable graphics hardware. In *VMV*, pages 255–262, 2004.

[FBH$^+$10] Kayvon Fatahalian, Solomon Boulos, James Hegarty, Kurt Akeley, William R. Mark, Henry Moreton, and Pat Hanrahan. Reducing shading on gpus using quad-fragment merging. In *ACM SIGGRAPH 2010 papers*, pages 67:1–67:8, 2010.

[fer10] NVIDIA Fermi Architecture White Paper. `http://www.nvidia.com/object/fermi_architecture.html`, 2010.

[FLB$^+$09] Kayvon Fatahalian, Edward Luong, Solomon Boulos, Kurt Akeley, William R. Mark, and Pat Hanrahan. Data-parallel rasterization of micropolygons with defocus and motion blur. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 59–68, 2009.

[For07] Joe Fordham. *Pirates of the Caribbean: At World's End* visual effects. *CINEFEX*, 110, 2007.

[Fou92a] Alain Fournier. Filtering normal maps and creating multiple surfaces. Technical report, 1992.

[Fou92b] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface'92 Workshop on Local Illumination*, pages 45–52, May 1992.

[FP02a] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, us edition, August 2002.

[FP02b] Frisken and Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7, 2002.

[FP02c] Sarah F. Frisken and Ronald N. Perry. Simple and efficient traversal methods for quadtrees and octrees. *Journal of Graphics Tools*, 7(7):2002, 2002.

[FS97] Jason Freund and Kenneth Sloan. Accelerated volume rendering using homogeneous region encoding. In *Proceedings of the 8th conference on Visualization '97*, pages 191–ff., 1997.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *HWWS: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, 2005.

[FWS] Heiko Friedrich, Ingo Wald, and Philipp Slusallek. Interactive Iso-Surface Ray Tracing of Massive Volumetric Data Sets. In *Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization*.

[GBK06] Michael Guthe, Ákos Balázs, and Reinhard Klein. Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Eurographics Symposium on Rendering 2006*, June 2006.

[GBSF05] Anselm Grundhöfer, Benjamin Brombach, Robert Scheibe, and Bernd Fröhlich. Level of detail based occlusion culling for dynamic scenes. In *Proceedings of GRAPHITE '05*, pages 37–45, 2005.

[GHFP08] Jean-Dominique Gascuel, Nicolas Holzschuch, Gabriel Fournier, and Bernard Peroche. Fast non-linear projections using graphics hardware. In *Proceedings of I3D*, February 2008.

[GK96] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *VVS: Proceedings of the symposium on Volume visualization*, 1996.

[GKY08] Enrico Gobbetti, Dave Kasik, and Sung-eui Yoon. Technical strategies for massive model visualization. In *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 405–415, 2008.

[GM05] Enrico Gobbetti and Fabio Marton. 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.

[GMAG08] Enrico Gobbetti, Fabio Marton, Jose Antonio, and Iglesias Guitian. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, 2008.

[Gro] Khronos Group. Registre des extensions OpenGL. http://www.opengl.org/registry/.

[GS04] S. Guthe and W. Strasser. Advanced techniques for high quality multiresolution volume rendering. In *Computers & Graphics*, pages 51–58. Elsevier Science, 2004.

[GWGS02] Stefan Guthe, Michael Wand, Julius Gonser, and Wolfgang Strasser. Interactive rendering of large volume data sets. In *Proceedings of the conference on Visualization '02*, pages 53–60, 2002.

[GY98] Michael E. Goss and Kei Yuasa. Texture tile visibility determination for dynamic texture loading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 55–ff., 1998.

[Had02] Hadwiger et al. High-quality volume graphics on consumer PC hardware. In *Course Notes 42 - SIGGRAPH*, 2002.

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

[HH84] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. In *Proceedings of SIGGRAPH '84*, pages 119–127, 1984.

[HHS93] H. C. Hege, T. Hollerer, and D. Stalling. Volume rendering mathematical models and algorithmic aspects, 1993.

[HLSR09] Markus Hadwiger, Patric Ljung, Christof Rezk Salama, and Timo Ropinski. Advanced illumination techniques for GPU-based volume raycasting. In *ACM SIGGRAPH 2009 Courses*, pages 2:1–2:166, 2009.

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH '96*, pages 99–108, 1996.

[Hor05] D. Horn. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter "Stream reduction operations for GPGPU applications", pages 573–589. 2005.

[HPB07] Miloš Hašan, Fabio Pellacini, and Kavita Bala. Matrix row-column sampling for the many-light problem. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 26(3), 2007.

