Accepted at Pacific Graphics 2011
Bing-Yu Chen, Jan Kautz, Tong-Yee Lee, and Ming C. Lin
Authors' version (The definitive version is available at http://diglib.eg.org/ and http://onlinelibrary.wiley.com/)

Volume 30 (2011), Number 7

Interactive Indirect Illumination Using Voxel Cone Tracing

Cyril Crassin[1,2] Fabrice Neyret[1,3] Miguel Sainz[4] Simon Green[4] Elmar Eisemann[5]

[1] INRIA Rhone-Alpes & LJK [2] Grenoble University [3] CNRS [4] NVIDIA Corporation [4] Telecom ParisTech

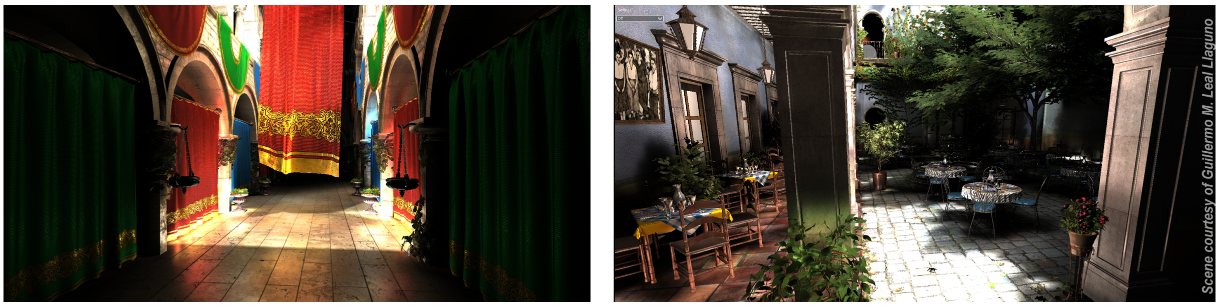


Figure 1: *Real-time indirect illumination (25-70 fps on a GTX480): We rely on a voxel-based cone tracing to ensure efficient integration of 2-bounce illumination and support diffuse and glossy materials on complex scenes.* (Right scene courtesy of G. M. Leal Llaguno)

Abstract

Indirect illumination is an important element for realistic image synthesis, but its computation is expensive and highly dependent on the complexity of the scene and of the BRDF of the involved surfaces. 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 a hierarchical voxel octree representation generated and updated on the fly from a regular scene mesh coupled with an approximate voxel cone tracing that allows for a fast estimation of the visibility and incoming energy. Our approach can manage two light bounces for both Lambertian and glossy materials at interactive framerates (25-70FPS). It exhibits an almost scene-independent performance and can handle complex scenes with dynamic content thanks to an interactive octree-voxelization scheme. In addition, we demonstrate that our voxel cone tracing can be used to efficiently estimate Ambient Occlusion.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: global illumination, indirect lighting, final gather, cone-tracing, voxels, GPU, real-time rendering

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. Secondly, it requires integrating lighting information over a large number of directions for each shaded point. Nowadays, with the complexity of 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 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 paper, we present a novel algorithm that computes two light bounces (using final gathering [Jen96]) and is entirely implemented on the GPU. It does not suffer from noise or temporal discontinuities, avoids costly precomputation steps, and is not restricted to low-frequency illumination. It exhibits almost scene-independent performance because we manage to mostly avoid involving the actual mesh in our computations. Further, our solution can be extended to out-of-core rendering, hereby handling arbitrarily complex scenes. 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 a pre-filtered hierarchical voxel representation of the scene geometry. For efficiency, this representation is stored on the GPU in the form of a dynamic sparse voxel octree [CNLE09, LK10] generated from the triangle meshes. 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 technique. 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. 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).

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 together with a fast GPU-based mesh voxelization and octree-building algorithm;
- An efficient splatting scheme to inject and filter incoming radiance information into our voxel structure;
- An efficient approximate cone-tracing integration;

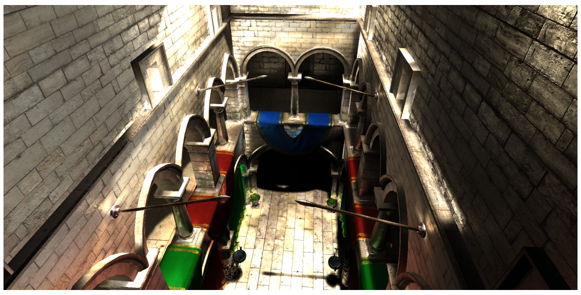


Figure 2: *Our method supports diffuse and glossy reflections.*

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, CB04], or hierarchical scene structures [WZPB09], 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. Recent GPU implementations of photon mapping [Hac05, WZPB09] use clustering and exploit the spatial coherence of illumination to reduce the sampling cost of final gathering. These approaches allow multiple bounce global illumination, but they still do not reach real-time performances, and suffer from temporal flickering artifacts.

