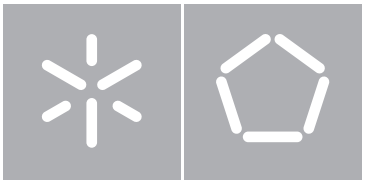
Universidade do Minho
Escola de Engenharia

Daniel José Taveira Gomes

Voxel Based Real-Time Global Illumination Techniques

Abril de 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Daniel José Taveira Gomes

Voxel Based Real-Time Global Illumination Techniques

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de
Professor António Ramires Fernandes

Abril de 2015

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Professor Ramires for his patient guidance and valuable suggestions and critiques throughout the development of this thesis.

I would also like to express my gratitude to all my friends, who heard my complaints when things did not go as planned and provided advice whenever they could.

Finally, I wish to thank my parents for their invaluable support and encouragement during my studies.

ABSTRACT

One of the goals in computer graphics is to be able to generate photo-realistic images and do it in real time. Unfortunately most lighting algorithms are not able to satisfy both objectives at the same time.

Most of the algorithms nowadays are based on rasterization to generate images in real time at the expense of realism, or based on ray tracing, achieving photo-realistic results but lacking performance, which makes them unsuitable to compute at interactive frame rates with the computational power currently available.

Over the last years, some hybrid approaches have emerged that try to combine the best features of both types of algorithms.

What is proposed in this thesis is the study and analysis of a class of algorithms based on voxels to approximate global illumination in 3D scenes at interactive frame rates.

The goal of this study is an analysis on the practicability of such algorithms in real-time applications. It is shown that these techniques are very similar in their structure. They use a volumetric pre-filtered representation of the scene and a rendering algorithm based on ray tracing or cone tracing to compute an approximation to global illumination in real time.

The main problem of filtering the appearance of objects in the scene using voxels is that these representations can require quite a large quantity of memory. Most of the described algorithms try to deal with this problem in some way and so multiple data structures are presented to try to solve the problem.

Another approach is described that uses the wasted space of a regular data structure to compute volumetric effects together with the approximation to global illumination, in real time.

A selection of the described algorithms are implemented and analysed in terms of performance and visual quality of the resulting image and some conclusions are drawn, describing some possible future improvements.

RESUMO

Um dos maiores objetivos da computação gráfica é conseguir gerar imagens fotorealistas e em tempo real. Infelizmente os algoritmos atuais de iluminação não conseguem atingir ambos os objetivos simultaneamente.

A maioria dos algoritmos hoje em dia baseiam-se na rasterização para gerar imagens em tempo real, à custa da perda de realismo, ou então baseados em *ray-tracing*, conseguindo obter imagens fotorealistas, à custa da perda de interatividade.

Nos últimos anos têm surgido novas técnicas para tentar juntar o melhor dos dois tipos de algoritmos.

Propõe-se neste trabalho o estudo e análise de uma classe de algoritmos baseados em *voxels* para calcular uma aproximação à iluminação global, de forma interativa.

Através deste estudo pretende-se por um lado analisar a viabilidade dos algoritmos em aplicações em tempo real e aplicar as novas capacidades da API do OpenGL de forma a simplificar/otimizar a sua implementação.

É demonstrado que estas técnicas possuem uma estrutura muito semelhante. Estas técnicas usam uma pré-filtragem da cena usando uma representação volumétrica da cena e um algoritmo baseado em *ray tracing* ou *cone tracing* para calcular uma aproximação da iluminação global em tempo real.

O problema principal gerado pela necessidade de usar uma pré-filtragem da cena usando *voxels* é que este tipo de representações requerem usualmente uma grande quantidade de memória. A maioria dos algoritmos descritos tentam lidar com este problema de alguma forma e múltiplas estruturas de dados são apresentadas para esse efeito.

Outra abordagem é descrita que utiliza o espaço desperdiçado de uma estrutura regular para computar efeitos volumétricos em conjunto com a aproximação à illuminação global em tempo real.

Uma selecção dos algoritmos descritos é implementada e analisada em termos de desempenho e da qualidade visual da imagem gerada e algumas conclusões são tiradas, descrevendo alguns melhoramentos possíveis.

CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

1

INTRODUCTION

One of the greater objectives in computer graphics is to generate fotorealistic images.

The efficient and realistic rendering of scenes on a large scale and with very detailed objects is a great challenge, not just for real-time aplications, but also for offline rendering (e.g. special effects in movies). The most widely used techniques in the present are extremely inefficient to compute indirect illumination, and the problem is aggravated for very complex scenes, since calculating the illumination in this kind of scene is deeply dependent on the number of primitives present on the scene.