[HQK05] Wei Hong, Feng Qiu, and A. Kaufman. GPU-based object-order ray-casting for large datasets. In *Volume Graphics, Fourth International Workshop on*, pages 177–240, 2005.

[HSA91] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (Proc. SIGGRAPH)*, 25(4):197–206, 1991.

[HSC$^+$05] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum (Proc. Eurographics)*, 24(3):547–555, 2005.

[HSHH07] 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.

[HSO07] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[HSRG07] Charles Han, Bo Sun, Ravi Ramamoorthi, and Eitan Grinspun. Frequency domain normal map filtering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*, 26(3):28:1–28:12, 2007.

[HVAPB08] Miloš Hašan, Edgar Velázquez-Armendáriz, Fabio Pellacini, and Kavita Bala. Tensor Clustering for Rendering Many-Light Animations. *Comput. Graph. Forum (Proc. of EGSR)*, 27(4):1105–1114, 2008.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[Jen01] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, 2001.

[JLBM05] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics*, 24(4):1462–1482, 2005.

[JMW07] Stefan Jeschke, Stephan Mantler, and Michael Wimmer. Interactive smooth and curved shell mapping. In *Rendering Techniques 2007 (Proceedings Eurographics Symposium on Rendering)*, pages 351–360. Eurographics, 6 2007.

[Kaj86] James T. Kajiya. The rendering equation. *Comput. Graph. (Proc. SIGGRAPH)*, 20(4):143–150, 1986.

[Kap02] Alan Kapler. Evolution of a vfx voxel tool. In *SIGGRAPH Sketch*, 2002. http://portal.acm.org/ft_gateway.cfm?id=1242192.

[Kap03] Alan Kapler. Avalanche! snowy FX for XXX. In *SIGGRAPH Sketch*, 2003. http://portal.acm.org/ft_gateway.cfm?id=965492.

[KB08] Martin Kraus and Kai Bürger. Interpolating and downsampling rgba volume data. In *Proceedings of Vision, Modeling, and Visualization 2008*, 2008.

[KD02] Oliver Kersting and Jürgen Döllner. Interactive 3D visualization of vector data in GIS. In *GIS : Proceedings of the 10th ACM international symposium on Advances in geographic information systems*, pages 107–112, 2002.

[KD10] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of I3D*, 2010.

[KE02] Martin Kraus and Thomas Ertl. Adaptive texture maps. In *ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS)*, pages 7–15, 2002.

[Kel97] Alexander Keller. Instant radiosity. In *Proceedings of SIGGRAPH*, pages 49–56, 1997.

[KH84] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *SIGGRAPH: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 1984.

[KH01] Alexander Keller and Wolfgang Heidrich. Interleaved sampling. In *Proceedings of EGWR*, pages 269–276, 2001.

[KH05] Joshua Krall and Cody Harrington. Modeling and rendering of clouds on "stealth". In *SIGGRAPH Sketch*, 2005. http://portal.acm.org/ft_gateway.cfm?id=1187214.

[Khr] Group Khronos. Opengl official website. http://www.opengl.org.

[KHW07] Aaron Knoll, Charles D Hansen, and Ingo Wald. Coherent Multiresolution Isosurface Ray Tracing. Technical Report UUSCI-2007-001, 2007. (submitted for publication).

[Kir86] D B Kirk. The simulation of natural features using cone tracing. In *Proceedings of Computer Graphics Tokyo '86 on Advanced Computer Graphics*, pages 129–144, 1986.

[Kir09] David Kirk. The CUDA hardware model. courses.ece.uiuc.edu/ece498/al/lectures/lecture8-9-hardware.ppt, 2009.

[Kis98] Gokhan Kisacikoglu. 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.

[KK89] J. T. Kajiya and T. L. Kay. Rendering fur with three dimensional textures. In *SIGGRAPH*, pages 271–280, 1989.

[KKH02] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

[Kno08] Aaron Knoll. A survey of octree volume rendering methods. 2008.

[KPHE02] Joe Kniss, Simon Premoze, Charles Hansen, and David Ebert. Interactive translucent volume rendering and procedural modeling. In *VIS: Proceedings of the conference on Visualization*, pages 109–116, 2002.

[KS01] James T. Klosowski and Cláudio T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7:365–379, October 2001.