To achieve higher framerates, the light transport is often discretized [DK09]. 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-like solutions [RGK*08, REG*09, HREB11, RHK*11] 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.

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]. Thiedemann et al. [THGM11] rely on a regular voxel grid to speed-up ray-tracing used for computing visibility with VPLs stored inside a *reflective shadow map* (RSM) [DS05]. It offers real-time performances but only for near-field single bounce indirect illumination and diffuse materials. It is restricted to low-resolution voxel grids, and suffers from classical precision problems of RSMs, as well as noise and flickering artifacts coming from the limited number of rays that can be used in real-time to perform final gathering. Our approach does not suffer from such artifacts and always produce smooth temporally coherent results, thanks to the use of our voxel cone-tracing.

A near-interactive approach has been proposed in [LWDB10] to render glossy reflections with indirect highlights, caused by two successive reflections on glossy surfaces. But this solution is not fast enough to provide real-time performances. To support indirect highlights, our approach uses a simpler, less precise, Gaussian lobe representation to encode a NDF *(Normal Distribution Function)* and a distribution of light directions, that can be convolved with a set of pre-defined BRDFs to reconstruct the radiance response.

Our work derives a hierarchical representation of the scene that produces a regular structure to facilitate light transfer and achieve real-time performance. This structure is similar as the one in [KD10], where diffuse indirect il-

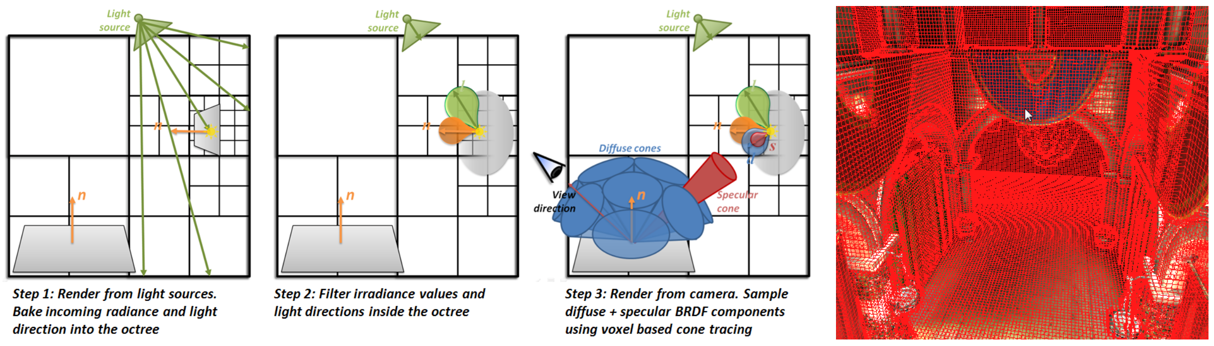

Light source
n
Step 1: Render from light sources. Bake incoming radiance and light direction into the octree
Light source
n
Step 2: Filter irradiance values and light directions inside the octree
Light source
Diffuse cones
View direction
n
Specular cone
s
d
Step 3: Render from camera. Sample diffuse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

Figure 3: *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.*

lumination is produced via a diffusion process inside a set of nested regular voxel grids. While relatively fast, this approach suffers from a lack of precision and temporal discontinuities coming from the relatively low resolution of the usable voxel grids. The resolution is limited by the cost of the diffusion process as well as memory occupancy. Consequently, only diffuse indirect illumination is possible, while our approach can manage both diffuse and specular indirect lighting (glossy refections with indirect highlights). This improvement is enabled by using ray tracing to collect the radiance stored in the structure and allows us to use a sparse storage of the incoming radiance and scene occlusion information, while maintaining high precision.

3. Algorithm overview

Our approach is based on a three-step algorithm as detailed in Fig. 3. 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 into the higher levels of the octree (mipmap). We rely on a compact Gaussian-Lobes representation to store the filtered distribution of incoming light directions, which is done efficiently in parallel by relying on a screen-space quad-tree analysis. Our voxel filtering scheme also treats the NDF *(Normal Distribution Function)* 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 BRDFs, 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.

4. Our hierarchical voxel structure

The core of our approach is built around a pre-filtered hierarchical voxel version of the scene geometry. For efficiency, this representation is stored in the form of a sparse voxel octree in the spirit of [LK10] and [CNLE09].

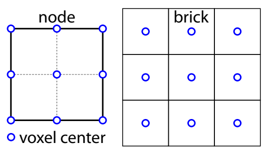
Having a hierarchical structure allows us to avoid using the actual geometric mesh and leads to indirect illumination in arbitrary scenes at an almost geometry-independent cost. It is possible to improve precision near the observer and to abstract energy and occupancy information farther away. We reach 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, contrary to path-tracing or photon mapping [Jen96], our approach always gives smooth results.

