2015-12-18

Many-Light Real-Time Global Illumination using Sparse Voxel Octree

Che Sun
Worcester Polytechnic Institute

Many-Light Real-Time Global Illumination using Sparse Voxel Octree

by

Che Sun

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

December 2015

APPROVED:

Professor Emmanuel Agu, Major Thesis Advisor

Professor Robert W. Lindeman, Thesis Reader

Abstract

Global illumination (GI) rendering simulates the propagation of light through a 3D volume and its interaction with surfaces, dramatically increasing the fidelity of computer generated images. While off-line GI algorithms such as ray tracing and radiosity can generate physically accurate images, their rendering speeds are too slow for real-time applications. The many-light method is one of many novel emerging real-time global illumination algorithms. However, it requires many shadow maps to be generated for Virtual Point Light (VPL) visibility tests, which reduces its efficiency. Prior solutions restrict either the number or accuracy of shadow map updates, which may lower the accuracy of indirect illumination or prevent the rendering of fully dynamic scenes.

In this thesis, we propose a hybrid real-time GI algorithm that utilizes an efficient Sparse Voxel Octree (SVO) ray marching algorithm for visibility tests instead of the shadow map generation step of the many-light algorithm. Our technique achieves high rendering fidelity at about 50 FPS, is highly scalable and can support thousands of VPLs generated on the fly. A survey of current real-time GI techniques as well as details of our implementation using OpenGL and Shader Model 5 are also presented.

Acknowledgements

Many thanks to my thesis advisor Emmanuel Agu and my thesis reader Robert W. Lindeman who supervised my research work on this master thesis and provided me with lots of useful hints and suggestions.

Thanks also to the fellow students at Zoo lab of the WPI Computer Science Department for their enthusiastic questions and discussions. Specially, I would like to thank Frank who spent a lot of time helping me improve our programming framework.

Finally, I thank my parents for their invaluable encouragement and patience, without which I could not finish this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Global illumination (GI) simulates the propagation of light and its interaction with surfaces and volumes within a 3D volume, dramatically increasing the fidelity of computer generated images. Since the formal introduction of the rendering equation [Kaj86], many offline GI algorithms such as ray tracing and radiosity have been proposed to solve it. While these techniques can generate physically accurate images, their rendering speeds are too slow for real-time applications.

More recently, a new wave of real time GI algorithms have emerged, targeting real-time applications such as video games. One class of these approaches is called many-light based method [DKH+14], derived from Kellers instant radiosity technique [Kel97]. The many-light method provides a simple lighting computation framework that transforms the problem of solving lighting transport equation to the calculation of the direct illumination from many virtual light sources. This algorithm is hardware-friendly and can easily be implemented on modern GPUs. One shortcoming of the many lights algorithm is that it often requires many shadow maps to be generated for Virtual Point Light (VPL) visibility tests, which reduces the its efficiency. While imperfect shadow maps [RGK+08] can alleviate this issue,

they are noticeably inaccurate for indirect shadows.

Another class of real-time global illumination methods use so-called voxels to discretize the original scene representation. Scene voxelization has several advantages: First, voxels are geometry-independent scene descriptions (algorithms using voxels as input do not need to deal with the original triangle meshes or other types of primitives) and some efficient scene voxelization methods have previously been proposed. Secondly, ray-geometry intersection and visibility tests on voxel data structures are very fast. Thirdly, high quality anti-aliasing techniques can be implemented using voxel data as well [CNS+11].

The goal of this thesis:

In this thesis, we first investigate existing state-of-the-art real-time GI methods. Then we introduce a novel hybrid real-time global illumination system which combines the advantages of the many-light method and voxelization-based method. We remove the expensive shadow maps generation step of many-light method and use an efficient Sparse Voxel Octree (SVO) ray-marching algorithm for VPL visibility tests. Finally we show that our technique can achieve high fidelity rendering quality at real time speeds.

1.1 Papers Generated by this Thesis

The work performed in this thesis has already been submitted to the International Symposium on Visual Computing (ISVC) 2015. And the paper has been accepted.

Chapter 2

Background and Related Work

In this chapter, techniques that are either related to our proposed approach, or utilized as building blocks for many GI algorithms are reviewed.

2.1 The VPL approach to solve the rendering equation

Global illumination takes into account not only the light coming directly from a light source (direct illumination), but also indirect light bounced off from other objects (indirect illumination). To solve the global illumination problem, Kajiya [Kaj86] introduced the following rendering equation to model it.

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) cos\theta_i d\omega_i \tag{2.1}$$

where x is the surface point being shaded. ω_o is the outgoing direction. ω_i is the light incident direction. θ_i is the angle between the normal of the shading point and ω_i. f_r is the Bidirectional Reflectance Distribution Function (BRDF). To calculate the outgoing radiance L_o, self-emit radiance L_e and surface point integration of the

incoming radiance L_i must be accumulated together. Here, L_i is acquired from $L_o\prime$, which is the outgoing radiance from another surface position. We can use operator notation to simplify equation 2.1 and yield the following equation

$$L = L_e + T_{f_r} L \tag{2.2}$$

where T_{f_r} is an notation of the integral in equation 2.1.

Thus, we can easily conclude that this equation is an infinite dimensional integration of the incoming radiance, which would be very inefficient if solved naïvely using recursion. To solve the problem in an iterative fashion, Keller [Kel97] introduced a two-pass algorithm that approximates the solution efficiently based on Neumann series, which provides a one dimensional infinite summation solution to equation 2.2. In realistic scenes the integral operator norm $||T_{f_r}|| < 1$, meaning that no surface reflects more light energy than it gains. So the Neumann series converges and can be approximated by a finite sum

$$L = \sum_{i=0}^{\infty} T_{f_r}^i L_e \approx \sum_{i=0}^{N} T_{f_r}^i L_e \tag{2.3}$$

Using this approximation, in the first pass the algorithm exploits the Quasi-Random Walk to generate a number of so-called Virtual Point Lights (VPLs), starting from the light sources then shooting them into the scene. Each time we hit a surface position, a VPL is distributed there and we can terminate a specific path based on probability. Afterwards, these VPLs are used to compute the sum L in equation 2.3. Therefore, the original high dimensional numerical integration can be evaluated by this elegant one dimensional integration, which is easy to implement on modern graphics hardwares.

2.2 Instant Radiosity methods

In this section, we give a brief introduction to several instant radiosity based real-time GI methods.

2.2.1 Reflective Shadow Maps

One of the early techniques derived from instant radiosity is called Reflective Shadow Maps (RSM) [DS05]. RSM extends the standard shadow map with additional data buffers to hold data needed by indirect lighting computation. This algorithm considers all pixels of an extended shadow map as first-bounce indirect illumination. RSM not only contains depth information of the scene viewed from the light source, but also the reflected radiance flux and geometric data such as world space positions and normals. Similar to the algorithm generating standard shadow map for the depth information, but RSM generation also employs modern GPU's Multiple Render Targets (MRTs) capability, storing the data mentioned above simultaneously.

The indirect illumination at a surface point x is then accumulated by sampling individual contributions of pixel lights from the RSM. Figure 2.1 illustrates this idea.

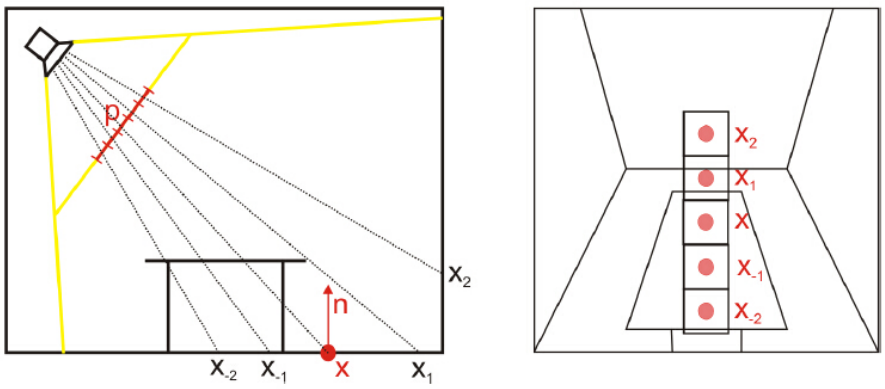

Figure 2.1: *RSM sampling. Left: A surface point x being shaded by several pixel lights. Right: A view generated from a scene light. (Image courtesy of [DS05]).*

Based on the fact that the important indirect lights that are close to the shading position will also be close in the shadow map, the algorithm first projects the shading position into the shadow map position (s, t). Then it selects the samples in polar coordinates relative to (s, t) with a weighted uniform random distribution sequence, as is shown in Figure 2.2.

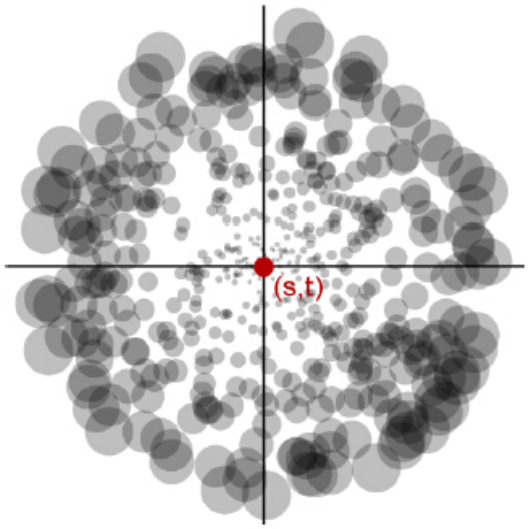

Figure 2.2: *RSM sampling pattern example. (Image courtesy of [DS05]).*

Note that this algorithm ignores visibility tests for pixel lights since spatial occlusion information cannot be captured by an RSM. This is one drawback using pixel lights directly from RSM for indirect illumination.

While the final rendering result of RSM is not as good as modern complex GI methods, the algorithm introduces an efficient way of sampling first-bounce VPLs for real-time GI applications. As we will see later in this thesis, many advanced real-time GI methods have employed RSM as a sub-algorithm for their multi-pass algorithms.

2.2.2 Incremental Instant Radiosity

Incremental instant radiosity is a one-bounce real-time instant radiosity based algorithm [LSK+07]. It samples the 3D scene using ray tracing for one-bounce VPL generation and exploits paraboloid shadow maps (PSM) [BAS02][OBM06] for VPL visibility tests. The basic idea of the algorithm is to maintain reusable VPLs across rendering frames and regenerate new VPLs in an incremental fashion such that not too much of the rendering time budget will be spent on VPL shadow map regeneration. According to the authors [LSK+07], the algorithm can be outlined as follows:

1. **Determine if each VPL is valid for the current frame:** A VPL is considered as valid if it is not occluded by the scene objects and is inside the region of the light source (only for spot light).

2. **Remove invalid VPLs and some of the valid VPLs to make room for VPL re-distribution:** Deletion of valid VPLs is based on a Delauney triangulation technique, which helps find a more uniform distribution by removing VPLs in areas with high densities.

3. **Create new VPLs to achieve VPL re-distribution:** Here the Delauney triangulation technique is used again to place new VPLs in regions that have low density. Then shadow maps are generated for the new VPLs.

4. **Compute intensities for VPLs:** While some of the VPLs are considered as valid across multiple frames, their intensities have to be adjusted due to the fact that the light sources may move from frame to frame.

5. **Create a G-buffer for deferred rendering.** Geometry buffer (G-buffer) [ST90] usually stores position, normal and materials data needed by a deferred

rendering system for lighting computation.

6. **Split the G-buffer:** Split the G-buffer into a number of tiles such that each tile only references a subset of VPLs for interleaved sampling.

7. **Indirect illumination using VPL interleaved sampling.**

8. **Merge the indirect illumination image.**

9. **Apply a geometric-aware filter on the merged indirect illumination image.**

The algorithm is implemented partially on CPU side (VPL management, steps 1-4 except paraboloid shadow maps generation) and partially on GPU side (rendering, steps 5-9). This separation makes it possible to execute the two parts in parallel both on CPU and GPU. Here we will not explain the details of their VPL management algorithm. Readers interested in the VPL management part could get insights from their original paper [LSK+07].

This technique requires that the scene geometry used for VPL sampling remains static. Therefore, dynamic objects cannot generate indirect illumination effects, but they can receive indirect illumination from the static part of the scene. Restricted by this limitation, while the technique yields high real-time frame rates, it is not a fully dynamic algorithm. Figure 2.3 shows three examples of their rendering result.

2.2.3 Imperfect Shadow Maps

Indirect illumination usually consists of smooth gradations, which tend to mask errors due to incorrect visibility. Imperfect shadow maps (ISM) [RGK+08] exploits this nature of indirect illumination by approximating VPL visibility tests with so

Figure 2.3: *Incremental Instant Radiosity (Image courtesy of [LSK+07]).*

called imperfect shadow maps, which are low resolution shadow maps generated from a point based representation of the scene.

The algorithm requires a one-time scene preprocessing step to generate point-based representation of the scene. For each point, the barycentric coordinates relative to its triangle and the triangle index are stored such that the point cloud can be reused to support fully dynamic scene. Rendering this point cloud is much more efficient than rendering the original triangle mesh representation. Therefore, hundreds of inaccurate shadow maps can be generated on the fly for those VPLs.

Note that an inaccurate shadow map cannot be used for VPL visibility tests directly since there are holes and discontinuous areas in it. This inaccuracy highly depends on the triangle representation of the original scene. Therefore, triangle mesh tessellation has to be applied to objects that have a low number of triangles. A sensible depth map is reconstructed by using a pull-push image processing approach [GD98][MKC07].

For multiple bounces, the technique generalizes ISMs to imperfect reflective shadow maps (IRSMs) similar to RSMs. Given the first-bounce VPLs and their IRSMs, second-bounce VPLs are sampled from those IRSMs. This iteration of re-rasterizing the point based scene representation for n-bounce VPLs generation makes the algorithm less efficient for n-bounce indirect illumination as high mem-

ory bandwidth is necessary to perform such rendering tasks.

While the rendering speed of this technique is very fast for one-bounce indirect illumination, the quality of the final image depends on carefully chosen point samples by hand. This limitation makes ISM not a fully automatic algorithm. Figure 2.4 shows a dynamic Cornell box scene rendered using ISM.

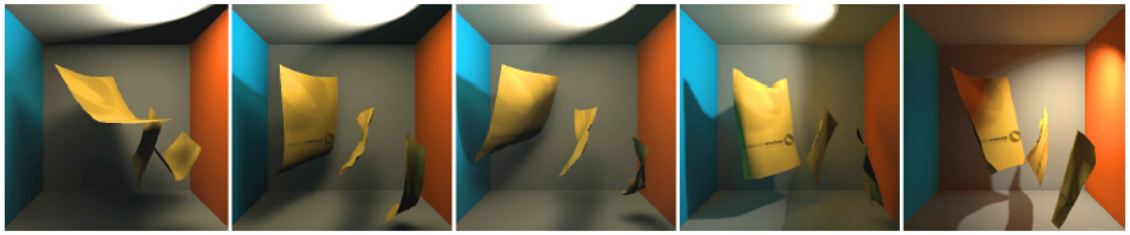

Figure 2.4: *Dynamic Cornell box scene rendered using ISM. (Image courtesy of [RGK+08]).*

2.2.4 Instant Radiosity using Temporal Coherence

Previous instant radiosity methods need a high number of VPLs to achieve visually compelling results especially for scenes with moving objects. This is due to the fact that a VPL sampled from a moving object could abruptly change its position and orientation, which may lead to unwanted sudden illumination changes. This kind of unpleasant flickering is called temporal aliasing. A natural way of solving it is to increase the number of VPLs such that ideally a shading point can acquire enough VPLs to compensate for a subset of VPL illumination changes. While this approach is acceptable for off-line GI applications, it is often impractical for real-time GI applications due to the time budgets spent on VPL shadow map generation and shading. Knecht [Kne09] proposed a method that alleviates this kind of temporal aliasing caused by insufficient VPLs using a technique based on temporal coherence. Figure 2.5 shows a scene rendered using their method.