[KTI$^+$01] T. Kaneko, T. Takahei, M. Inami, N. Kawakami, Y. Yanagida, T. Maeda, and S. Tachi. Detailed shape representation with parallax mapping. In *In Proceedings of ICAT*, pages 205–208, 2001.

[KW03a] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 2003.

[KW03b] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization*, 2003.

[KW05] Jens Krüger and Rüdiger Westermann. GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Computer Graphics Forum*, 24(3), 2005.

[KWAH06] Ralf Kaehler, John Wise, Tom Abel, and Hans-christian Hege. Abstract GPU-Assisted Raycasting for Cosmological Adaptive Mesh Refinement Simulations. In *Eurographics / IEEE VGTC Workshop on Volume Graphics*, pages 103 – 110, 2006.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. In *SIGGRAPH : Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, 1987.

[LD07] Sylvain Lefebvre and Carsten Dachsbacher. Tiletrees. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, 2007.

[LDN04] Sylvain Lefebvre, Jerome Darbon, and Fabrice Neyret. Unified texture management for arbitrary meshes. Technical Report RR5210-, INRIA, May 2004. http://www-evasion.imag.fr/Publications/2004/LDN04.

[Lev88] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8:29–37, May 1988.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9:245–261, July 1990.

[LGS+09] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH*, pages 769–776, 2004.

[LH06] Sylvain Lefebvre and Hugues Hoppe. Perfect spatial hashing. In *SIGGRAPH*, pages 579–588, 2006.

[LHJ99] Eric LaMar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proceedings of Visualization (VIS)*, pages 355–361, 1999.

[LHN05a] 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", pages 595–613. Addison Wesley, 2005.

[LHN05b] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Texture sprites: Texture elements splatted on surfaces. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)*, April 2005.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, pages 55–63, 2010.

[LKHW05] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, and Ross T. Whitaker. A streaming narrow-band algorithm: interactive computation and visualization of level sets. In *ACM SIGGRAPH 2005 Courses*, 2005.

[LKS+06] Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, Efficient, Random-Access GPU Data Structures. *ACM Transactions on Graphics*, 25(1), 2006.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH*, pages 451–458, 1994.

[LLY06] Patric Ljung, Claes Lundström, and Anders Ynnerman. Multiresolution interblock interpolation in direct volume rendering. In *Eurographics/IEEE-VGTC Symposium on Visualization*, 2006.

[LMK03] Wei Li, Klaus Mueller, and Arie Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of IEEE Visualization (VIS)*, page 42, 2003.

[LN03] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics (I3D)*. ACM, 2003.

[LSK+06] Aaron E. Lefohn, Shubhabrata Sengupta, Joe Kniss, Robert Strzodka, and John D. Owens. Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics*, 25(1), 2006.

[LSK$^+$07] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of EGSR*, pages 277–286, 2007.

[LZT$^+$08] Jaakko Lehtinen, Matthias Zwicker, Emmanuel Turquin, Janne Kontkanen, Frédo Durand, François Sillion, and Timo Aila. A meshless hierarchical representation for light transport. In *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2008)*, volume 27, 2008.

[Max86] Nelson Max. Light diffusion through clouds and haze. *Computer Vision, Graphics, and Image Processing*, 33(3), 1986.

[Max94] Nelson L. Max. Efficient Light Propagation for Multiple Anisotropic Volume Scattering. In *Fifth Eurographics Workshop on Rendering*, 1994.

[Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

[MBW08] Oliver Mattausch, Jiří Bittner, and Michael Wimmer. Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings Eurographics 2008)*, 27(2):221–230, April 2008.

[MHR02] K. Engel M. Hadwiger, J.M. Kniss and C. Rezksalama. High-quality volume graphics on consumer pc hardware. In *SIGGRAPH, Course Notes 42*, 2002.

[Mic11] Microsoft. Official DirectX developer website and technical documentation. http://msdn.microsoft.com/en-us/directx, 2011.

[Mil94] Gavin S. P. Miller. Efficient algorithms for local and global accessibility shading. In *SIGGRAPH*, pages 319–326, 1994.

[Mit07] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA, 2007. ACM.

[MMMY97] Torsten Möller, Raghu Machiraju, Klaus Mueller, and Roni Yagel. A comparison of normal estimation schemes. In *Proceedings of VIS (IEEE Conference on Visualization)*, pages 19–, 1997.

[MN00] A. Meyer and F. Neyret. Multiscale shaders for the efficient realistic rendering of pine-trees. In *Proceedings of GI (Graphics Interface)*, 2000.