4.1. Structure description

The root node of the tree represents the entire scene. Its children each represent an eighth of the parent's volume and so forth. Therefore, we will also sometimes associate a node directly to a bounding box of the volume in space that it represents. This structure allows us to query filtered scene information (energy intensity and direction, occlusion, and local normal distribution functions - NDFs) with increasing precision by descending the tree hierarchy. Through this adaptivity, we handle large and complex scenes.

GPU representation Our sparse voxel octree is a very compact pointer-based structure, similar to [CNLE09]. Octree nodes are stored in linear GPU memory and nodes are grouped into $2\times2\times2$ tiles. This grouping allows us to store a single pointer to all $2\times2\times2$ children. Further, each node contains a pointer to a small voxel volume, a *brick*, stored in texture memory. This texture approximates the scene section represented by the node.The representation is sparse, meaning that empty nodes are collapsed to gain memory. Since we use a brick instead of a single value per node, we can use hardware texture trilinear interpolation to interpolate values.

Our representation has some differences compared to [CNLE09]. When voxelizing a mesh-based scene, most of the volume remains empty which implies that small bricks are more efficient in a sparse octree. However, the brick representation proposed by [CNLE09] is not optimal in case of small bricks, due to the need of a one voxel border duplicating neighboring voxels to ensure a correct inter-brick interpolation. This border becomes very expensive in terms of storage when small bricks are used.



To solve this problem, we propose to use $3 \times 3 \times 3$ voxel bricks, but assume that the voxel centers are located at the node corners instead of the node centers (Fig. left). This ensures that interpolated values can always be computed inside a brick covering a set of $2 \times 2 \times 2$ nodes. Redundancy is not eliminated, but we use less than half the amount of memory compared to adding a boundary, for the same sampling precision.

This memory reduction allows us to add neighbor pointers to the structure which enable us to quickly visit spatially neighboring nodes and the parent. We will see that these links are particularly important to efficiently distribute the direct illumination over all levels of the tree.

4.2. Interactive voxel hierarchy construction and dynamic updates

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. It must be fast enough to be performed in each frame to be able to handle 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 constructs the octree directly. To speed-up the process, we observe that, in many scenes, large parts of the environment are usually static or updated locally upon user interaction. Consequently, most updates are restricted and only applied when needed, but fully-dynamic objects need a per-frame voxelization.

Both semi-static and fully dynamic objects are stored in the same octree structure for an easy traversal and a unified filtering. A time-stamp mechanism is used to differentiate both types, in order to prevent semi-static parts of the scene to be destructed in each frame. Our structure construction algorithm performs in two steps: octree building and MIP-mapping of the values.

4.2.1. Octree building

We first create the octree structure itself by using the GPU rasterization pipeline. We rasterize the mesh three times, along the three main axis 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, we generate at least one fragment shader thread for each potentially-existing leaf node in the tree. Each thread traverse the octree from top-to-bottom and directly subdivide it when needed. Once the correct leaf node is found the surface attributes (typically texture color, normal and material) are written.

Whenever a node needs to be subdivided, a set of $2 \times 2 \times 2$ sub-nodes are "allocated" inside a global shared *node buffer* which is pre-allocated in video memory. The address of this set of new sub-nodes is written to the "child" pointer of the subdivided node and the thread continue its descent down to the leaf. These allocations are made using the atomic increment of a global shared counter, which indicates the next available page of nodes in the shared node buffer. Since, in a massively parallel environment, multiple threads can request the subdivision of the same node at the same time an incoherent structure could result. Such conflicts are prevented by a per-node mutex that ensures that the subdivision is only performed by the first thread. Unfortunately, it is not possible to put the other threads asleep 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) and their values written in the tree. Such deferred passes can possibly generate new deferred threads and are re-executed as long as the global thread list is not empty. Values in the leaves are written directly inside the brick associated to 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 provides CUDA-like video memory pointers as well as atomic operations directly inside OpenGL shaders.

4.2.2. Dynamic update

Dynamic update of the octree structure for animated objects is done with the same rasterization-based building algorithm we just described. Animated objects are rasterized every frame, the only difference is that new voxels generated by these objects cannot overwrite existing static parts of the structure. To prevent overwriting static octree nodes and bricks, and allow for a fast clearing every frame, these elements are stored at the end of the buffers.

4.2.3. MIP-mapping

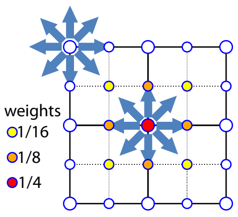


Figure 4: *Lower-level voxels surround higher-level voxels. During MIP-mapping, shared voxels are "evenly distributed" resulting in gaussian weights.*

Once the octree structure is built and the surface values written inside the leafs, these values must be MIP-mapped and filtered in the inner nodes of the tree. 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 (see Sec. 4.1), each node contains a 3^3-voxel brick, whose boundary reappears in neighboring bricks. Consequently, when computing the filtered data, one has to weight each voxel with the inverse of its multiplicity (Fig. 4). In practice, this results in a 3^3-Gaussian weighting kernel, which, for our case, is an optimal reconstruction filter [FP02].