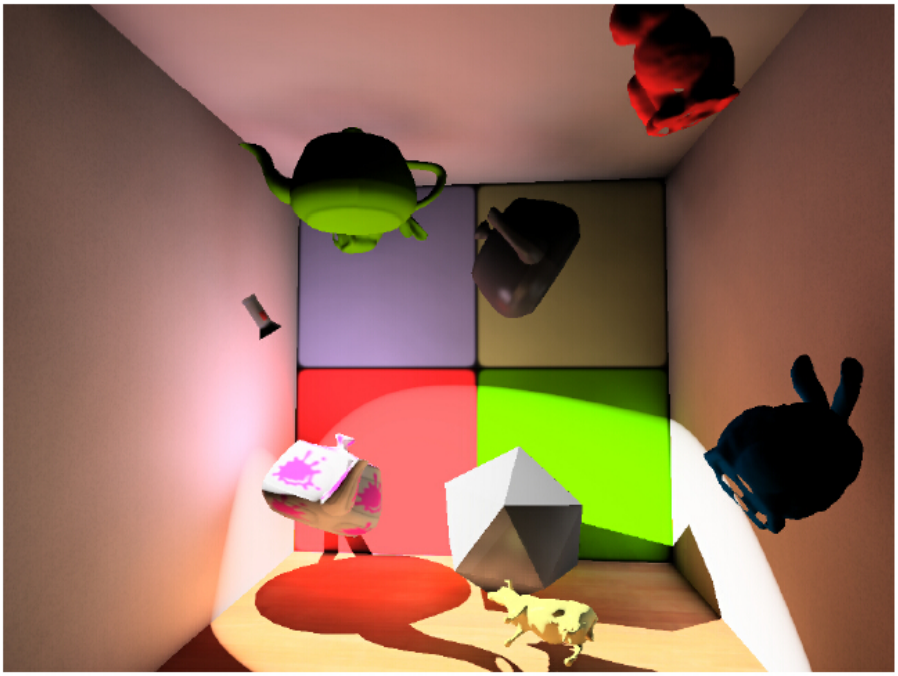

Figure 2.5: *Instant radiosity using temporal coherence. (Image courtesy of [Kne09]).*

Their basic idea is to exploit temporal coherence between consecutive rendering frames. In this way, they can reuse VPL shading results from previous frames and accumulate indirect illumination across multiple frames. To support deferred rendering, a so-called movement buffer is created to store the position difference between the previous and current frame. Later, the difference data is used for confidence calculation.

Confidence Computation

Fetching indirect illumination of previous frames using the movement buffer is straightforward. However, this value cannot be used directly since the objects are moving and the lights and camera setting may change. Therefore, calculating how confident an indirect illumination value is from the previous frame is a necessary step. The authors take three different parameters into account:

1. The relative position change of the shading pixel.

2. The relative normal change of the shading pixel.

3. The difference of the previously calculated illumination and the current illumination.

Figure 2.6 shows an example of confidence visualization. The final indirect illumination for the current frame is then calculated as a linear combination of the current and previous frames as follows:

$$I_final = I_prev \times confidence + (1 - confidence) \times I_current \tag{2.4}$$

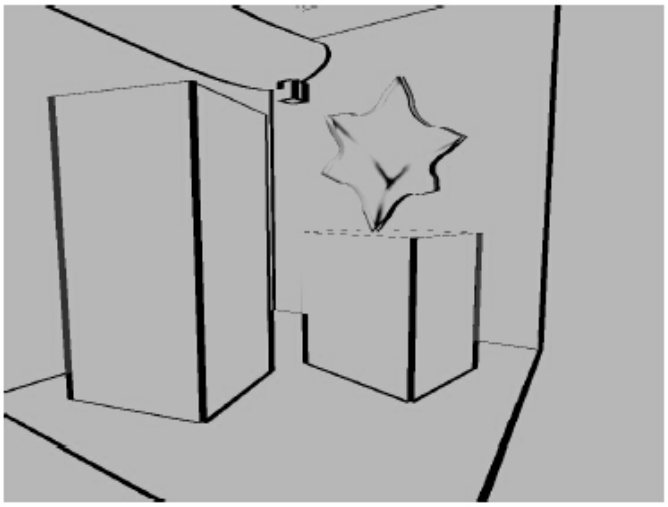

Figure 2.6: *An example scene visualizing confidence value. The brighter the pixels, the more confident a previous shading pixel is. (Image courtesy of [Kne09]).*

2.2.5 Metropolis Instant Radiosity

The distribution of VPLs within the 3D scene is also an important issue that affects the rendering quality of instant radiosity methods. Poor one-bounce VPL sampling leads to artifacts and unshaded scene surfaces [SIP07]. The standard VPL sampling strategy implemented by many instant radiosity methods suffers from high variance

due to the fact that it is not a view-dependent technique, which often results in poor VPL distribution for complex scenes. Metropolis Instant Radiosity (MIR) is a robust view-dependent algorithm that can handle such complex scenes. Figure 2.7 shows an example of the algorithm.

Figure 2.7: *A complex scene layout rendered with Metropolis Instant Radiosity. (Image courtesy of [SIP07]).*

The authors propose to sample the VPLs using an innovative Multiple-Try Metropolis-Hastings (MTMH) algorithm. Applying MTMH to arbitrary scene layout could yield unconditionally robust low variance rendering results. The basic idea is to adjust the positions of a set of VPLs in an iterative fashion to make them generate an optimal final indirect illumination. Key steps of it is described as follows:

1. Compute the power P_c received by the camera using bidirectional path tracing.

2. Compute a set of VPLs with a density proportional to the power they bring to the camera with a Metropolis-Hastings sampler.

3. Suppose for each VPL, its outgoing radiance equals to 1. Compute its intensity and the total power P' transmitted to the camera. Then accumulate its contribution and finally rescale the result by a factor of $\frac{P_c}{nP'}$.

MIR is view-dependent and constructs robust light transport sub-paths to the camera. However, it does not resolve the flickering issues when applied to dynamic scenes since flickering is mainly dominated by the sampling frequency of the incoming radiance field, not the sampling positions. Meanwhile, like other instant radiosity methods, MIR only handles diffuse or not-too-glossy surfaces.

2.2.6 Bidirectional Path Tracing via Global Ray-bundles

Till date, the MIR method mentioned in section 2.1.5 has not been implemented to achieve real-time frame rates due to the complex camera received power computation pass and Metropolis-Hastings VPL generation pass. Meanwhile, an alternative state-of-the-art bidirectional path tracing method for real-time GI applications has been proposed by Tokuyoshi [TO12] to produce low variance indirect illumination for real-time instant radiosity (shown in figure 2.8). The authors implemented a bidirectional path tracing technique by constructing robust light transport paths from both the light source and the camera with global ray-bundles and VPLs.

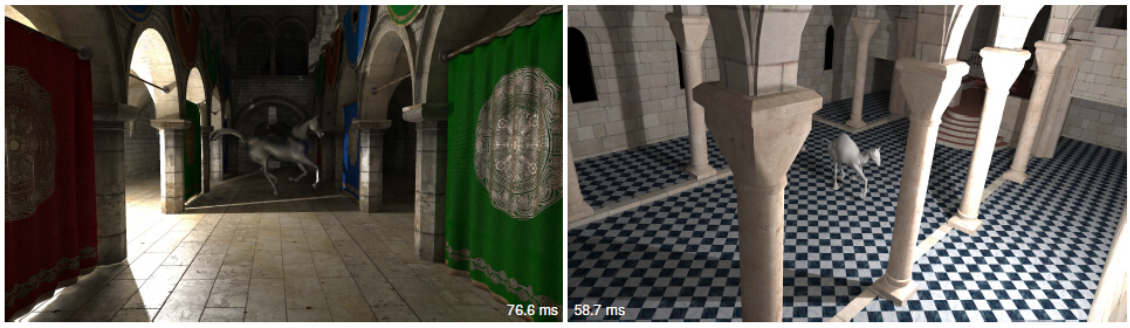

Figure 2.8: *Dynamic scenes rendered using bidirectional path tracing via global ray-bundles (Image courtesy of [TO12]).*

Global Ray-Bundles on the GPU

The idea of bidirectional path tracing is to construct robust light transport paths from both camera direction and light direction such that some issues that were

hard to deal with for complex scene layout can be addressed with bidirectional constructed paths. The authors approximate path tracing of the camera direction by using global ray-bundles [SiS96], which are groups of parallel rays sampling the scene geometric information. Here, instead of doing visibility tests for individual pixels, global ray-bundles focus on a single global direction, therefore, the visibility for all pixels is computed in parallel by taking advantage of the GPUs efficient rasterization mechanism. Figure 2.9 shows an example of this idea.

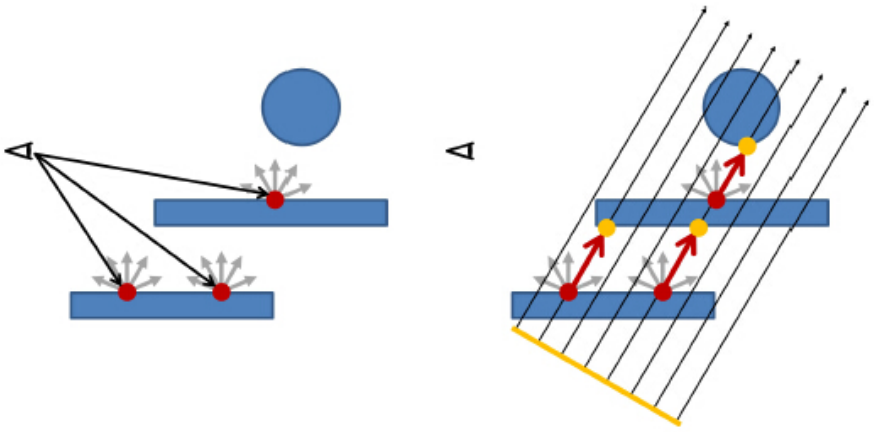

Figure 2.9: *One-bounce path tracing via global ray-bundles. Left: visible points from a camera. Right: global ray-bundles. (Image courtesy of [TO12]).*

For each pixel being shaded, global ray-bundles need to create an intersection list recording every intersection position of a specific ray with the scene. This algorithm can be implemented using the atomic operation and unordered access features provided by shader model 5. A so-called per-pixel linked list [YHGT10] is created for each shading pixel to store the ray-geometry intersection information. Surface positions, normals and surface material are stored in list nodes for later use.

When computing indirect illumination, a lookup operation is performed using the linked lists to get the intersection information for a specific shading surface position. This operation efficiently constructs a camera sub-path for it. The light sub-path is

constructed using standard VPL sampling technique (RSM for first-bounce VPLs). Connecting both the light sub-path and the camera sub-path establishes a complete light transport shading path, as is shown in figure 2.10.

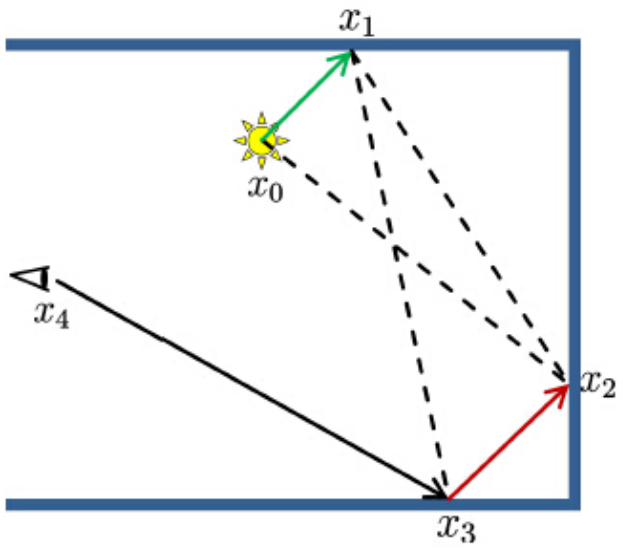

Figure 2.10: *Indirect illumination paths sampled by their bidirectional path tracing. Green arrow: light sub-path. Red arrow: camera sub-path. (Image courtesy of [TO12]).*

While global ray-bundles can construct more robust light transport paths for VPLs than the standard light direction sampling strategy, it introduces additional burdens, since every time a VPL is used, scene polygons must be rasterized again in order to create a corresponding global ray-bundle. Consequently, although their technique produces more photorealistic images for complex scenes, it does not run at high frame rates.

2.3 Voxel-based methods

Voxel-based methods discretize the scene description and store discretized data in 3D grid cells (voxels). An outstanding feature of scene voxelization is that it generates an approximation of scene geometric information extremely fast (usually less than several milliseconds on current commodity GPUs for a scene complexity of a hundred thousand-level triangles).

2.3.1 Light Propagation Volume

Light Propagation Volume (LPV) [KD10] is one of the most efficient voxel-based real-time GI methods. It uses low resolution lattices and spherical harmonics to represent the spatial and angular distribution of radiance field in the scene. This algorithm does not require any precomputation and handles large scenes with cascaded lattices grids.

Figure 2.11: *The Crytek Sponza scene rendered using Cascaded Light Propagation Volumes. (Image courtesy of [KD10]).*

Discrete Ordinates Method

LPV is based on the Discrete Ordinates Method (DOM) [Cha13] technique, which discretizes continuous quantities and approximates the radiative transfer equation (RTE) in space and orientation. The radiance distribution is stored in a voxel grid by applying light energy exchange between neighboring volume elements. This idea is similar to the Finite Element Method (FEM) which is used

to solve partial differential equations (PDE) numerically. However, setting up a boundary condition is not necessary for the diffusion process of light transport duo to the time budgets of real-time rendering requirements.

LPV represents the lighting in a scene sampled on a lattice and models the light transport using local operations that can be performed using modern GPUs in a parallel manner. According to the authors, computing the indirect illumination can be described as the following steps [KD10]:

1. Initialize the light propagation volume (LPV) with surfaces causing indirect illumination. (Light injection step)

2. Sample the scene geometry and create a so-called geometry volume using depth peeling from the camera and reflective shadow maps from scene lights. The geometry volume represents visibility information that is used to block light transfer.

3. Perform light propagation starting from the initial LPV and accumulate the intermediate results. This step is carried out in an iterative fashion that leads to the final light distribution.

4. Render the scene geometry using the propagated LPV.

To initialize the LPV, a significant amount of VPLs are created using RSM since they do not compute VPLs contribution individually, but only use them for LPV injection. Then the VPLs are transformed into the SH representation and stored in the LPV voxel grid. Similar to the VPL injection, scene geometry occlusion information is injected into the geometry voxel grid and represented with SH as well. The geometry volume has the same resolution as the LPV but is shifted by half a cell to achieve better interpolation of the visibility during light propagation.