[NB04] S. Nirenstein and E. Blake. Hardware accelerated aggressive visibility preprocessing using adaptive sampling. In *Rendering Techniques 2004: Proceedings of the 15th symposium on Rendering*, pages 207–216, 2004.

[NBS06] Diego Nehab, Joshua Barczak, and Pedro V. Sander. Triangle order optimization for graphics hardware computation culling. In *ACM SIGGRAPH 2006 Sketches*, 2006.

[Ney95] Fabrice Neyret. Animated texels. In *Computer Animation and Simulation '95*, pages 97–103. Eurographics, September 1995. ISBN 3-211-82738-2.

[Ney98] Fabrice Neyret. Modeling, animating, and rendering complex scenes using volumetric textures. *Visualization and Computer Graphics, IEEE Transactions on*, 4(1):55 –70, 1998.

[Ney03] Fabrice Neyret. Advected textures. In *ACM-SIGGRAPH/EG Symposium on Computer Animation (SCA)*, july 2003.

[Ney06] Fabrice Neyret. Créer, simuler, explorer des univers naturels sur ordinateur. http://www-evasion.imag.fr/Publications/2006/Ney06, 2006. invited paper.

[NSW09] Greg Nichols, Jeremy Shopf, and Chris Wyman. Hierarchical image-space radiosity for interactive global illumination. *Computer Graphics Forum (Proc. EGSR)*, 28(4), 2009.

[NVIa] NVIDIA. NVIDIA developer web site. http://developer.nvidia.com.

[NVIb] NVIDIA. NVIDIA GeForce 8800 Technical Brief. http://www.nvidia.com/page/8800_tech_briefs.html.

[NVI09] NVIDIA. Alternative rendering pipelines on nvidia cuda, 2009.

[NVI11a] NVIDIA. CUDA Programming Guide 4.0. http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/NVIDIA_CUDA_Programming_Guide_4.0.pdf, 2011.

[NVI11b] NVIDIA. *CUDA Programming Guide 4.0*. 2011.

[NVI11c] NVIDIA. Optix interactive ray-tracing engine for the gpu, 2011.

[Nyq28] H. Nyquist. Certain topics in telegraph transmission theory. *American Institute of Electrical Engineers, Transactions of the*, 47(2):617 –644, 1928.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *SIGGRAPH*, pages 359–368, 2000.

[Oli08] Jon Olick. Next generation parallelism in games. In *ACM SIGGRAPH 2008 classes*, pages 21:1–21:89, 2008.

[OM90] Masataka Ohta and Mamoru Maekawa. Ray-bound tracing for perfect and efficient anti-aliasing. *The Visual Computer*, 6:125–133, May 1990.

[ON97] M. Olano and M. North. Normal distribution mapping. *Univ. of North Carolina Computer Science Technical Report 97-041*, 1997.

[PBFJ05] Serban D. Porumbescu, Brian Budge, Louis Feng, and Kenneth I. Joy. Shell maps. In *SIGGRAPH*, pages 626–633, 2005.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. (Proceedings of ACM SIGGRAPH 2002).

[Pea85] Darwyn R. Peachey. Solid texturing of complex surfaces. In *Proceedings of SIGGRAPH '85*, pages 279–286, 1985.

[Per85a] Ken Perlin. An image synthesizer. In *Proceedings of SIGGRAPH '85*, pages 287–296, 1985.

[Per85b] Ken Perlin. An image synthesizer. In *SIGGRAPH*, pages 287–296, 1985.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2*. 2005.

[PH89] K. Perlin and E. M. Hoffert. Hypertexture. In *SIGGRAPH*, pages 253–262, 1989.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[PKGH] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. In *Proceedings of SIGGRAPH '97*.

[PN01] Ken Perlin and Fabrice Neyret. Flow noise. In *SIGGRAPH Sketch*, page 187, Aug 2001.

[PO07] Fabio Policarpo and Manuel Oliveira. *GPU Gems 3*, chapter 18: "Relaxed Cone Stepping For Relief Mapping". Addison-Wesley, 2007.

[Pur04] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, 2004. Adviser-Hanrahan, Patrick M.

[RE02] Stefan Roettger and Thomas Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *VVS : Proceedings of the IEEE symposium on Volume visualization and graphics*, pages 23–28, 2002.

[REG+09] Tobias Ritschel, Thomas Engelhardt, Thorsten Grosch, Hans-Peter Seidel, Jan Kautz, and Carsten Dachsbacher. Micro-rendering for scalable, parallel final gathering. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 28(5), 2009.