4.3. Voxel representation

Each voxel at a given level must represent the light behavior of the lower levels - and thus, of the whole scene span it represents. For this, we model the directional information with distributions that describe the underlying data. We apply this to normals and to light directions. This is more accurate than single values, as shown in [HSRG07]. 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. Following [Tok05], to ease the interpolation, the variance is encoded via the norm $|D|$ such that $\sigma^2 = \frac{1-|D|}{|D|}$. In Sec. 7, we will detail how to calculate light interaction with such a data representation.

We also estimate occlusion information, in form of visibility (percentage of blocked rays), and not volumetric density. We choose to store a single average value to keep voxel data compact, leading to a lack of view-dependency that is a disadvantage for large thin objects. Material color is actually represented as an opacity weighted color value (alpha pre-multiplied) for better interpolation and integration during the rendering stage (Sec. 7). Normal information is also pre-multiplied by the opacity value in order to correctly take its visibility into account.

5. Approximate Voxel Cone Tracing

Global illumination generally requires many sampling rays to be tested against the scene, which is expensive. These rays are spatially and directionally coherent which is a property that many approaches such as packet ray-tracing [WMG*] use. Similarly, in order for us to exploit this coherence, we introduce a voxel cone tracing method that was inspired by filtering for anti-aliasing [CNLE09].

While the original cone tracing technique is complex and often expensive, we can use our filtered voxel structure to approximate the result for all rays in the same bundle in parallel. The idea is to step along the cone axis and perform lookups in our hierarchical representation at the level corresponding to the cone radius (Fig. 5). During this step, we use quadrilinear interpolation to ensure a smooth variation that does not exhibit aliasing. Because of the way we pre-filtered the voxel data, this approximation produces visual plausible results although it might differ from true cone tracing.

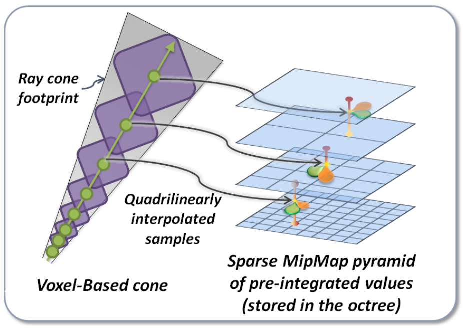

Ray cone footprint
Quadrilinearly interpolated samples
Voxel-Based cone
Sparse MipMap pyramid of pre-integrated values (stored in the octree)

Figure 5: *Voxel-based cone tracing using pre-filtered geometry and lighting information from a sparse MIP-map pyramid (stored in a voxel octree structure).*

While stepping along the cone, we use the classical emission-absorption optical model described in [Max95, EHK*04] to accumulate the values along the cone. We keep track of occlusion α and, potentially, a color value c representing the reflected light towards the cone origin. In each step, we retrieve the appropriately filtered scene information and the occlusion value α_2 from the octree to compute a new outgoing radiance c_2 (Sec. 7). We then update the values using the classical volumetric front-to-back accumulation: $c := \alpha c + (1-\alpha)\alpha 2 c_2$ and $\alpha = \alpha + (1-\alpha)\alpha 2$. Also, to ensure good integration quality, even for large cones, the distance d' between successive sample locations along a ray does not coincide with the current voxel size d. To account for the smaller step size, we use the correction $\alpha_s' = 1-(1-\alpha_s)^{\frac{d'}{d}}$.

6. Ambient Occlusion

To illustrate the use of our approximate cone tracing 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, else 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 $\alpha_f(p+r\omega) := f(r)\alpha(p+r\vec{\omega})$.

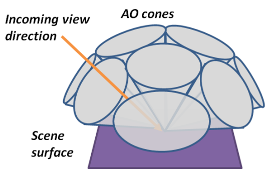


Figure 6: *Ambient Occlusion with a small set of voxel cones.*

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 approximate their contribution with our voxel-based cone tracing, as illustrated in Fig. 6, left. 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.

7. Voxel Shading

For indirect illumination, we will be interested not only in occlusion, but need to compute the shading of a voxel. For this, we have to account for the variations in the embedded directions and scalar attributes, as well as the span of the cone that is currently accumulating that voxel. As shown in [Fou92,HSRG07], this can conveniently be translated into convolutions, provided that the elements are decomposed in lobe shapes. In our case, we have to convolve the BRDF, the NDF, and the span of the view cone, the first already being represented as Gaussian lobes. 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. 7), 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.