Light is propagated along XYZ axial directions to the six neighboring voxels of the current voxel. During the propagation, geometric blocking of light is integrated using the geometry volume. Figure 2.12 illustrates the aforementioned step.

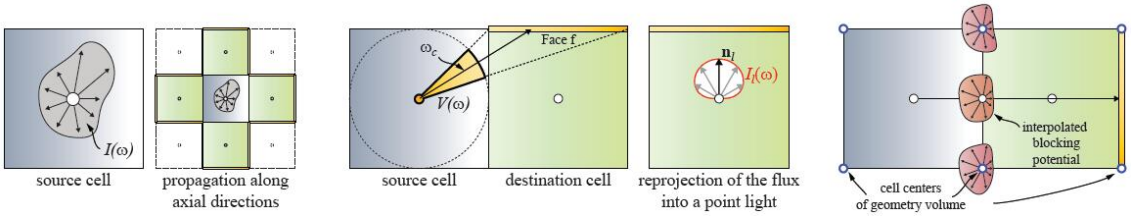

Figure 2.12: *Light propagation. Left: Each voxel of the LPV stores the directional intensity used to compute the light that is propagated to six neighboring voxels along axial directions. Center: The flux onto the faces of the destination voxel is computed to preserve directional information. Right: Geometry volume is used to account for fuzzy occlusion for light propagation. (Image courtesy of [KD10]).*

LPV has the advantages of low cost and flicker-free renderings. The latter is achieved with the high sampling frequency of VPL generation and injection, which is hard to achieve at real-time frame rates for instant radiosty methods. One limitation of LPV is that the spatial discretization might be visible as light bleeding. Another limitation is that the SH representation of the intensity and the propagation leads to strongly diffuse reflection appearance, which makes handling glossy indirect illumination very hard.

2.3.2 Real-Time Near-Field Single Bounce Indirect Light

Thiedemann et al [THGM11] introduces a regular grid based voxelization technique that supports fast near-field VPL visibility tests. Their technique make heavy use of 2D/3D textures to store voxel fragments and regular voxel grid. For binary voxelization (each voxel only needs to indicate if its spatial region is occupied by scene geometry), a 2D texture is enough to represent a regular voxel grid. However, to support multi-valued voxelization, a bunch of 3D textures have to be employed

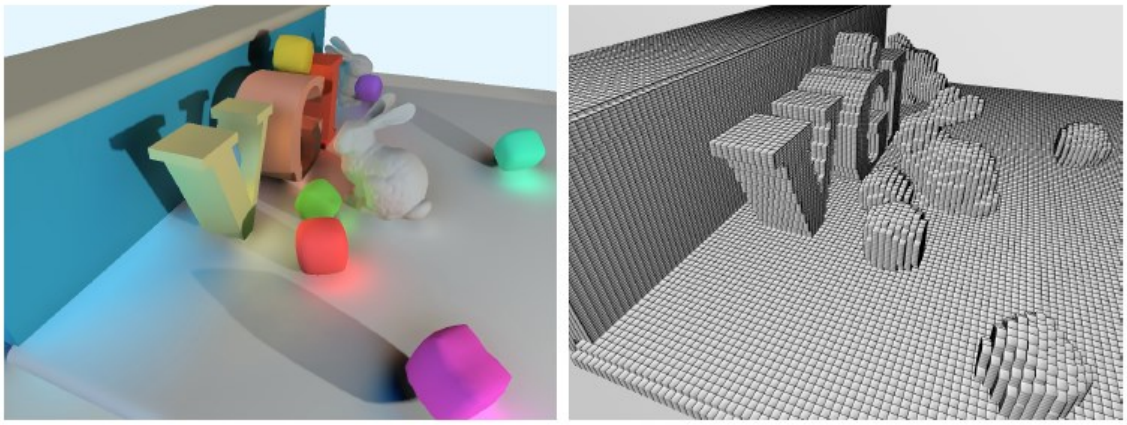

Figure 2.13: *A sample scene rendered using real-time near-field technique. (Image courtesy of [THGM11]).*

as Multiple Render Targets (MRTs) to store more complicated scene geometric information as needed by the application. This may lead to high consumption of GPU memory due to the nature of regular voxel grid representation.

In addition, this technique also requires a pre-computation step to establish a geometry-to-texture-atlas mapping such that when the geometry is rendered, its surface positions are mapped to texture atlas space and generate voxel fragments there. Later after texture atlas generation is finished, scene voxelization is performed by dispatching vertex shader threads (point rendering) for each non-empty texel of the texture atlases. Figure 2.14 illustrates this two pass voxelization technique.

Hierarchical ray/voxel intersection test

The main advantage of using the aforementioned voxel grid representation is that ray/voxel intersection test can be implemented very efficiently. To achieve real-time frame rates, the authors limit their ray-tracing algorithm to near field single bounce indirect illumination. A binary voxelization is used to store scene geometric occlusion into a 2D texture. Each bit of a texel encodes whether there is scene geometry occupying the space. Based on the idea of Forest et al [FBP09], a mip-map hierarchy is built for the 2D texture. Each mip-map level is created using a

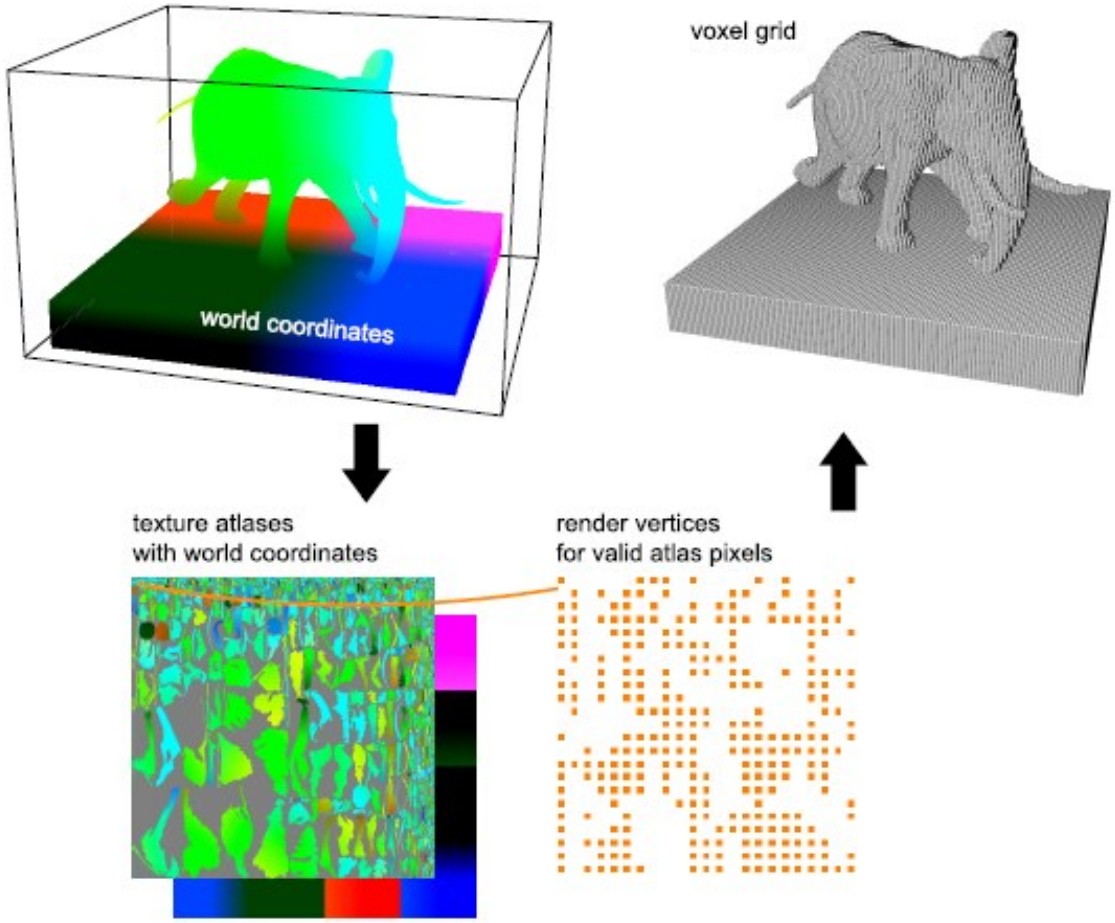

Figure 2.14: *Two pass scene voxelization using texture atlas. (Image courtesy of [THGM11].*

bitwise logical OR-operation from a previous mip-map level. Note that the depth resolution is preserved to stay the same at each level (shown in Figure 2.15).

After the mip-map generation, a hierarchical voxel grid traversal algorithm can be performed to implement visibility query. It starts at the coarsest mip-map level and transforms the ray into Normalized Device Coordinates (NDC). Appropriate texel which represents a column of voxels is chosen based on the starting point of the ray. The bounding box of the texel in NDC space is then generated and ray-box intersection test is performed. If no intersection occurs, the ray marches forward to its intersection point with the bounding box and a coarser mip-map level is used for

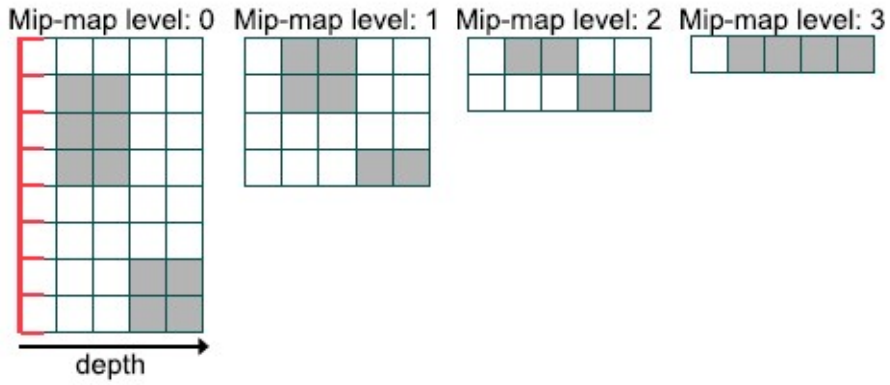

Figure 2.15: *Mip-map hierarchy generated in two dimensions. (Image courtesy of [THGM11]).*

the next iteration. If an intersection occurs, a finer mip-map level has to be used for the next iteration (The finest mip-map level is level 0). The algorithm terminates when a hit is found or the length of the ray becomes zero. Figure 2.16 illustrates the algorithm.

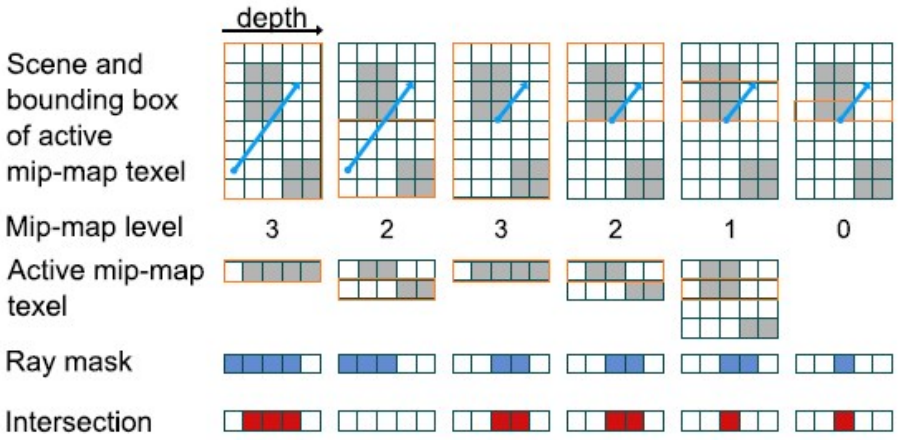

Figure 2.16: *Hierarchical traversal in two dimensions. (Image courtesy of [THGM11]).*

As mentioned in the beginning of this section, one drawback of the technique is the high consumption of GPU memory. Furthermore, using 2D texture to store binary voxel grid restricts the depth dimension to the maximum size of texel format,

which is 128 bit for 4-channel floating-point texture. These limitations make the binary regular grid representation impractical to support large-scale complex scenes.

2.3.3 Interactive Indirect Illumination Using Voxel Cone Tracing

Interactive Indirect Illumination Using Voxel Cone Tracing was introduced by Crassin et al [CNS+11]. The authors developed a fast dynamic GPU Sparse Voxel Octree (SVO) generation method and use it for high quality GI rendering: they utilize an injection and pre-filtering process to filter data related to indirect illumination in a bottom-up fashion. The resulting hierarchical octree structure is then used by their voxel cone tracing technique to produce high quality indirect illumination which supports both diffuse and high glossy indirect illumination. Figure 2.17 shows their rendering results.

Figure 2.17: *Real-time indirect illumination rendered using voxel cone tracing. (Image courtesy of [CNS+11]).*

The algorithm can be summarized as the following steps:

1. **Scene voxelization step:** Static objects are voxelized only once during the initialization phase of the application. Dynamic objects are re-voxelized every frame and added to the same SVO in which static objects reside.

2. **Direct lighting injection step:** RSMs are used for scene lights to inject incoming radiance (energy and direction) into the leaves of the SVO.

3. **Filtering step:** The incoming radiance values are filtered into the higher levels of the SVO to form a complete 3D mip-map hierarchy.

4. **Rendering step:** Deferred shading is performed for direct and indirect illumination. For indirect illumination, several diffuse cones (usually 5) and one specular cone is traced from the shading point to integrate incoming radiance above the hemisphere.

Scene Voxelization

The SVO structure is created using the GPU rasterization pipeline. Meshes are rasterized three times along the three main axes of the scene. A fragment shader thread is generated for each potential leaf node. The thread traverses the octree from top-to-bottom and subdivides it as needed. The surface attributes are stored to the leaf node.

To efficiently make use of GPU cache coherence, child nodes of a specific parent node are grouped together into a 2^3 tile. A global node buffer is created to be used as a node memory pool. Node allocations are made using a global atomic counter shared by multiple threads. A so-called brick is used to store per-node information. Bricks are allocated from a separate texture memory pool using a similar allocation scheme as node memory pool. The idea of using it is to take advantage of hardware-supported trilinear filtering of texture data.

Direct lighting injection

Inspired by Dachsbacher and Stamminger [DS05], the authors employ RSM to capture direct lighting and store the photons in RSM as a direction distribution and an energy proportional to the subtended solid angle of the pixel as seen from the light.

A fragment shader thread is generated for each pixel in the RSMs. Afterwards, lighting information is injected into leaves of the SVO. To make use of hardware-supported filtering, each brick stores 3^3 voxels. The redundancy is necessary since neighboring voxels need to be accessed when filtering a specific higher level node. To propagate redundant voxel data as fast as possible, the authors introduce a six-pass technique to transfer voxel values between neighboring bricks, as is illustrated in figure 2.18.