[RGK+08] Tobias Ritschel, Thorsten Grosch, Min H. Kim, Hans-Peter Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 27(5), 2008.

[RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. Approximating dynamic global illumination in image space. In *Proceedings of I3D*, pages 75–82, February 2009.

[RGW+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*, 2003.

[RKE00] S. Rottger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In Proceedings of IEEE Visualization., 2000.

[RMMD04] Alex Reche-Martinez, Ignacio Martin, and George Drettakis. Volumetric reconstruction and interactive rendering of trees from photographs. In *SIGGRAPH proceedings*, pages 720–727, 2004.

[RPV93] H. Rushmeier, C. Patterson, and A. Veerasamy. Geometric simplification for indirect illumination calculations. In *Proceedings of Graphics Interface*, pages 227–236, 1993.

[RSEB+00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware (HWWS)*, 2000.

[SA08] Erik Sintorn and Ulf Assarsson. Fast parallel GPU-sorting using a hybrid algorithm. *Journal of Parallel Distributed Compututing*, 68:1381–1388, October 2008.

[Sam90] Hanan Samet. *Applications of spatial data structures: Computer graphics, image processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[SBS06] Dirk Staneker, Dirk Bartz, and Wolfgang Straßer. Occlusion-driven scene sorting for efficient culling. In *Afrigaph*, pages 99–106, 2006.

[Sch97] Andreas Schilling. Towards real-time photorealistic rendering: challenges and solutions. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 7–15, 1997.

[Sch05] Henning Scharsach. Advanced GPU raycasting. In *Central European Seminar on Computer Graphics*, pages 69–76, 2005.

[SD95] François Sillion and George Drettakis. Feature-based control of visibility error: A multi-resolution clustering algorithm for global illumination. In *Proceedings of SIGGRAPH*, volume 29, pages 145–152, August 1995.

[SEA08] Erik Sintorn, Elmar Eisemann, and Ulf Assarsson. Sample based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum (Proc. EGSR)*, 27(4):1285–1292, June 2008.

[SFH97] Rajagopalan Srinivasan, Shiaofen Fang, and Su Huang. Volume rendering by template-based octree projection, 1997.

[SGG08] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[Shp11] Shpagin, Andrew. 3d-coat : A voxel scultping software, 2011.

[SIP06] Benjamin Segovia, Jean-Claude Iehl, and Bernard Peroche. Bidirectional instant radiosity. In *Proceedings of EGSR*, June 2006.

[SjCC+02] Claudio Silva, Yi jen Chiang, Wagner Correa, Jihad El-sana, and Peter Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *In Visualization'02 Course Notes*, 2002.

[SKS02] Peter-Pike J. Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 21(3):527–536, 2002.

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

[Spe08] Scott Spencer. *ZBrush Character Creation: Advanced Digital Sculpting*. SYBEX Inc., pap/dvdr edition, 2008.

[SS10] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on GPUs. In *ACM SIGGRAPH Asia 2010 papers*, pages 179:1–179:10, 2010.

[SSKE05] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Volume Graphics, 2005. Fourth International Workshop on*, pages 187–241, June 2005.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-D shapes. *Computer Graphics (Proc. SIGGRAPH)*, 24(4):197–206, 1990.

[STN87] Mikio Shinya, T. Takahashi, and Seiichiro Naito. Principles and applications of pencil tracing. In *Proceedings of SIGGRAPH '87*, pages 45–54, 1987.

[Tan08] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 3. edition, 2008.

[TL04] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. In *Proceedings of SIGGRAPH*, pages 469–476, 2004.

[TLQ$^+$] Ping Tan, Stephen Lin, Long Quan, Baining Guo, and Heung-Yeung Shum. Multiresolution reflectance filtering. In *EGSR'05 Rendering Techniques*, pages 111–116.

[TMJ98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, 1998.

[TNF89] D. Thomas, Arun N. Netravali, and D.S. Fox. Anti-aliased ray tracing with covers. *Computer Graphics Forum*, 8(4):325–336, 1989.

[Tok05] Michael Toksvig. Mipmapping normal maps. *journal of graphics, gpu, and game tools*, 10(3):65–71, 2005.

[TS05] Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Technical report, 2005.

[TWTT99] X Tong, WP Wang, WW Tsang, and Z Tang. Efficiently rendering large volume data using texture mapping hardware. In *IEEE TVCG Symposium on Visualization Proceedings*, 1999.