Therefore, the lighting calculation generates two problems: how to do it efficiently (in terms of performance) and how to do it correctly or at least perceptually correctly (in terms of the quality of the resulting image).

Figure 1: Rasterization vs Ray tracing. *Source:* Intel

Over the last years, mostly due to the previously mentioned problems and the existing hardware, the algorithms that have emerged have been focusing on solving only one of the problems. Thus we have algorithms that focus on fotorealism at the expense of performance, and other algorithms that focus on performance at the expense of realism (Figure 1).

To reach fotorealism, several algorithms have been proposed, such as recursive ray tracing (Whitted, 1980), bi-directional path-tracing (Lafortune and Willems, 1993), photon-mapping (Jarosz et al., 2008) or metropolis light transport (Veach and Guibas, 1997).

However, all these algorithms share a drawback: their performance. All these algorithms try to mimic the interactions of light rays between the objects in a scene, reflecting and refracting the photons according to the

characteristics of the materials of each object. This kind of simulation is very computationally expensive, existing however some implementations that can generate several frames per second and with a very good graphic result (e.g. Brigade 3).

To generate images in real-time, the most popular technique is rasterization. Rasterization is simply the process of mapping the triangles that compose the geometry of the objects in a scene to pixels. This process has been optimized over several years by the graphic card manufacturers to maximize the number of triangles that can be processed, but however, due to the nature of triangles and the rasterization process itself, the calculus of indirect illumination is very inefficient. Also, since these algorithms are deeply dependent on the number of primitives in the scene, it is necessary to simplify the geometry of the objects to be able to deal with scenes on a large scale (Figure 2).

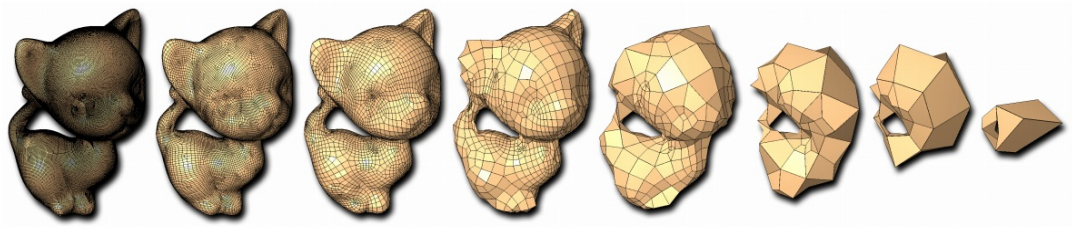


Figure 2: Geometry simplification. Information about the geometry is lost with an increasing level of filtering. *Source:* Daniels et al. (2008)

The problem is then to keep the necessary detail and at the same time maintain the rasterization at interactive frame rates and the memory consumption reasonable.

Since these previous approaches do not scale well with the required complexity level, the arising of new solutions is necessary.

Recently, new approaches have emerged that restrict the incoming light to the space visible by the camera, which permits to compute an approximation to the global illumination at interactive frame rates. It is possible to reach plausible results with these techniques but they still have some problems, mostly due to the restrictions imposed by the camera space. Since only the lights and objects visible by the camera are taken into account for the final illumination, this results in shadows and indirect light that appear and disappear depending on the movements of the camera and objects in the scene (Figure 3).

The name voxel comes from volumetric element and it represents the 3D generalization of a pixel. Voxels are usually arranged on an axis-aligned grid which structures and subdivides space regularly. Their main advantage is its own spatial representation and its regular structure, which

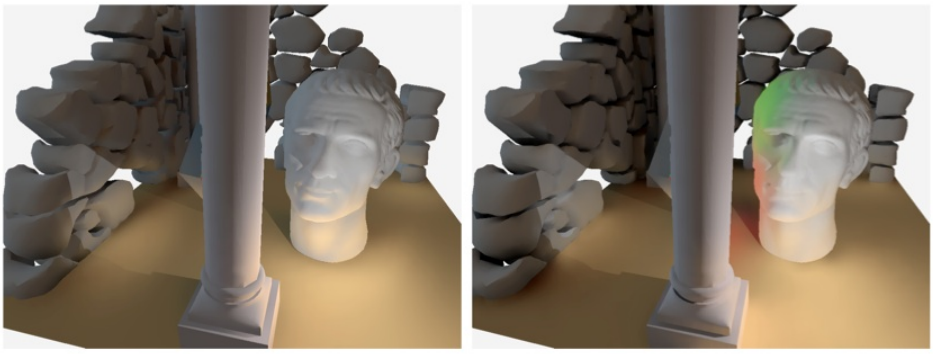


Figure 3: Indirect illumination on a scene with a hidden object behind the column. In the left image, only objects in camera space are taken into account and thus the hidden objects are disregarded since they are not visible by the current camera. *Source:* Thiedemann et al. (2011)

makes it easily manipulable. These features have turned voxel-based structures an excellent way of representing volumetric data.