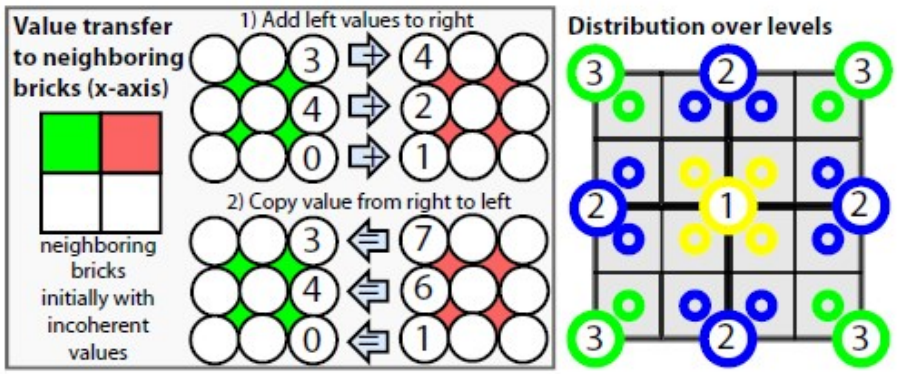

Figure 2.18: *Left: Value transfer to neighboring bricks (x-axis). Right: Three-pass filtering of a lower to a higher level. (Image courtesy of [CNS+11]).*

Filtering

The goal of SVO filtering is to propagate and convolve lighting related values from a lower to a higher tree level and so forth. A nave solution would be to launch one thread for each voxel of the higher level and fetch data from the lower level. But this method introduces redundant computations many times due to the need of neighboring voxel accessing. Thus the authors proposed a three-pass filtering technique to filter the voxel values in a divide-and-conquer fashion: the first pass filters only the center voxel using the relevant 27 lower level voxel values. The second pass then generates values for 4 voxels residing on the edges of the brick, only 18 voxel values are needed for each of them. Finally, the third pass gather values for the

4 voxels left at the corners of the brick. This time only 8 voxel values are necessary for each of them. See the right image of Figure 2.18 for understanding this idea.

Rendering

For indirect illumination, several diffuse cones (usually 5) and one specular cone are traced from the shading point to integrate incoming radiance above the hemisphere, which implements a Phong BRDF shading model. For each ray cone, ray marching is performed to fetch values from the SVO node based on the current cone radius. Meanwhile, quadrilinear interpolation is used to remove aliasing. Incoming radiance values are accumulated along the cone for each ray marching step. Figure 2.19 shows the technique mentioned above.

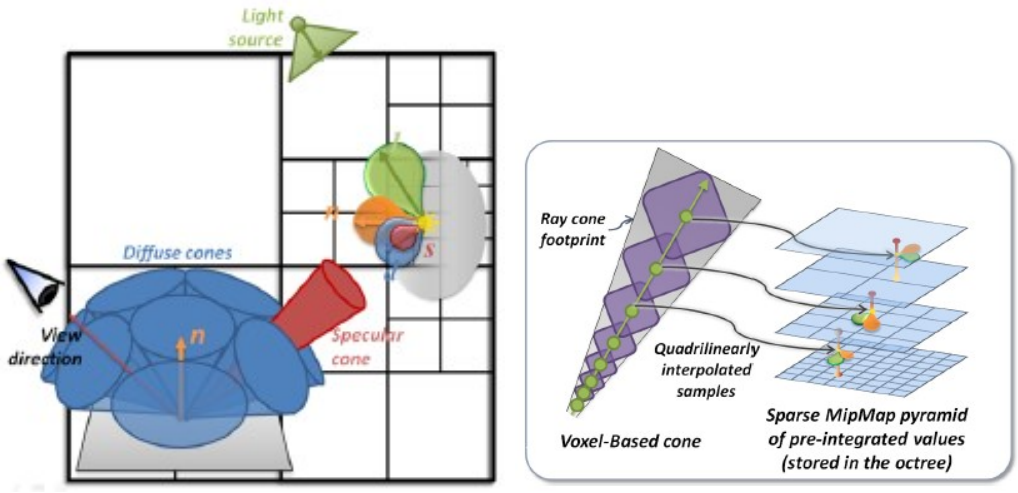

Figure 2.19: *Voxel-based cone tracing using pre-filtered geometry and lighting information from a sparse mip-map pyramid. (Image courtesy of [CNS+11]).*

Chapter 3

Our Algorithm

We now present our proposed real-time GI algorithm. Our lighting system is a deferred renderer, which supports illumination by multiple scene lights. First, we voxelize the scene and create an SVO to represent geometric occlusion. Similar to other deferred rendering systems, we then create a geometry buffer (G-buffer) [ST90] that stores position, normal and materials for direct and indirect illumination. The G-buffer is then split for VPL interleaved sampling [KH01]. For each scene light we render the scene from the lights view and create a reflective shadow map [DS05] into which we store position, normal and flux information that is needed by VPL importance sampling [CJAMJ05].

Direct illumination is calculated using standard shadow maps and illumination by all the scenes lights is accumulated into a direct illumination buffer. Indirect illumination is calculated by accumulating contributions of VPLs into an indirect illumination buffer. To accelerate indirect illumination, we adopt a technique similar to Segovia [SIMP06]: each pixel being shaded is only assigned a subset of the VPLs using a pre-generated interleaved sampling pattern. A visibility test is performed by shooting shadow rays from the pixel toward the VPL subset. Here, instead of

performing shadow map lookup and comparison, we perform SVO ray-marching to query if the shadow ray is occluded. To make use of cache coherence of modern GPUs, we group pixels that utilize the same VPL subset together by splitting the G-buffer into 4×4 tiles. After indirect illumination is complete, the indirect illumination buffer needs to be merged and filtered so that it can be combined with the direct illumination buffer to create the final image. Figure 3.1 shows the lighting pipeline steps described here.

3.1 Data Representation

Ray-tracing triangle meshes for real-time applications is still a formidable task even using the most recent generation of high-end desktop GPUs. Thus, several voxelization based methods have been developed to convert the original triangle mesh scene to a voxel grid in order to accelerate ray-geometry intersection calculations. In our case, since we eliminated the expensive VPL shadow maps generation step, we needed an alternative spatial data structure for fast VPL visibility tests. Here, we exploit SVO as our geometric representation of the scene.

Similar to Crassin [CNS+11], we first build the SVO in a non-recursive fashion and store it in GPU memory. Later the SVO structure will be read by multiple GPU threads concurrently during VPL visibility tests. To efficiently take advantage of GPU thread-group caches, the tree node is designed to be as tight as possible. Groups of eight tree nodes are grouped and placed together based on the idea of Crassin [CNS+11]. Fig.3.2 illustrates the memory layout of the tree node.

16 bytes (4 unsigned integer) are used to represent a tree node such that it does not straddle the GPU cache line. The first data item is node info, in which three data items are encoded: the most significant bit indicates whether or not this is

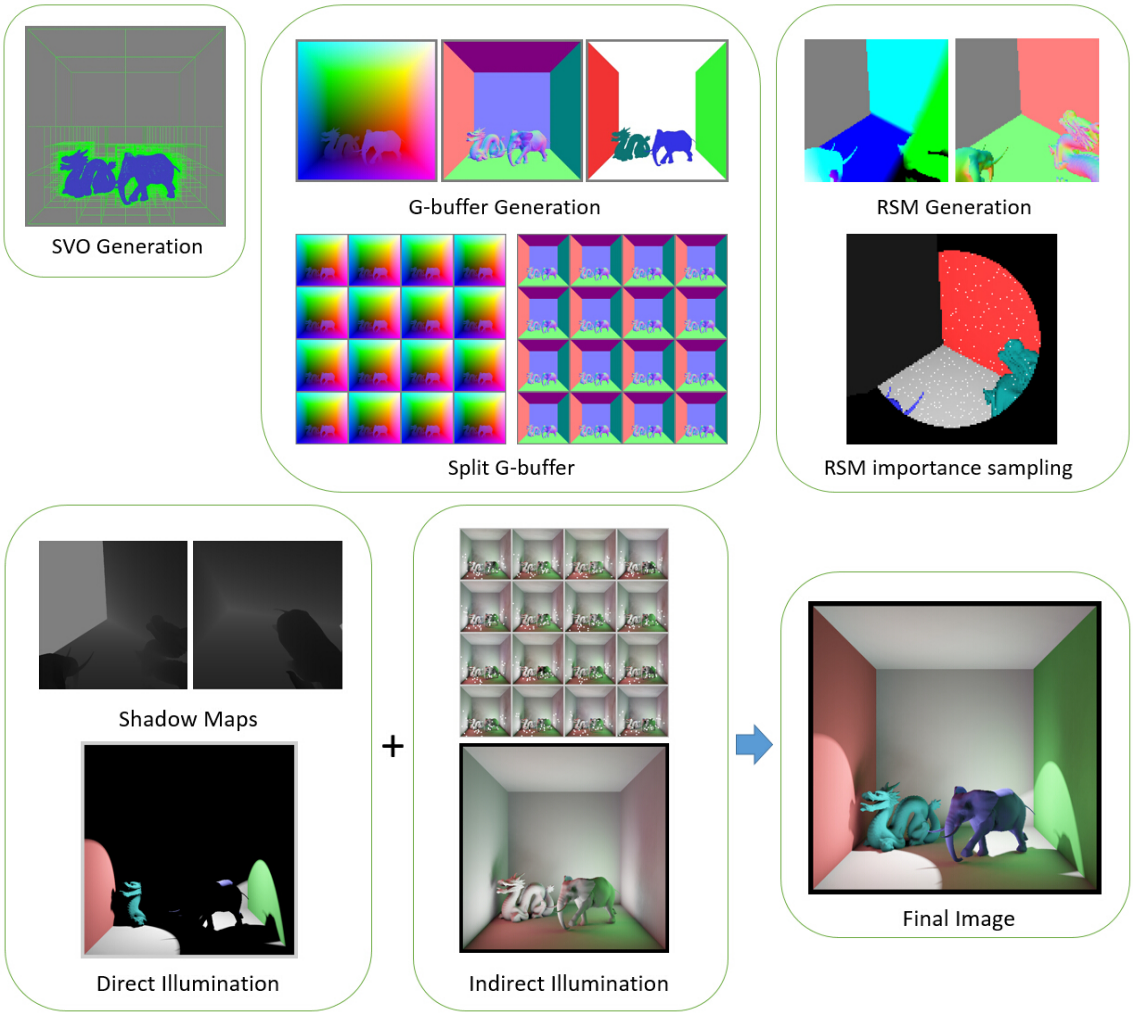

Figure 3.1: *Lighting pipeline steps. Top left: Scene voxelization and SVO generation. Top middle: G-buffer generation. World position, normal and diffuse materials are rendered into floating-point textures. Position and normal buffers are split for calculating indirect illumination. Top right: RSM generated from scene lights view. First-bounce VPLs are created by sampling the RSM. Bottom left: Direct illumination using scene lights and corresponding shadow maps. Bottom middle: Indirect illumination calculated with split G-buffer and SVO ray-marching. Merging and filtering are performed to produce the final indirect illumination result. Bottom right: Adding direct and indirect illumination together. A simple HDR tone mapping is applied to produce the final image*

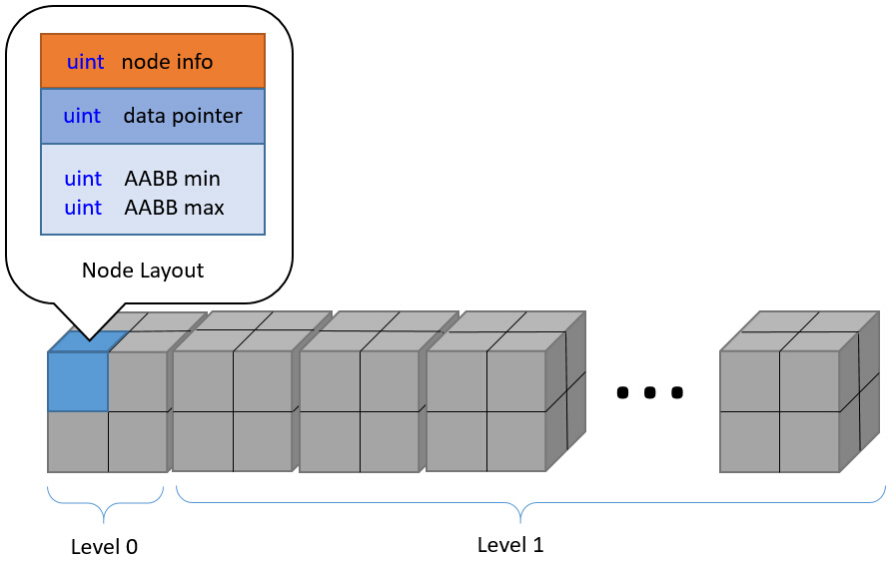

Figure 3.2: *SVO node memory layout*

a leaf node. The next most significant bit indicates if the node has been flagged as a node containing geometric data during the process of tree construction. The remaining 30 bits are used to store the tile ids of child nodes, which is a pointer to the tile block of eight child nodes of the current node. The second data item is a data pointer that could be used to store user specific data. The idea here is to separate the sparse voxel octree itself with user specific per node data such that shader programs constructing and traversing the tree could benefit from less cache miss rates. Currently we have not used the data pointer since our main purpose using the tree is for VPL visibility tests, which requires no further information besides scene geometric data already presented in the tree. Finally, we encode an axis-aligned bounding box (AABB) for the node as two unsigned integer values, which are retrieved efficiently using bitwise operations during ray marching. Each component (x, y and z) is represented using 10 bits so that the maximum resolution of the SVO grid is 1024^3.

3.2 VPL visibility tests using Sparse Voxel Octree

We use SVO to perform shadow ray marching between each surface point being shaded and the VPLs. To exploit data coherence, surface positions accessing the same VPL subset are grouped together. We also assume that the surface material only has a diffuse property for indirect illumination. This assumption leads to incoherent tree node access patterns due to the nature of diffuse reflection. Thus, packet traversal [AL09] cannot be applied easily as grouping individual shadow rays that have a uniform distribution over a hemisphere does not benefit much from fetching the same nodes during ray marching. However, dividing shadow rays into packets based on VPLs that reside in the same solid angle is a viable option to improve node access coherence, which we will explore in future work. Currently we only implement a per-ray traversal method. In section 4 we will show that the algorithm has a good performance for diffuse indirect illumination.

The kd-restart traversal algorithm is an efficient method for performing ray marching with thousands of rays concurrently on modern GPUs [FS05][HSHH07]. Consequently, we implement an SVO-restart traversal algorithm for VPL visibility tests. We also selected a kd-tree since it splits spatial regions in two as compared to the octree which splits the space into eight parts, making packet traversal possibly more efficient on kd-trees. Our ray marching algorithm in pseudocode is shown in Fig. 3.3.