[TZL$^+$02] Xin Tong, Jingdan Zhang, Ligang Liu, Xi Wang, Baining Guo, and Heung-Yeung Shum. Synthesis of bidirectional texture functions on arbitrary surfaces. *ACM Transactions on Graphics*, 21(3):665–672, 2002. (Proceedings of ACM SIGGRAPH 2002).

[VSE06] J. E. Vollrath, T. Schafhitzel, and T. Ertl. Employing Complex GPU Data Structures for the Interactive Visualization of Adaptive Mesh Refinement Data. In *Proceedings of Eurographics / IEEE VGTC Workshop on Volume Graphics*, 2006.

[Wal04] Ingo Wald. Realtime Ray Tracing and Interactive Global Illumination. *PhD thesis, Saarland University*, 2004.

[WB05] Michael Wimmer and Jiří Bittner. *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter "Hardware Occlusion Queries Made Useful", pages 91–108. 2005.

[WBB$^+$07] Michael Wand, Alexander Berner, Martin Bokeloh, Arno Fleck, Mark Hoffmann, Philipp Jenke, Benjamin Maier, Dirk Staneker, and Andreas Schilling. Interactive editing of large point clouds. In *Symposium on Point-Based Graphics 2007 : Eurographics / IEEE VGTC Symposium Proceedings*, pages 37–46, 2007.

[WBZC$^+$10] Magnus Wrenninge, Nafees Bin Zafar, Jeff Clifford, Gavin Graham, Devon Penney, Janne Kontkanen, Jerry Tessendorf, and Andrew Clinton. Volumetric methods in visual effects. In *Course Notes - SIGGRAPH*, 2010.

[WDS05] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses*, 2005.

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH '98*, pages 169–177, 1998.

[Wes90] Lee Westover. Footprint evaluation for volume rendering. In *Proceedings of SIGGRAPH '90*, pages 367–376, 1990.

[WFA+05] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 24(3):1098–1107, 2005.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23:343–349, June 1980.

[Whi09] Daniel White. The real 3d mandelbulb, 2009.

[Wik] Wikipedia. Z-buffering algorithm description. http://en.wikipedia.org/wiki/Z-buffering.

[Wik11] Wikipedia. Amdahl's law, 2011.

[Wil83] Lance Williams. Pyramidal parametrics. In *Proceedings of SIGGRAPH '83*, pages 1–11, 1983.

[Wil05] Donnelly William. *GPU Gems 2*, chapter "Per-Pixel Displacement Mapping with Distances Functions", pages 123–136. 2005.

[WKB+02] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive global illumination using fast ray-tracing. In *Proceedings of EGRW*, pages 15–24, 2002.

[WM92] Peter L. Williams and Nelson Max. A volume density optical model. In *VVS: Proceedings of the workshop on Volume visualization*, 1992.

[WMG+07] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In *STAR Proceedings of Eurographics 2007*, pages 89–116, September 2007.

[WPSAM10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235 –246, march 2010.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive distributed ray tracing of highly complex models. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 277–288, 2001.

[WWH+00] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmermann, and Thomas Ertl. Level-of-detail volume rendering via 3d textures. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 7–13, 2000.

[WWS01] Michael Wimmer, Peter Wonka, and François X. Sillion. Point-based impostors for real-time visualization. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 163–176, 2001.

[WZF+03] Xiaoming Wei, Ye Zhao, Zhe Fan, Wei Li, Suzanne Yoakum-Stover, and Arie Kaufman. Blowing in the wind. In *SCA: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 75–85, 2003.

[YCK+09] I. Yu, A. Cox, M. H. Kim, T. Ritschel, T. Grosch, C. Dachsbacher, and J. Kautz. Perceptual influence of approximate visibility in indirect illumination. In *ACM Transactions on Applied Perception (Proc. APGV)*, pages 24:1–24:14, 2009.

[YS93] Roni Yagel and Zhouhong Shi. Accelerating volume animation by space-leaping. In *Proceedings of the 4th conference on Visualization '93*, pages 62–69, 1993.

[ZHR+09] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on gpus. In *ACM SIGGRAPH Asia 2009 papers*, pages 155:1–155:11, 2009.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff, III. Visibility culling using hierarchical occlusion maps. In *Proceedings of SIGGRAPH '97*, pages 77–88, 1997.

[ZTTS06] G. Ziegler, A. Tevs, C. Theobalt, and H.-P. Seidel. GPU point list generation through histogram pyramids. In *Proceedings of VMV*, pages 137–141, 2006.