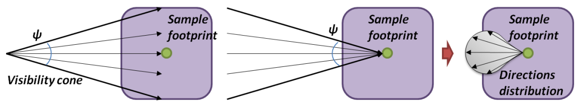


Figure 7: *Illustration of the direction distribution computed on a filtered surface volume from an incident cone.*

8. Indirect Illumination

To compute indirect illumination in the presence of a point light is more involved than AO. We use on 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.

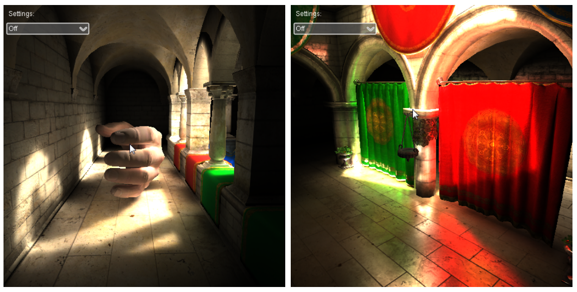


Figure 8: *Left: Indirect diffuse lighting with animated object. Right: Glossy reflections and color bleeding*

8.1. 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 Sec. 6 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. 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.

8.2. Capturing Direct Illumination

To complete our indirect-illumination algorithm, we finally need to describe how to store incoming radiance. Inspired by [DS05], we render the scene from the light's view using standard rasterization to 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.

To splat the photons, we basically use a fragment shader with one thread per light-view-map pixel. 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. 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 Sec. 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 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, only left), each thread will add voxel data from the current node 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.

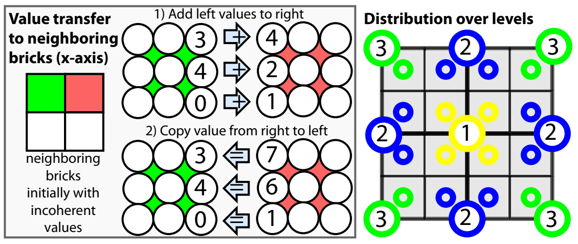

Value transfer to neighboring bricks (x-axis)
neighboring bricks initially with incoherent values
1) Add left values to right
3
4
0
4
2
1
2) Copy value from right to left
3
4
0
7
6
1
Distribution over levels
3
2
3
2
1
2
3
2
3

Figure 9: *Left: After photon splatting, an addition and copy along each axis is used to correct inconsistencies on for duplicated voxels at bricks boundaries. 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);*

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 (MIP-map). 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, 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). 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 leafs 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 much resources: During the filtering, we launched threads for all nodes, even those that did not contain any photon, where filtering was applied to zero values. Direct light often only affects a small part of the scene, so avoiding filtering of zero values is crucial. One solution is to launch a thread per light-view-map pixel, find the node in the level on which the filtering should be applied and execute it. The result would be correct, but whenever threads end up in the same node (which would happen very often in high levels nodes), work would be performed multiple times.

Our idea to reduce the number of threads, and get close to the optimal thread set for each filtering pass, is to rely on a 2D *node map*, derived from the light-view map. This node-map is in-fact a MIP-map pyramid. The pixels of the lowest node-map level (highest resolution) store the indices of the 3D leaf nodes containing the corresponding photon of the light-view map. Higher-level node-map pixels (lower resolution) store the index of the lowest common ancestor node for the preceding nodes of the previous level (Fig.10).

We still launch one thread per pixel of the lowest node-map to compute each MIP-mapping pass in our octree structure. But when a thread is going down in the tree, looking for the node that it must compute the MIP-mapped value for, it first looks in the current level of the node map to verify whether there is no common-ancestor with another thread. If there is, all threads detecting this common-ancestor are terminated except the upper left one, which is then in charge of computing the remaining filtered values. This strategy is quite successful and we obtain > 2 speedup compared to not using our heuristic to terminate threads.

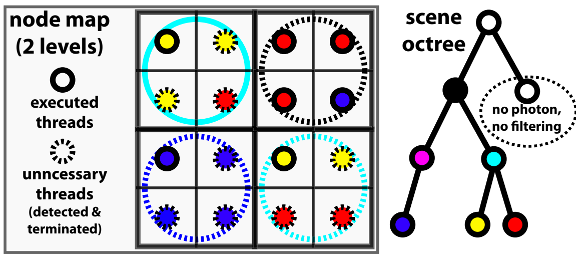


Figure 10: *The node map (left) is constructed from the light-view map. It is hierarchical (like a MIP-map) and has several levels (large circles represent the next level). On the lowest level, a pixel contains the index of the node in which the corresponding photon is located. Higher levels contain the common lowest ancestor of all underlying nodes. This structure allows to reduce the number of threads launched during the filtering of the photons (dashed are stopped).*

9. Anisotropic voxels for improved cone-tracing

While the voxel representation as described up-to-now already provides a very good approximation of the visibility when tracing cones, some quality problems remain. The first problem is known as the two red-green wall problem and is illustrated in Fig. 11. It is linked to averaging of values in the octree to a pre-integrated visibility value. Whenever two opaque voxels with different colors (or any other value) -coming from two plain walls for instance- are averaged in the upper level of the octree, their colors are mixed as if the two walls were semi-transparent. The same problem occurs for opacity, when a set of 2x2x2 voxels is half filled with opaque voxels and half filled with fully transparent ones, the resulting averaged voxel will be half-transparent. In our application, this under-estimation of the opacity and the wrong mix of materials can cause undesirable light leaking for large cones.