The ray marching algorithm takes world space ray end positions and SVO root as input. It first transforms the ray end positions to SVO space. In line 2 the ray segment is shrunk slightly to avoid intersection issues because both the shading position and VPL position are on geometry surfaces that have been flagged as SVO leaf nodes. From line 3 to 7 we initialize some variables used by the subsequent

```
1 Transform world space ray start and end positions to SVO space
2 Offset SVO space ray start and end positions by epsilon
3 rayDirSVO = rayendpositionraystartposition
4 sceneMaxT = length(rayDirSVO)
5 normalize(rayDirSVO)
6 hit = 0
7 minT = maxT = 0.0
8 while( maxT < sceneMaxT )
9     curNode = root
10    mint = maxT
11    maxT = sceneMaxT
12    rayentryposition = raystartposition + raydirection * mint
13    while( !isLeafNode(curNode) )
14        Figure out which child node of curNode the ray entry position is in
15        Update curNode to be the child node
16    if( Flaged(curNode) )
17        hit = 1 and break
18    Fetch AABB for curNode
19    Compute ray-AABB intersection positions and possibly return hit = 0
20    Update maxT = t1 + SVO_RAY_T_EPSILON
21 return hit
```

Figure 3.3: *Pseudocode for our ray marching algorithm*

while loop. Here, sceneMaxT is the distance between the shrink ray end positions. It is used as the ray marching termination condition when the ray goes through the leaf nodes. The hit variable indicates whether or not the ray hits a leaf node which has geometric information in it. Since we use SVO ray marching primarily for VPL visibility tests, whenever the ray hits geometry, the algorithm terminates and returns immediately. If we were to implement glossy reflection in future, this SVO-restart method could be extended easily to fetch hit position geometric data. Lines 8 to 21 lists the main while loop. The idea is similar to that of kd-restart algorithm: marching the ray through the leaf nodes iteratively and checking if shadow ray occlusion occurs. The differences here are 1) how we determine through which subdivided region we should keep marching the ray and 2) how we clip the ray each time it intersects a subdivided region. In our implementation we use the octree nodes AABB, which is easy and efficient to be implemented on a GPU. The implementation on line 19 is based on the ray-AABB intersection detection algorithm described in PBRT [PH04]. In line 20 we update maxT using the result t1 from line 19, which is the maximum t value of the intersection position with the current nodes AABB. To make the algorithm numerically robust, we offset maxT using a user specified epsilon value. Otherwise the ray may stop marching forward in the next iteration due to inaccuracies caused by a floating point representation. Finally in line 21, the result is returned indicating if the ray passed the VPL visibility test.

3.3 Rendering

Since we replace the time-consuming VPL shadow map generation with a scene voxelization pass, the GPUs main burden changes from doing hundreds or even thousands of scene drawing and rasterization to a one-time scene voxelization and

subsequent shadow ray tests for indirect illumination. One important advantage of our method is that implementing the quasi-random walk [Kel97] for n-bounce VPL distribution is much more straightforward than shadow maps based real-time instant radiosity method. The reason is simple: we already have a voxelized scene representation, against which shooting rays is extremely efficient. We just needed to extend our VPL sampling pass by adding second-bounce VPL sampling from first-bounce VPLs sampled with reflective shadow maps. In complex scenes such as the Crytek Sponza, second and third-bounce VPLs are useful for illuminating the scenes complex geometry.

In our system, standard deferred shading with scene light shadow maps is applied to produce a direct illumination buffer. For indirect illumination, we accumulate all the VPLs that have influence on a shading fragment and store the results in an indirect illumination buffer. Since we have split the G-buffer into 4×4 tiles with an interleaved sampling pattern, all the fragments inside the same tile use exactly the same subset of VPLs. In this way, data access to VPLs and SVO nodes exhibits good locality, which in turn improves the performance of shadow ray marching tremendously. Similar to Dachsbacher et al [DS05], given a VPLs flux Φ_{VPL}, world space position p_{VPL}, world space normal n_{VPL}, shading surface points position p and normal n, the indirect illumination at a surface position due to a VPL is formulated as follows:

$$E_{VPL}(p, n, p_{VPL}, n_{VPL}) = \Phi_{VPL} G (1 - V) \tag{3.1}$$

$$G = \frac{max(0, cos\theta_i) max(0, cos\theta_o)}{max(B, |p - p_{VPL}|^2)} \tag{3.2}$$

$$cos\theta_i = dot(n, -\frac{p - p_{VPL}}{|p - p_{VPL}|}) \tag{3.3}$$

$$cos\theta_o = dot(n_{VPL}, \frac{p - p_{VPL}}{|p - p_{VPL}|}) \tag{3.4}$$

We fetch the VPL related information from our VPL buffer, which has been generated by the VPL sampling stage of the system. V is the visibility term between the VPL and shading surface point. To evaluate V, we have to perform a shadow ray test using our SVO structure. Note that for a shadow maps based instant radiosity method, evaluating V is just a simple shadow map lookup and comparison. In our case, since the evaluation of V is time-consuming, we could avoid a fruitless V term evaluation by first check the luminance of the VPL and the G term. If either the luminance of the VPL is too small or the G term equals zero, we then skip the attempted accumulation of this VPL to the indirect illumination buffer. B is the bouncing singularity constant factor used to clamp the distance between the VPL and the surface point. Since our technique currently only handles diffuse lighting, once the indirect illumination is done, the buffer is merged and filtered for final combination with the direct illumination buffer.

Chapter 4

Implementation

4.1 System Overview

In this chapter we present our real-time global illumination system in great detail. We exploit Shader Model 5 GPU programming features provided by OpenGL 4.3 to implement all sub-systems of our lighting system. Modern GPUs are designed as powerful computing devices that are not restricted to traditional computer graphics related tasks such as geometry rendering. The so-called general purpose GPU (GPGPU) programming technique has been introduced to solve problems covering many computation domains. Similar to other advanced real-time GI systems, our lighting system consists of a series of sub-system passes to process intermediate data generated from previous steps consecutively. As we will see later in this chapter that some of the sub-systems are designed and implemented as data consumers and producers by applying the idea of general purpose parallel computing. Figure 4.1 shows key sub-systems of our lighting system.

The main steps we performed for generating the final global illumination results are listed as follows:

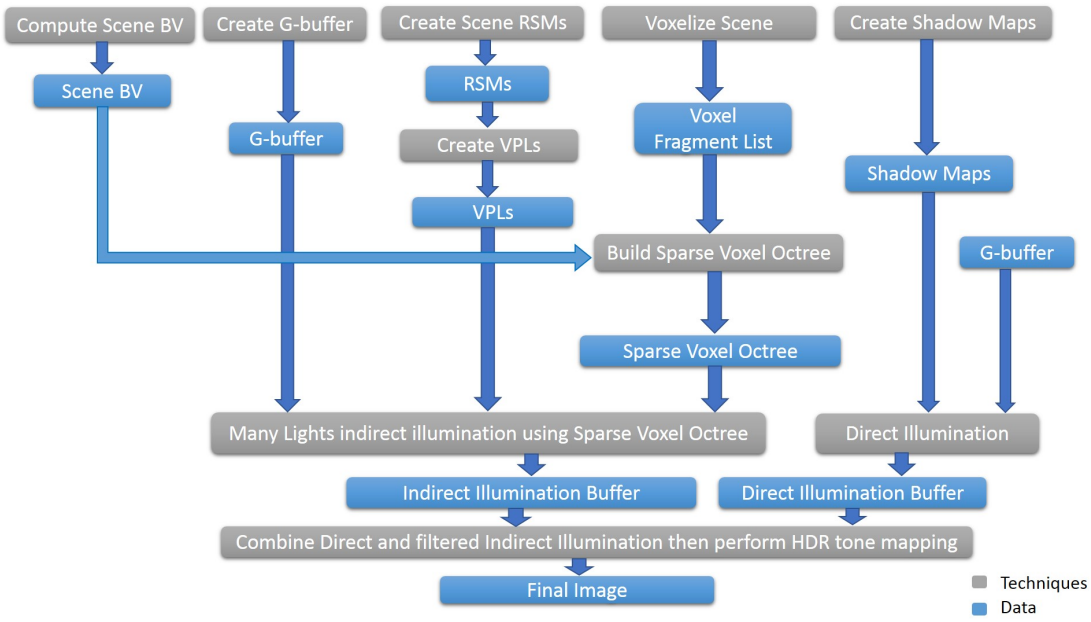

Figure 4.1: *Key sub-systems of our lighting system.*

1. Light uniform buffer management Scene lights information are gathered and transferred to GPU memory to be consumed by other steps of the system.

2. Scene voxelization A SVO is created dynamically by voxelizing the scene geometry. The visibility information is thus established.

3. Shadow maps generation For direct illumination, we capture the high frequency direct shadow using standard shadow map technique.

4. G-buffer generation G-buffer stores per-pixel shading information such as world position and normal needed by our deferred renderer.

5. Split G-buffer To improve GPU cache coherence, we group pixels using the same set of VPLs together such that the rendering speed increases.

6. Scene lights RSM generation RSMs are created for all the scene lights for first-bounce VPL sampling.

7. Sample RSMs First-bounce VPLs are generated using importance sampling on RSMs.

8. Deferred direct illumination In this step we perform standard direct illumination using G-buffer and shadow maps.

9. Deferred indirect illumination The indirect illumination from each VPL affecting the current pixel is accumulated into an indirect illumination buffer. We perform SVO ray marching for VPL visibility tests.

10. Merge indirect illumination buffer The indirect illumination results need to be merged back to form the original indirect illumination image.

11. Filter merged indirect illumination buffer A geometric-aware filter is applied to remove the noise patterns presented in the indirect illumination image.

12. Perform HDR tone mapping So far both the direct and indirect illumination results are stored as high dynamic range (HDR) images. To properly display them onto a monitor we need to perform HDR tone mapping to transform HDR values to values in range of [0, 1].

13. Display rendering results We display intermediate and final results in this step. Intermediate results are extremely helpful for debugging since intermediate data can be visualized and provide hints to indicate if there is something wrong.

Starting from the next section, we will walk through all the steps and give circumstantial explanations such that the reader could gain a clear understanding of how our system processes data and produces final image results.

4.2 Light Manager

We implement a light manager to manage all the scene lights presented in a specific scene. Currently we support point and spot lights. Light manager is responsible to gather all the lighting information and transfer them to GPU memory such that subsequent steps can fetch lighting related data as needed.

The idea here is that we only want to transfer lighting information to GPU once for each rendering frame to reduce CPU-GPU interoperability as less as possible since frequently transfer data between CPU and GPU may stall the GPU from execution while waiting for data to be transferred to GPU memory.

To efficiently upload data from CPU system memory to GPU memory, we use uniform buffer mechanism provided by OpenGL. We define a SceneLight struct to store necessary information to describe a scene light object. Then the scene light uniform buffer uses a 1-D array of SceneLight struct to store all scene lights information uploaded by the light manager. Figure 4.2 shows the SceneLight struct and the scene light uniform buffer.

4.3 Scene Voxelization

Following the idea of [CNS+11], we implement a SVO voxelizer which takes a set of scene triangle meshes as input and produces a SVO as output. The SVO generation is a multi-pass GPU algorithm. It first creates a voxel fragment list by rasterizing the input triangle meshes. A voxel fragment represents a potential voxel to be created at a specific location. Because multiple triangles could be rasterized at the same spatial location, multiple voxel fragments may be generated to create only one voxel. Therefore, the voxel fragment list may contain many redundant voxel fragments that need to be discarded when generating the SVO. After the voxel

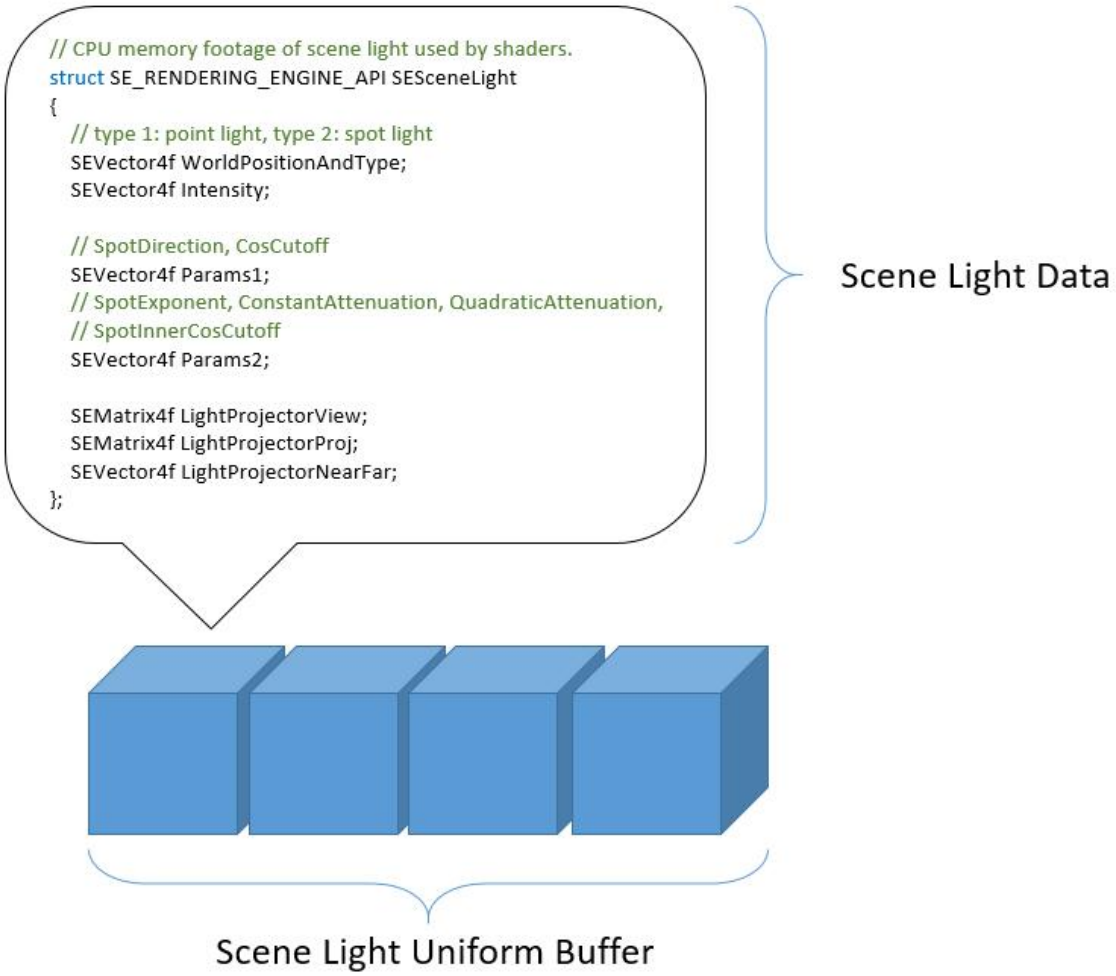

Figure 4.2: *Scene light data and its uniform buffer.*

fragment list is ready, the SVO voxelizer launches SVO building passes that take the voxel fragment list and scene bounding box as inputs then generate the final SVO into a SVO buffer.

4.3.1 Voxel Fragment List Generation

To create a voxel fragment list on GPU, we need to initialize a shader storage buffer provided by OpenGL. Basically, the shader storage buffer is viewed as a memory pool. When a GPU thread tries to create a voxel fragment, it acquires an allocation

from this memory pool and stores the result in it. To safely acquire allocations from the buffer, we also need a shader atomic operation counter to increase the next available location in the buffer atomically. This is achieved by using OpenGL atomic counter buffer and GLSL *atomicCounterIncrement* function.

The voxelization pass consists of vertex, geometry and fragment shaders. In the vertex shader we transform the input model space vertex to world space vertex. In the geometry shader we choose a world space axis along which we rasterize the input triangle primitive based on the triangles orientation. This is done by examining the dot products of the triangle face normal and the axes (x, y, z basis in world space). The idea here is to maximize the area along which the triangle primitive occupies such that enough voxel fragments can be generated by the rasterizer. Note that currently we have not implemented a conservative rasterization algorithm, which is an ideal solution generating voxel grid without holes. After the optimal axis is chosen, view and projection matrices are calculated and applied to transform the world space triangle to homogenous space triangle for rasterization. Finally, in the fragment shader we use GLSL *atomicCounterIncrement* function to increase the voxel fragment list counter to acquire the next available location and append a new voxel fragment to this location.