For a better pre-integration of voxel values, we propose an anisotropic voxel representation that is constructed during the MIP-mapping process, when building or updating the octree with irradiance values. Instead of a single channel of non-directional values, voxels will store 6 channels of directional values, one per major direction. 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 (Fig. 11). At render time, the voxel value is retrieved by linearly interpolating the values from the three closest directions with respect to the view direction.

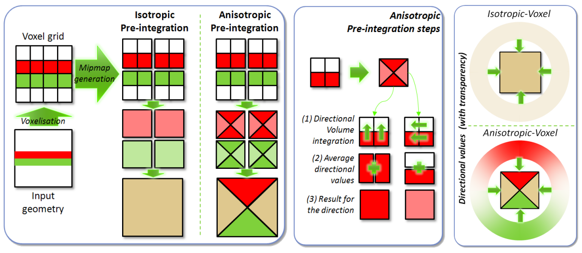

Voxel grid
Mipmap generation
Voxelisation
Input geometry
Isotropic Pre-integration
Anisotropic Pre-integration
Anisotropic Pre-integration steps
(1) Directional Volume integration
(2) Average directional values
(3) Result for the direction
Directional values (with transparency)
Isotropic-Voxel
Anisotropic-Voxel

Figure 11: *The left blue box illustrates the voxel MIP-mapping process without (left) and with (right) anisotropic voxels and directional pre-integration. Directional integration steps are illustrated in the middle blue box. The right box shows the result of directional sampling of a voxel.*

We chose this simple representation instead of a more complicated one because it can be fully interpolated and easily stored in our bricks. In addition, this directional representation only needs to be used for voxels that are not located in the last octree level (not at the full resolution). Hereby, storing directional values for all the properties only increases the memory consumption by 1.5x.

10. Results and Discussion

We implemented our approach on an NVIDIA GTX 480 system with an Intel Core 2 Duo E6850 CPU. Our solution delivers interactive to real-time framerates on various complex scenes as can be seen in the accompanying video. Table 1 shows the time per frame of each individual GI effects that our approach achieves for the Sponza scene (Fig. 1, 280K triangles) with a 9 levels octree (512^3 virtual resolution) and

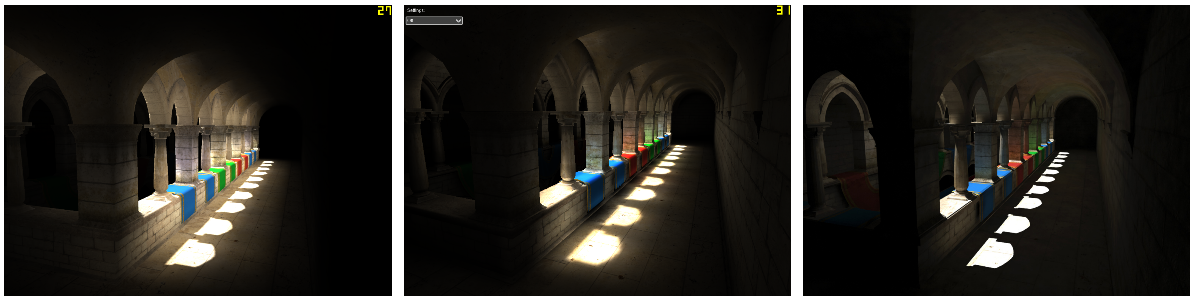


Figure 12: *Image quality and performance comparison between LPV (a, 37ms), our approach (b, 32ms) and ground truth (c, 14"12s) rendered with Mental Ray.*

a 512^2 screen 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. This time depends on the size and number of dynamic parts. The initial creation of the octree for the static environment takes approximatively 280ms. The light injection and filtering pass is computed only when the light source is moved, and takes approximately 16ms. On the sponza scene, with one dynamic object and a moving light, the average framerate is 30FPS with no specular cone traced, and 20FPS with one specular cone, for a 512^2 viewport. For a 1024x768 viewport (three diffuse and one specular cone), we get an average framerate of 11.4FPS.

Steps	Ras	Dir	Dif	Dir+dif	Spec	Total
Times	1.0	2.0	7.8	14.2	8.0	22.0

Table 1: *Timings (in ms) of individual effects : Mesh rasterisation, Direct lighting, Indirect diffuse, Total direct+indirect diffuse, indirect specular, direct+indirect diffuse and specular. Sponza scene, using 3 diffuse cones and one specular cone (10 deg. aperture)*

We compared the image quality of our approach (Fig.12(b)) with light propagation volumes (LPV) [KD10], the most closely-related real-time solution (Fig.12(a)), and a reference (Fig.12(c)) computed with Mental-Ray final gathering. We used NVIDIA's implementation of LPV in the Direct3D SDK. In all situations, our solution achieves higher visual quality (closer to the reference) and maintains better performance than LPV. Our approach can capture much more precise indirect illumination even far from the observer, with results close to ground truth in this case (cf. Fig. 12: Right wall and ceiling of the corridor, opposite exterior wall seen from left windows), while the LPV nested-grids representation prevents a correct capture of far indirect lighting. Both our approach and LPV suffer from light leaking due to the discrete representation of the geometry and irradiance, which can be observed on the interior side of the arches or the low wall on the left. Leaking appears very high for LPV, while it is more contained in our approach due to our higher voxel resolution.

One of our limitations is the relatively high memory consumption, even with our sparse structure, especially due to the support for indirect specularity. In practice, we allocate roughly 1024MB in the video memory. However, this consumption is usually lower than in [KD10] when trying to achieve comparable visual quality.

Cone aperture (deg)	10	20	30	60
AO	16.6	9.0	6.5	3.9

Table 2: *Timings in ms for the Sponza scene of the ambient occlusion computation for 3 cones and various cone apertures (512^2 resolution).*

Timings for ambient occlusion computations are shown in Table 2 and a comparison with a reference (path-tracing using NVIDIA OptiX) is shown Fig. 13. The differences are mainly due to the tested configuration using only 3 visibility cones. Wide cones suffer from a lack of precision for far visibility, that tends to overestimate occlusions.

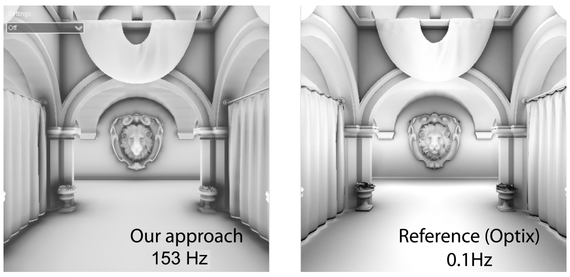


Figure 13: *Comparison with ground truth (OptiX) for ambient occlusion computation.*

11. Conclusion

We presented a novel real-time global-illumination algorithm using approximate voxel-based cone tracing to perform final gathering very efficiently. It is based on a sparse voxel octree structure encoding a pre-filtered representation of the scene geometry and incoming radiance. Using this adaptive representation, we are able to compute approximate indirect lighting in complex dynamic scenes. Our solution supports both diffuse and glossy effects with indirect highlights, and scales well when trading off quality against performance. In the future, we want to investigate making dynamic updates of the structure compatible with out-of-core caching schemes like [CNLE09], to overcome the memory consumption problem and allow arbitrary large and detailed scenes. We are also interested in new basis functions that we could use to provide higher precision distribution functions for better filtering of incoming radiance.

References

[AFO05] ARIKAN O., FORSYTH D. A., O'BRIEN J. F.: Fast and detailed approximate global illumination by irradiance decomposition. *ACM Trans. Graph. (Proc. SIGGRAPH) 24* (2005), 1108–1114.

[CB04] CHRISTENSEN P. H., BATALI D.: An irradiance atlas for global illumination in complex production scenes. In *Proc. of EGSR* (June 2004), pp. 133–141.

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. of I3D* (feb 2009), ACM, pp. 15–22.

[DK09] DACHSBACHER C., KAUTZ J.: Real-time global illumination for dynamic scenes. In *ACM SIGGRAPH 2009 Courses* (2009), ACM, pp. 19:1–19:217.

[DKTS07] DONG Z., KAUTZ J., THEOBALT C., SEIDEL H.-P.: Interactive global illumination using implicit visibility. In *Proc. of Pacific Graphics* (2007), pp. 77–86.

[DS05] DACHSBACHER C., STAMMINGER M.: Reflective shadow maps. In *Proc. of I3D* (2005), pp. 203–213.

[DSDD07] DACHSBACHER C., STAMMINGER M., DRETTAKIS G., DURAND F.: Implicit visibility and antiradiance for interactive global illumination. *ACM Trans. Graph. (Proc. SIGGRAPH) 26*, 3 (2007).

[EHK*04] ENGEL K., HADWIGER M., KNISS J. M., LEFOHN A. E., SALAMA C. R., WEISKOPF D.: Real-time volume graphics. In *SIGGRAPH Course Notes* (2004).