For the purpose of VPL visibility tests, the only mandatory data stored in a voxel fragment is its voxel grid coordinates. As we have mentioned in Chapter 3, encoding SVO node AABB to two unsigned integer values minimizes memory footprint used by SVO nodes and improves the SVO ray marching performance. Following the same convention, here we encode the voxel grid coordinates to one unsigned integer value as well.

4.3.2 SVO Generation

After the voxel fragment list is generated. The SVO voxelizer generates the tree level by level in an iterative fashion. The nature of this algorithm is a non-recursive multi-pass multi-threading tree construction procedure. While modern GPUs support recursion and calling stacks, it is very inefficient implementing a GPU algorithm in this way. In light of this, we try to realize GPU algorithms taking non-recursion, thread balance and multi-threading concurrency into consideration. The SVO generation algorithm is an example following this guideline and is listed as the following steps:

1. **Gather voxel fragment list information:** In this step we launch a single GPU thread via compute shader to fill out an indirect command buffer. The reason is that we want to dispatch a GPU thread for each item in the voxel fragment list and not let CPU interfere the procedure. Recall that the number of voxel fragments is stored in GPU memory as an atomic counter. Thus, instead of reading this value back to the system memory and let CPU dispatch threads based on it, we use indirect command buffer provided by OpenGL to dispatch GPU threads indirectly. This scheme is heavily used often when implementing a multi-pass GPGPU algorithm.

2. **Initialize SVO root:** If the voxel fragment list is not empty, we start building the tree level by level. The steps are:

   (a) We allocate one SVO node tile for the root.

   (b) Update the current level node tile range.

   (c) Update the indirect command buffer for SVO allocating node pass.

   (d) Create an SVO root node bounding box.

(e) Create the first level node bounding boxes.

3. **Set the level of the current tree being built to 1:** On the CPU side we need to use a loop to control how many levels we want to build. We start building the tree from level 1.

4. **Update the SVO uniform buffer:** Store the value of the current tree level to let the GPU threads know which SVO level is currently being worked on, a uniform buffer is used to update this value.

5. **Flag SVO nodes:** By using the voxel fragment lists indirect command buffer, we dispatch a GPU thread for each voxel fragment to flag the SVO nodes. Each thread starts from the SVO root and descends to the level of the tree currently being built based on its voxel fragment grid coordinates (recall that when we built the voxel fragment list, we stored the voxel fragment grid coordinates in it). Finally, the thread flags the node in the current level being built indicating whether there is some geometry occupying the space covered by the node.

6. **Allocate new SVO nodes:** Based on the number assigned to the SVO indirect command buffer, the corresponding number of GPU threads are dispatched to potentially allocate new node tiles for the current level being built. Each thread locates its node being built and checks if the node has been flagged. If it has, the thread then increments the atomic counter to get a new location for the new node tile. Afterwards, the current node is updated by pointing to the new children node tile and is masked as a non-leaf node. The children node tiles bounding boxes are created as well.

7. **Post-process step after allocating SVO nodes:** After children node tiles are allocated for the current level, we dispatch a single GPU thread to update

variables in the SVO indirect command buffer for the next tree building iteration. Specifically, the thread updates the current level node tile range, the number of threads needed for allocating new nodes and primitive count for the SVO visualization pass.

8. **Initialize SVO nodes:** We dispatch GPU threads to initialize the new SVO node tiles by marking them as leaf nodes and assigning their user data pointer to null.

9. **Increase the current tree level value by 1.**

10. **Go to step 4 until the maximum tree level is reached.**

4.3.3 GPU Program Debugging

When implementing complex GPU multi-threading algorithms such as the aforementioned SVO generation algorithm, debugging was not as easy as programming CPU code. Till date, there is no GLSL shader program debugger supporting Shader Model 5 features. Thus, we designed two ways assisting us in finding bugs in our GPU programs.

The first method we use is called auxiliary debugging buffers. The idea is that we allocate GPU shader storage buffers to cache the values we want to inspect. Additionaly, we wrote debug code to output those values manually. While at first glance this approach appears cumbersome, it is very effective in analyzing multi-threaded GPU algorithms. The second method we use is data visualization. The SVO data structure can be observed to gain an understanding of its correctness. Figure 4.3 shows an visualization of the Cornell box scenes SVO. Here, the SVO non-leaf nodes' bounding volumes are rendered as green boxes whereas the leaves are rendered as blue boxes, indicating the correctness of the hierarchical tree structure.

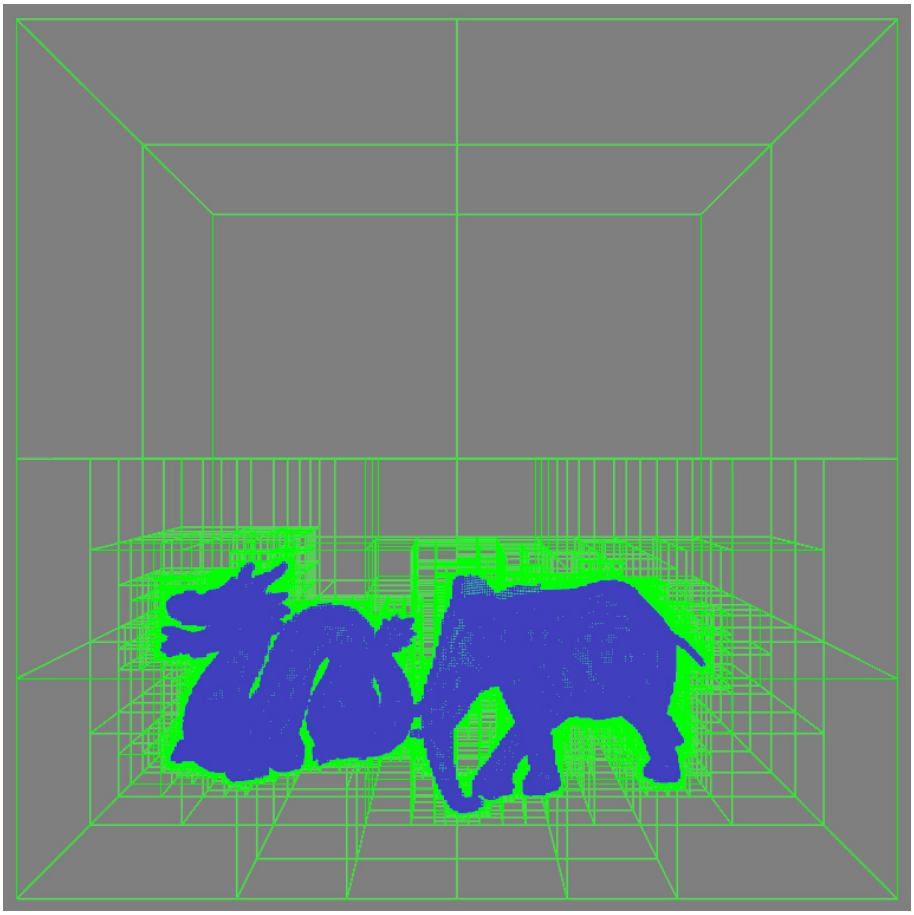

Figure 4.3: *SVO visualization for the Cornell box scene with dragon and running elephant models.*

4.4 Shadow Maps Generation

In our lighting system, standard shadow maps are generated for scene lights and used for direct illumination. We implement a shadow maps generator that takes the light manager, the shadow casters and shadow maps description as input. In the initialization phase, the shadow map generator figures out how many shadow maps are needed based on the numbers of point and spot lights. It then creates a 2D texture array and a shadow map information uniform buffer. Currently there is only one unsigned integer value that resides in the uniform buffer indicating which lights shadow map the generator is rendering. During the shadow map generation

pass, the shadow maps generator submits the shadow casters for each scene lights. The resulting 2D texture array holds all the shadow maps for later use.

4.5 G-buffer Generation

Camera G-buffer is a mandatory data source for deferred renderers. Similar to other real-time GI systems, we use the G-buffer for both direct and indirect illumination. We do not apply G-buffer compression techniques to save GPU memory bandwidth because for our target platform the G-buffer filling rate is not a bottleneck. We creates the G-buffer with three 4-channel floating-point render targets:

1. Render target 0 for world positions.
2. Render target 1 for world normal.
3. Render target 2 for albedo (diffuse materials).

Figure 4.4 shows an example of our G-buffer.

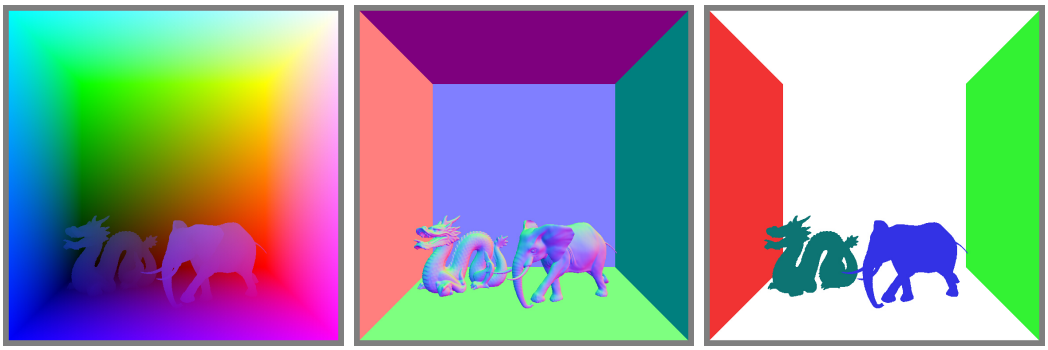

Figure 4.4: *G-buffer visualization for the Cornell box scene with dragon and running elephant models.*

4.6 Split G-buffer

Grouping G-buffer pixels using the same set of VPLs together makes the GPU execute multiple threads much more efficiently than a naïve G-buffer pixel layout. Therefore, we implement a G-buffer splitter to split the G-buffer into 4×4 tiles. After the G-buffer is generated, a two-triangle quad is submitted to trigger fragment shader invocations for the splitting algorithm. Note that we only use the split G-buffer for indirect illumination. Thus, there is no need to split the albedo buffer because material information is only applied in the final combination phase of direct and indirect illumination. Given the coordinates of current fragment coords, the tile width and height tileWH, the tile numbers in x and y directions mn, the position uv where the fragment shader fetches data from the G-buffer can be formulated as follows:

$$uv = ab \times mn + ij \tag{4.1}$$

$$ab = coords \,\%\, tileWH \tag{4.2}$$

$$ij = coords \div tileWH \tag{4.3}$$

Figure 4.5 shows an example of our split G-buffer.

4.7 Scene Lights RSM Generation

Similar to shadow maps generation, we implement a RSM generator that takes the light manager as input and creates reflective shadow maps for all the scene lights. RSMs are similar to G-buffer in that they store world space positions, normals and surface properties (flux in our case). The difference here is that instead of

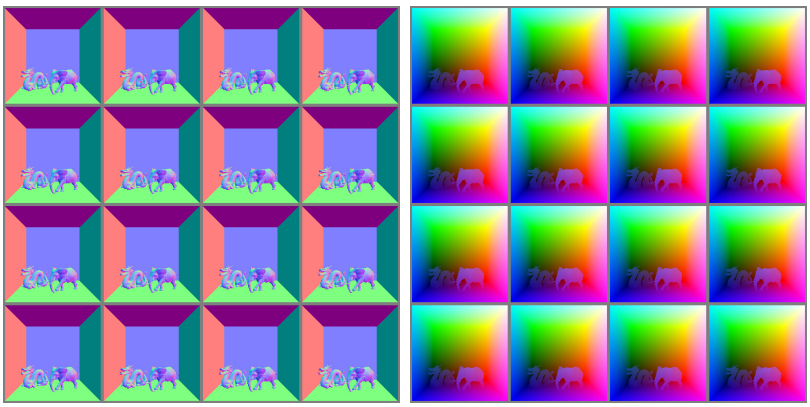

Figure 4.5: *Split G-buffer visualization for the Cornell box scene with dragon and running elephant models.*

using main camera, we have to apply individual scene lights view and perspective transformation matrices to generate the buffers. Fortunately, we already have our scene light related parameters grouped together and managed by the light manager. We can efficiently fetch data from the light uniform buffer as mentioned in section 4.2.

Our RSMs are created with three 4-channel floating-point textures and organized as follows:

1. Render target 0 stores world positions viewed from a light source.
2. Render target 1 stores world normal viewed from a light source.
3. Render target 2 stores flux viewed from a light source.

Figure 4.6 shows an example of RSM.

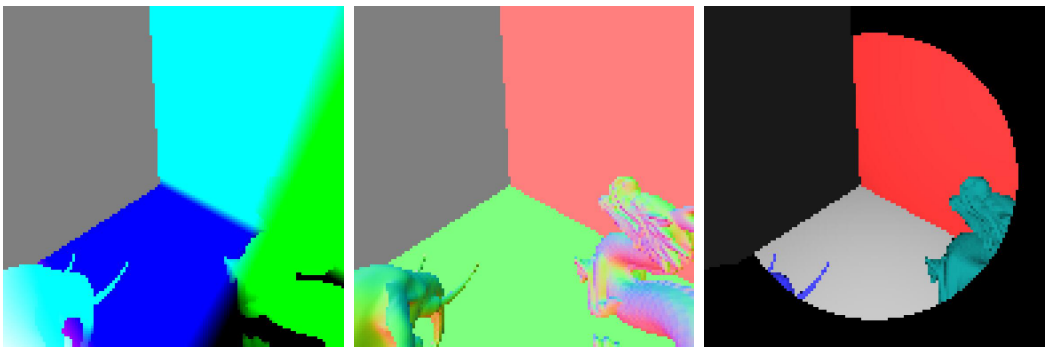

Figure 4.6: *RSM visualization for the Cornell box scene with dragon and running elephant models. A spot light is used to generate the RSM.*

4.8 VPL Generation

We create a VPL generator to sample the RSMs and store the sampling results into a shader storage buffer (VPL buffer). During the initialization phase, the VPL generator caches the RSM 2D texture arrays as its input data. Then it creates a VPL sampling pattern, which is a 1D 4-channel floating-point texture storing random numbers in range [0, 1]. Here, instead of using uniformly distributed random numbers, we employ a low-discrepancy sequence (also called quasi-random sequence) due to the requirement of quasi-Monte Carlo integration for computing indirect illumination. The basic idea is to sample the incoming radiance field more evenly and reduce the integration variance while using as few samples as possible. The low-discrepancy sequence we use is a Halton sequence generated using radical inverse function that takes prime numbers and halton index as inputs. The resulting data is first generated on system memory and then transferred to GPU texture memory.

In addition to the input data preparation mentioned above, the VPL generator also creates a so-called GPU compute task to consume the input data and produce the VPLs. In our system, a GPU compute task is designed and implemented as a general purpose GPU multi-threading task that is independent of the rendering pipeline. Thus, implementing a general algorithm on our system is much easier than

doing so using a traditional rendering-pipeline based 3D graphics engine.

Our VPL sampling algorithm is based on the technique introduced by Claberg et al [CJAMJ05]. The basic idea of this technique is applying importance sampling to BRDF and incoming radiance field, which are complex spatial variant functions that must be sampled according to their distributions. Figure 4.7 illustrates this technique.

Figure 4.7: *Importance sampling of BRDF and environment map. (Image courtesy of [CJAMJ05]).*

Since we only consider diffuse surface materials for indirect illumination, our importance sampling strategy thus only generates samples according to the incoming radiance field described in the RSM flux map. Following the idea of Claberg et al [CJAMJ05] and Knecht [Kne09], we perform a one-pass iterative importance sampling procedure using our GPU compute task scheduling mechanism. The idea of this iterative algorithm is illustrated in Figure 4.8.

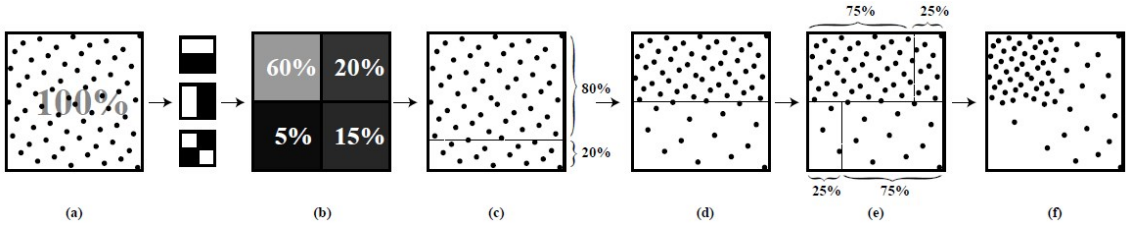

Figure 4.8: *One iteration of importance sampling using warping input points. (a) shows an initial random distribution. (b) A 2×2 image representing an importance distribution. (c) and (d) show a vertical warping step, input points are redistributed based on importance distribution in (b). (e) and (f) show two horizontal warping steps performed after the vertical warping step. (Image courtesy of Claberg et al [CJAMJ05]).*

To support importance sampling of RSM flux map, we create a mip-map hierarchy and let the RSM generator generates it dynamically. Note that it is not necessary for generating a mip-map for RSM position and normal maps since our importance distribution function is derived only from lighting intensity. To generate n VPLs, we dispatch a GPU compute task with n GPU threads, each of which starts from the highest mip-map level of the RSM flux map and warps its samples position iteratively until level 0 is reached. Figure 4.9 shows a visualization of VPL importance sampling.

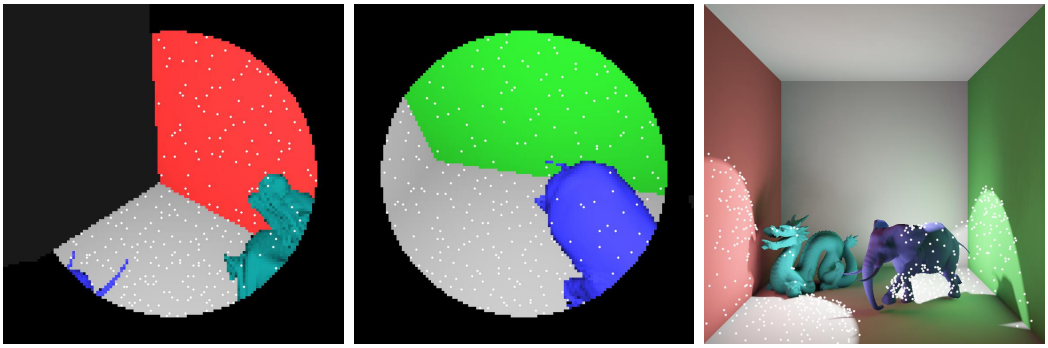

Figure 4.9: *VPL importance sampling. Two spot lights are sampled using their RSM flux maps to generate 512 VPLs dynamically.*

4.9 Deferred Direct Illumination

Direct illumination is implemented using a standard deferred shading technique: A full screen triangle quad is submitted to trigger fragment shaders which perform G-buffer data fetch, scene lighting computation and shadow map lookups and comparison. To mimic direct soft shadow effect, we apply a simple shadow map Percentage-Closer Filter (PCF) [RSC87] to compute an in-shadow strength for shading the fragments. The rendering results are stored in the direct illumination buffer. Figure 4.10 shows an example of direct illumination.

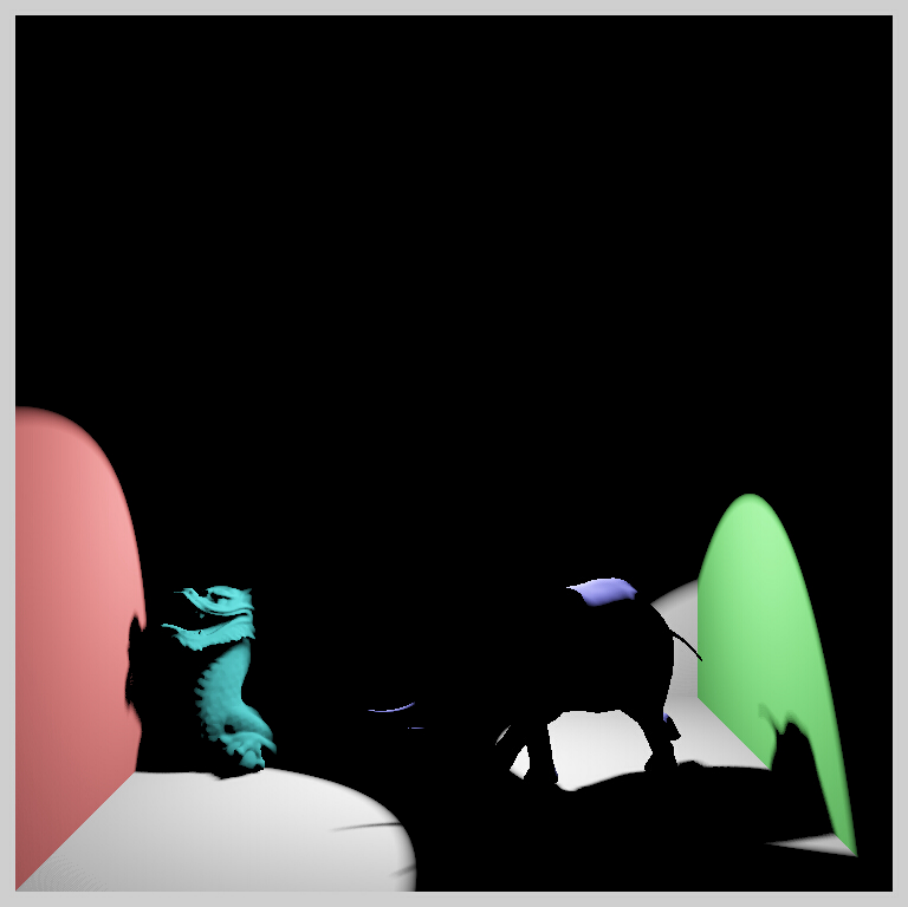

Figure 4.10: *Deferred direct illumination of the Cornell box scene.*

4.10 Deferred Indirect Illumination

We implemented an indirect illumination renderer that takes the split G-buffer, SVO buffer and VPL buffer as input. It generates the indirect illumination buffer using the same deferred shading technique as direct illumination renderer: a full screen triangle quad is used to trigger fragment shaders for shading the input split G-buffer.

The indirect illumination algorithm implemented by the fragment shader can be described as the following steps:

1. Fetch world position and normal from the split G-buffer.

2. Figure out how many VPLs are needed for the tile in which this fragment resides.

3. Figure out an index position from where the VPL subset will be fetched.

4. Initialize a local SVO root node variable which will be used for marching the SVO in parallel.

5. Set indirect color to zero.

6. Fetch a VPL from VPL buffer.

7. Calculate the VPLs luminance.

8. Compute the G term.

9. If the VPLs luminance is greater than a preset epsilon and the G term is greater than zero, perform VPL visibility test using the SVO ray-marching algorithm introduced in Chapter 3. Then possibly accumulate the VPLs contribution to the indirect color.

10. Go to step 9 until all the VPLs in the subset have been accumulated.

11. Evaluate the Monte Carlo integration for diffuse indirect illumination.

12. Output the final indirect color to indirect illumination buffer.

Figure 4.11 shows the result of indirect illumination. VPL subsets for each tile are visualized as well.

Figure 4.11: *Deferred indirect illumination using split G-buffer and VPL interleaved sampling.*

4.11 Merge Indirect Illumination Buffer

In this stage we merge the split indirect illumination buffer to form the original buffer. Similar to direct and indirect illumination, a full screen quad is used to trigger the merge shader. Given the coordinates of current fragment coords, the tile width and height tileWH, the tile numbers in x and y directions mn, the position uv where the fragment shader fetches data from the split indirect illumination buffer can be formulated as follows:

$$uv = ij \times tileWH + ab \tag{4.4}$$

$$ij = coords \,\%\, mn \tag{4.5}$$

$$ab = coords \div mn \tag{4.6}$$

The merged buffer is shown in figure 4.12.

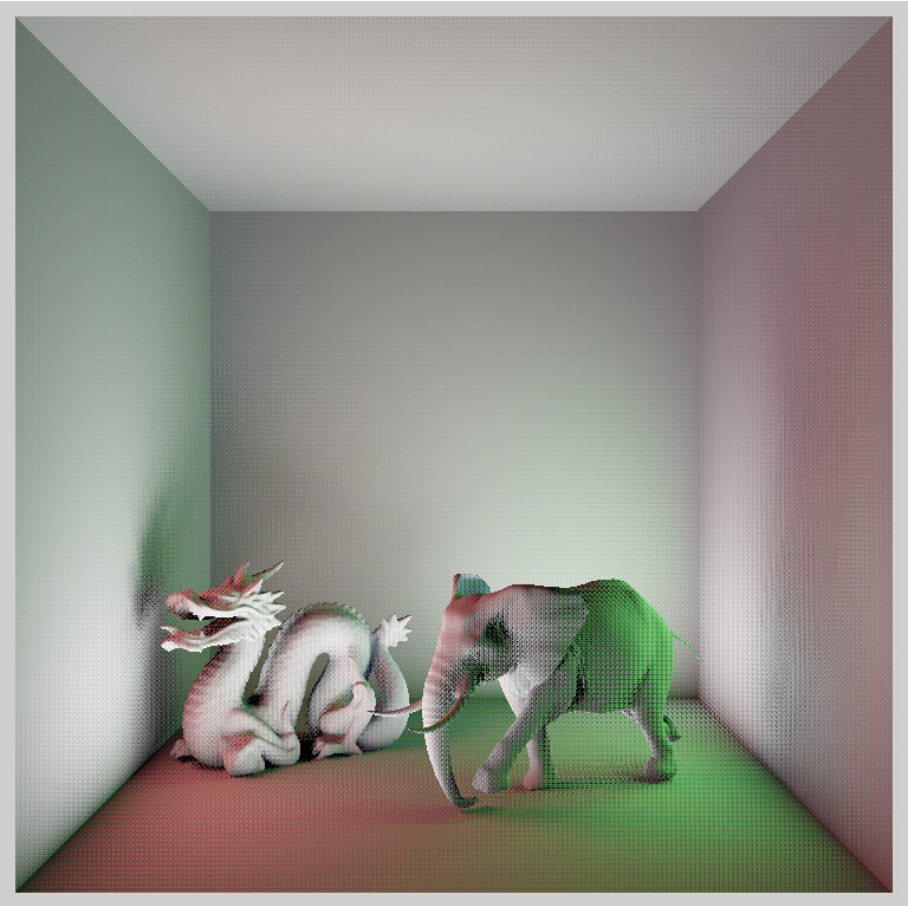

Figure 4.12: *Merged indirect illumination buffer.*

4.12 Geometric-aware Filtering and HDR Tone Mapping

Geometric-aware Filtering

The indirect illumination buffer generated from the previous step must be filtered to

remove the noise pattern introduced by VPL interleaved sampling. Here we adopt a technique similar to Laine et al [LSK+07]. The basic idea is to apply a box filter to each pixel based on geometric thresholds specified by the user. World position and normal difference are used for threshold comparison. If a neighboring pixel covered by the filter kernel passed the threshold comparison, its contribution is then accumulated to the final image. Figure 4.13 shows a filtered result of merged indirect illumination buffer.

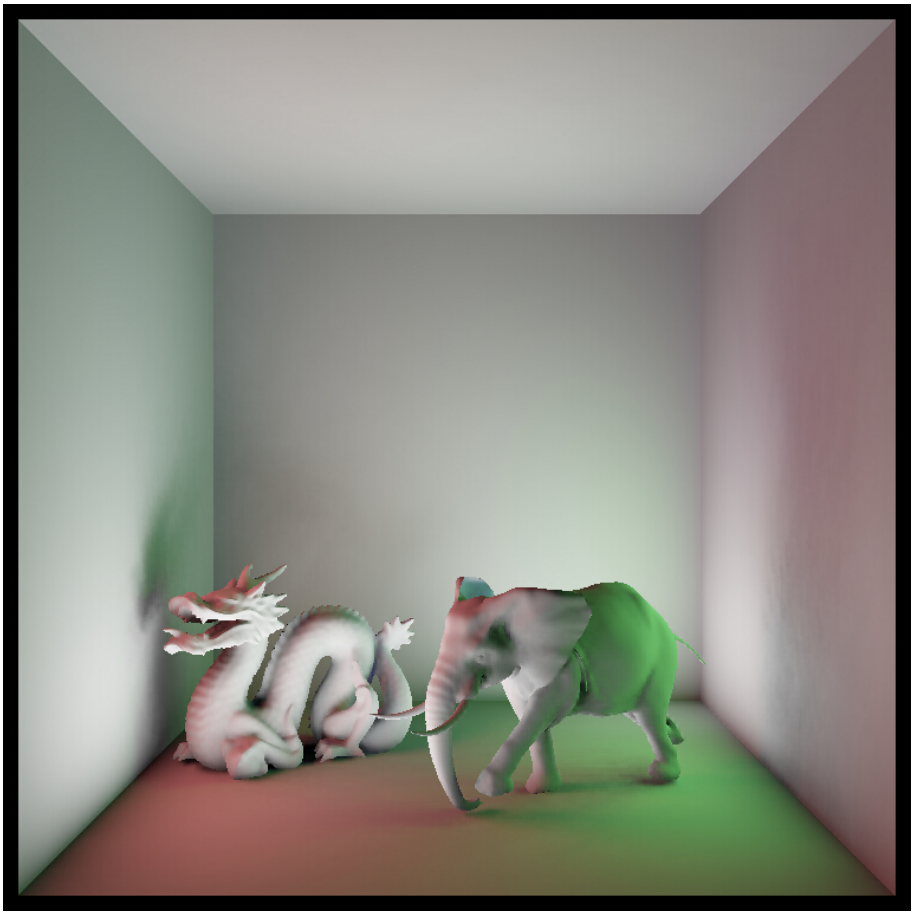

Figure 4.13: *Applying geometric-aware filter to the merged indirect illumination buffer.*

HDR Tone Mapping

Physically based global illumination requires lighting computations to be performed

with colors of high dynamic range (HDR). To support HDR lighting, we store all intermediate lighting results using floating-point textures such that values are not restricted to the range of [0, 1]. In the final combination and intermediate illumination buffers visualization steps we apply a simple local HDR operator to map the HDR values to low dynamic range (LDR) values suitable for display. For more information on HDR image processing we encourage the reader to reference the book on this topic by Reinhard et al [RHD$^+$10].

Chapter 5

Results

In this chapter, we present the rendering results of our real-time global illumination method. Our testing platform is an Intel i7 3930k CPU with an NVIDIA GeForce Titan X GPU. We tested our lighting system with two scenes: A Cornell box with complex models (dragon and running elephant) and Crytek Sponza. All scenes are fully dynamic and no preprocessing methods are required.