[Fou92] FOURNIER A.: Normal distribution functions and multiple surfaces. In *Graphics Interface'92 Workshop on Local Illumination* (May 1992), pp. 45–52.

[FP02] FORSYTH D. A., PONCE J.: *Computer Vision: A Modern Approach*, us edition ed. Prentice Hall, August 2002.

[Hac05] HACHISUKA T.: *GPU Gems 2*. 2005, ch. High-Quality Global Illumination Rendering Using Rasterization.

[HPB07] HAŠAN M., PELLACINI F., BALA K.: Matrix row-column sampling for the many-light problem. *ACM Trans. Graph. (Proc. SIGGRAPH) 26*, 3 (2007).

[HREB11] HOLLÄNDER M., RITSCHEL T., EISEMANN E., BOUBEKEUR T.: Manylods: Parallel many-view level-of-detail selection for real-time global illumination. *Computer Graphics Forum (Proc. EGSR)* (2011), 1233–1240.

[HSRG07] HAN C., SUN B., RAMAMOORTHI R., GRINSPUN E.: Frequency domain normal map filtering. *Proc. of SIGGRAPH 26*, 3 (2007), 28:1–28:12.

[Jen96] JENSEN H. W.: Global illumination using photon maps. In *Proc. of EGSR* (1996), Springer-Verlag, pp. 21–30.

[Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, 2001.

[Kaj86] KAJIYA J. T.: The rendering equation. *Proc. of SIGGRAPH 20*, 4 (1986), 143–150.

[KD10] KAPLANYAN A., DACHSBACHER C.: Cascaded light propagation volumes for real-time indirect illumination. In *Proc. of I3D* (2010).

[Kel97] KELLER A.: Instant radiosity. In *Proc. of SIGGRAPH* (1997), pp. 49–56.

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. In *Proc. of I3D* (2010), ACM Press, pp. 55–63.

[LSK*07] LAINE S., SARANSAARI H., KONTKANEN J., LEHTINEN J., AILA T.: Incremental instant radiosity for real-time indirect illumination. In *Proc. of EGSR* (2007), pp. 277–286.

[LWDB10] LAURIJSSEN J., WANG R., DUTRE P., BROWN B.: Fast estimation and rendering of indirect highlights. *Computer Graphics Forum 29*, 4 (2010), 1305–1313.

[LZT*08] LEHTINEN J., ZWICKER M., TURQUIN E., KONTKANEN J., DURAND F., SILLION F., AILA T.: A meshless hierarchical representation for light transport. *ACM Trans. Graph. (Proc. SIGGRAPH) 27* (2008), 37:1–37:9.

[Max95] MAX N.: Optical models for direct volume rendering. *IEEE Trans. on Vis. and Computer Graphics 1*, 2 (1995), 99–108.

[Mil94] MILLER G.: Efficient algorithms for local and global accessibility shading. In *Proc. of SIGGRAPH* (1994), pp. 319–326.

[NSW09] NICHOLS G., SHOPF J., WYMAN C.: Hierarchical image-space radiosity for interactive global illumination. *Computer Graphics Forum (Proc. EGSR) 28*, 4 (2009), 1141–1149.

[REG*09] RITSCHEL T., ENGELHARDT T., GROSCH T., SEIDEL H.-P., KAUTZ J., DACHSBACHER C.: Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 28*, 5 (2009), 132:1–132:8.

[RGK*08] RITSCHEL T., GROSCH T., KIM M. H., SEIDEL H.-P., DACHSBACHER C., KAUTZ J.: Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph. (Proc. SIGGRAPH Asia) 27*, 5 (2008), 129:1–129:8.

[RHK*11] RITSCHEL T., HAN I., KIM J. D. K., EISEMANN E., SEIDEL H.-P.: Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes. *Computer Graphics Forum (Proc. EGSR)* (2011).

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-D shapes. *Proc. of SIGGRAPH 24*, 4 (1990), 197–206.

[THGM11] THIEDEMANN S., HENRICH N., GROSCH T., MÜLLER S.: Voxel-based global illumination. Proc. of I3D, ACM, pp. 103–110.

[TL04] TABELLION E., LAMORLETTE A.: An approximate global illumination system for computer generated films. *ACM Trans. Graph. (Proc. SIGGRAPH) 23* (2004), 469–476.

[Tok05] TOKSVIG M.: Mipmapping normal maps. *Journal of Graphics, GPU, and Game Tools 10*, 3 (2005), 65–71.

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: A scalable approach to illumination. *ACM Trans. Graph. (Proc. SIGGRAPH) 24*, 3 (2005), 1098–1107.

[WMG*] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. In *Eurographics*.

[WZPB09] WANG R., ZHOU K., PAN M., BAO H.: An efficient gpu-based approach for interactive global illumination. *ACM Trans. Graph. (Proc. SIGGRAPH) 28* (2009), 91:1–91:8.