Table 5.1 shows the time spent on critical stages of our system in milliseconds. The Cornell box scene is rendered at 768×768 (128^3 SVO) and Crytek Sponza scene is rendered at 1280×720 (256^3 SVO). The G-buffer, direct illumination buffer and indirect illumination buffer all have the same resolution as the back-buffer. No down-sampling is performed. For interleaved sampling of VPLs, the G-buffer is split into 4×4 tiles such that the VPL set is divided into 16 subsets, each of which is assigned to a G-buffer tile. The number of VPLs is varied and their influence on rendering speed and image quality is observed. Fig.5.1 shows the Cornell box scene rendered with 512 and 2048 VPLs respectively. Note that the visual difference between two settings is very small due to the accurate VPL visibility tests, which makes the integration of incoming radiance from VPLs converge very quickly. We also tested

	Cornell box with dragon and running elephant	Crytek Sponza
Scene Voxelization and SVO Generation	0.57 / 0.57 / 0.58	4.65 / 4.65 / 4.65
RSM rendering	0.46 / 0.46 / 0.46	0.61 / 0.62 / 0.62
VPL Generation	0.02 / 0.02 / 0.02	0.02 / 0.02 / 0.02
Direct Illumination	0.17 / 0.17 / 0.18	0.31 / 0.31 / 0.31
Indirect Illumination	16.0 / 33.0 / 69.0	19.5 / 36.2 / 73.0

Table 5.1: *Detailed timings for the Cornell box (*128^3 *SVO,* 768×768 *resolution, no MSAA) and Crytek Sponza scene (*256^3 *SVO,* 1280×720 *resolution, no MSAA) measured in milliseconds for critical stages of our lighting system. Three VPL number settings are used: 512, 1024 and 2048. The hardware environment is an Intel i7 3930k CPU with an NVIDIA GeForce Titan X GPU*

the Cornell box scene using a SVO resolution of 256^3, but the visual improvement is negligible. For the Crytek Sponza scene, one-bounce indirect illumination is not enough to illuminate all the regions of the scene. Artifacts may appear in regions where an insufficient number of VPLs are accumulated. Fig.5.2 shows the rendering result of the sponza scene using 1024 and 2048 first-bounce VPLs.

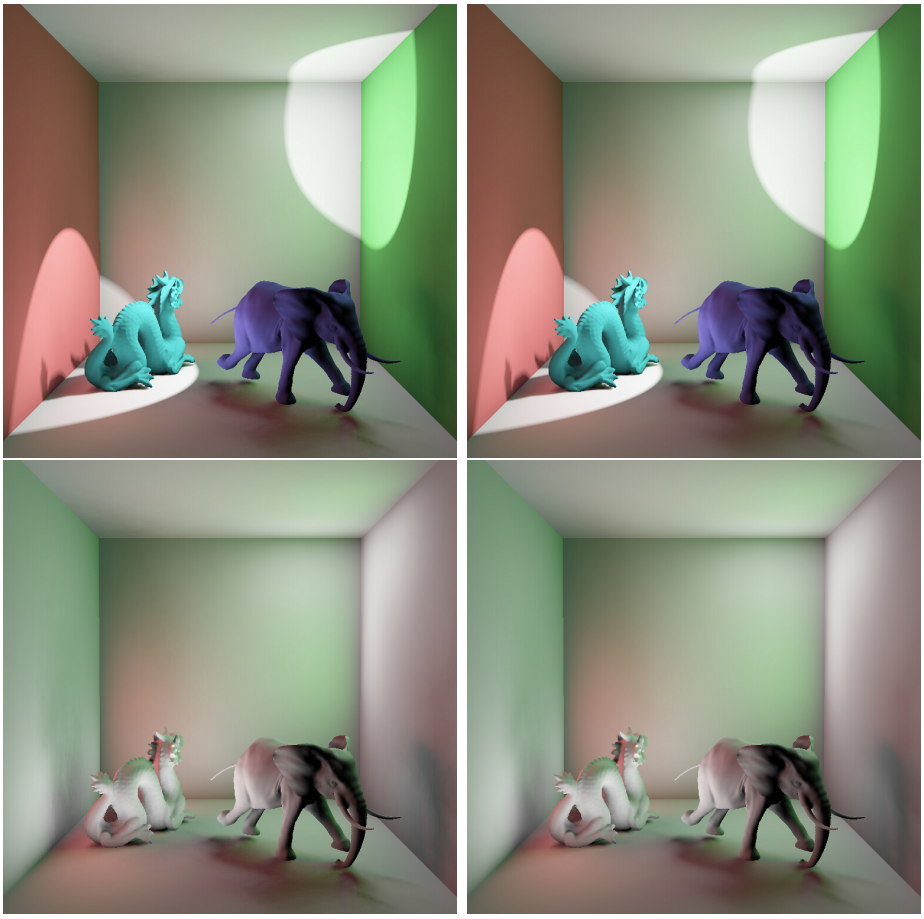

Figure 5.1: *Fully dynamic Cornell box scene with dragon and running elephant (150k triangles) rendered using two spot lights. SVO grid dimension is set to* 128^3*. Top left: Final image (512 first-bounce VPLs, 50 fps). Top right: Final image (2048 first-bounce VPLs, 15 fps). Bottom left: Indirect illumination (512 first-bounce VPLs, 50 fps). Bottom right: Indirect illumination (2048 first-bounce VPLs, 15 fps)*

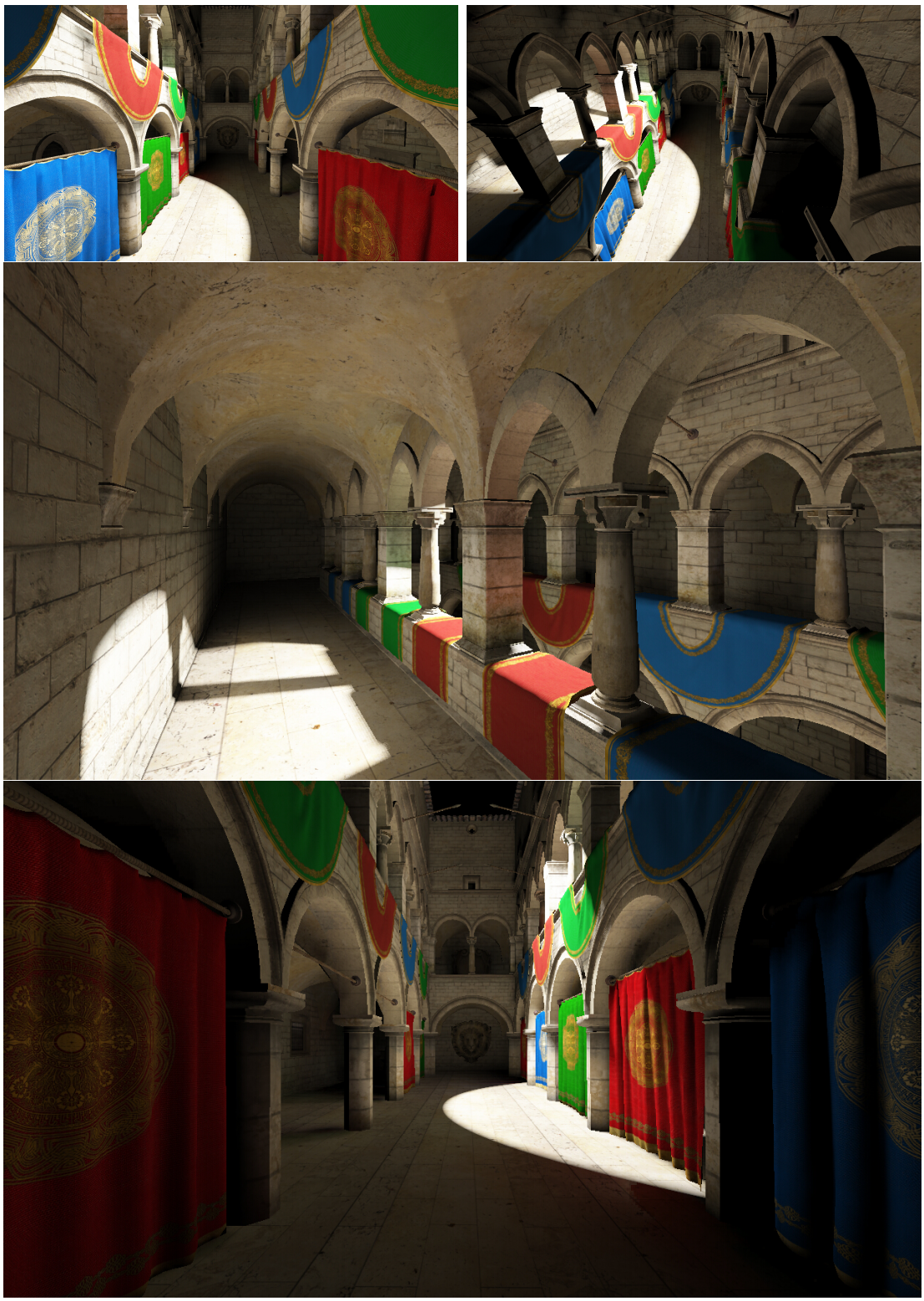

Figure 5.2: *Fully dynamic Crytek Sponza scene (180k triangles) rendered using one spot light. SVO grid dimension is set to* 256^3*. Top left: Final image of camera position 1 (1024 first-bounce VPLs, 19 fps). Top right: Final image of camera position 2 (1024 first-bounce VPLs, 20 fps). Middle: Final image of camera position 3 (1024 first-bounce VPLs, 23 fps). Bottom: Final image of camera position 4 (2048 first-bounce VPLs, 11 fps).*

Chapter 6

Conclusion

We have presented an alternate method to perform VPL visibility tests for instant radiosity-based real-time global illumination applications. While our method is not as efficient as fast shadow maps based methods for computing one-bounce indirect illumination, it may be promising to perform well for second- and third bounce VPL sampling and generation in complex scenes facilitating the use of ray tracing while maintaining real-time frame rates. Our method is automatic and supports fully dynamic scenes, no scene preprocessing or input data specification is needed.

Similar to other real-time instant radiosity techniques, our method inherits limitations from instant radiosity. For instance, highly glossy surfaces cannot be reconstructed without adequate VPLs distributed evenly in the entire scene. Meanwhile, applying interleaved sampling to scene surfaces not directly illuminated exhibits noticeable artifacts if a shading surface point fetches VPLs, most of which are blocked due to poor VPL distribution. To solve these issues, as mentioned before, our next step is implementing a robust VPL distribution and gathering method. We believe it is worth attempting by taking advantage of the SVO data structure, using it as a scene geometric data representation for n-bounce VPL generation and

rendering-time VPL acquisition. To support large scale scenes, a cascaded SVO grid management scheme could be implemented by following the idea of cascaded Light Propagation Volume (LPV) [KD10] as well.

Bibliography

[AL09] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proc ACM Conf High Performance graphics 2009*, pages 145–149, 2009.

[BAS02] Stefan Brabec, Thomas Annen, and Hans-Peter Seidel. Shadow mapping for hemispherical and omnidirectional light sources. In *Advances in Modelling, Animation and Rendering*, pages 397–407. Springer, 2002.

[Cha13] Subrahmanyan Chandrasekhar. *Radiative transfer*. Courier Corporation, 2013.

[CJAMJ05] Petrik Clarberg, Wojciech Jarosz, Tomas Akenine-Möller, and Henrik Wann Jensen. Wavelet importance sampling: efficiently evaluating products of complex functions. In *ACM Transactions on Graphics (TOG)*, volume 24, pages 1166–1175, 2005.

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

[DKH+14] Carsten Dachsbacher, Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Bruce Walter, and Jan Novák. Scalable realistic rendering with many-light methods. In *Computer Graphics Forum*, volume 33, pages 88–104. Wiley Online Library, 2014.

[DS05] Carsten Dachsbacher and Marc Stamminger. Reflective shadow maps. In *Proc.ACM Symposium on Interactive 3D graphics and games*, pages 203–231, 2005.

[FBP09] Vincent Forest, Loic Barthe, and Mathias Paulin. Real-time hierarchical binary-scene voxelization. *Journal of Graphics, GPU, and Game Tools*, 14(3):21–34, 2009.

[FS05] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proc ACM SIGGRAPH/EUROGRAPHICS conf Graphics hardware*, pages 15–22, 2005.

[GD98] JP Grossman and William J Dally. *Point sample rendering*. Springer, 1998.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive kd tree gpu raytracing. In *Proc ACM Symp Interact. 3D graphics and games*, pages 167–174, 2007.

[Kaj86] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150, 1986.

[KD10] Anton Kaplanyan and Carsten Dachsbacher. Cascaded light propagation volumes for real-time indirect illumination. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 99–107, 2010.

[Kel97] Alexander Keller. Instant radiosity. In *Proc ACM SIGGRAPH*, pages 49–56, 1997.

[KH01] Alexander Keller and Wolfgang Heidrich. *Interleaved sampling*. Springer, 2001.

[Kne09] Martin Knecht. *Real-time global illumination using temporal coherence*. 2009.

[LSK+07] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Proc. Eurographics Conf Rendering Techniques*, pages 277–286. Eurographics Association, 2007.

[MKC07] Ricardo Marroquim, Martin Kraus, and Paulo Roma Cavalcanti. Efficient point-based rendering using image reconstruction. In *SPBG*, pages 101–108, 2007.

[OBM06] Brian Osman, Mike Bukowski, and Chris McEvoy. Practical implementation of dual paraboloid shadow maps. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 103–106. ACM, 2006.

[PH04] Matt Pharr and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2004.

[RGK+08] Tobias Ritschel, Thorsten Grosch, Min H Kim, H-P Seidel, Carsten Dachsbacher, and Jan Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics (TOG)*, 27(5):129, 2008.

[RHD+10] Erik Reinhard, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski. *High dynamic range imaging: acquisition, display, and image-based lighting*. Morgan Kaufmann, 2010.

[RSC87] William T Reeves, David H Salesin, and Robert L Cook. Rendering antialiased shadows with depth maps. In *ACM Siggraph Computer Graphics*, volume 21, pages 283–291. ACM, 1987.

[SIMP06] Benjamin Segovia, Jean Claude Iehl, Richard Mitanchey, and Bernard Péroche. Non-interleaved deferred shading of interleaved sample patterns. In *Proc ACM SIGGRAPH/Eurographics Symposium on Graphics hardware: Vienna, Austria*, volume 3, pages 53–60, 2006.

[SIP07] Benjamin Segovia, Jean Claude Iehl, and Bernard Péroche. Metropolis instant radiosity. In *Computer Graphics Forum*, volume 26, pages 425–434. Wiley Online Library, 2007.

[SiS96] Mateu Sbert and Xavier Pueyo i Sàndez. *The Use of global random directions to compute radiosity: global Montecarlo techniques*. 1996.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 197–206, 1990.

[THGM11] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Müller. Voxel-based global illumination. In *ACM Symposium on Interactive 3D Graphics and Games*, pages 103–110, 2011.

[TO12] Yusuke Tokuyoshi and Shinji Ogaki. Real-time bidirectional path tracing via rasterization. In *Proc. ACM Symp/ Interactive 3D Graphics and Games*, pages 183–190, 2012.

[YHGT10] Jason C Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. In *Computer Graphics Forum*, volume 29, pages 1297–1304. Wiley Online Library, 2010.