Voxels have been used to represent several types of scientific data such as 3D scans or tomographic reconstruction of radiological data (Figure 4). They are also used in simulation processes such as fluid simulation based on Euler grids.

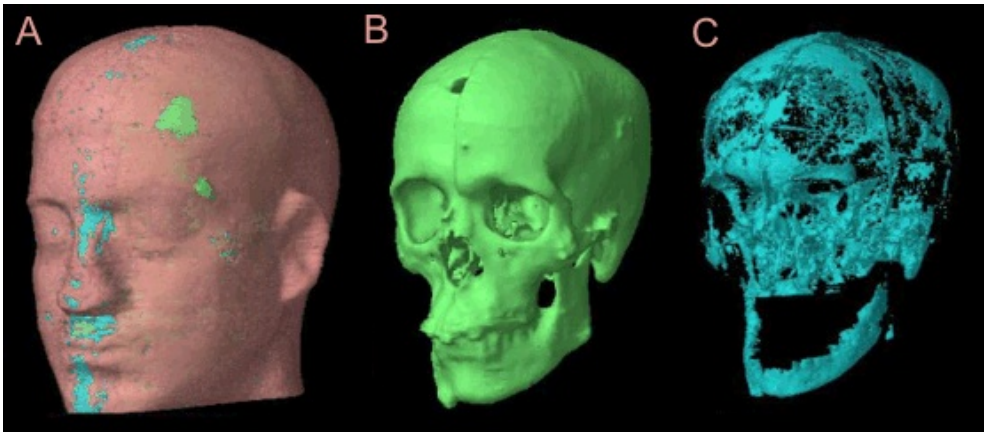


Figure 4: Voxels used to view medical data. *Source:* www.ustur.wsu.edu

More recently, new approaches have emerged that use a pre-filtering of the scene using voxels in order to simplify the scene and making it possible to approximately compute indirect illumination in real time. Since the whole scene is taken into account (or at least the volume that is voxelized), these algorithms are not view-dependent such as the screen-space approaches.

1.1 OBJECTIVES

What is proposed in this thesis is a study of algorithms for solving the indirect illumination problem in interactive frame-rates based on voxels to filter the scene. The proposed study is based on a review of the state of the art and search for existing algorithms for solving the problem, as well as their advantages and disadvantages.

This analysis seeks an evaluation in terms of performance of each step of the algorithms, as well as a qualitative comparison with rasterization and algorithms based on ray tracing.

An analysis on the introduction of new features available in the most recent versions of OpenGL is also intended. This features introduce new paradigms, which may imply a considerable redefintion of the initial algorithm.

1.2 DOCUMENT STRUCTURE

This document will be divided in 5 Chapters:

- Chapter 1 describes the motivation behind the choice of this theme and the Objectives (1.1) of this work.
- Chapter 2 provides some theoretical background as well as the description of some techniques that are used by the algorithms described in this thesis.
- In Chapter 3 the state-of-the-art of algorithms that calculate real-time indirect illumination using pre-filtered voxelized versions of the scene are presented.
- A detailed description of the development and analysis of the algorithms is made in Chapter 4, as well as a comparison of the several solutions obtained.
- Chapter 5 summarizes the work performed as well as the results obtained and proposes improvements for future work.

2

RELATED WORK

This chapter covers some of the building blocks of the methods that will be explored in chapter 3, namely voxelization, reflective shadow maps, volume ray casting, and cone tracing.

2.1 VOXELIZATION

Voxelization, or 3D scan conversion, is the process of mapping a 3D object built with polygons into a 3D axis aligned grid, obtaining a volumetric representation of the object made of voxels. The term 'voxelization' was first referenced on the paper "3D scan-conversion algorithms for voxel-based graphics" (Kaufman and Shimony, 1987). Since then, multiple approaches have been proposed to convert the surface of a triangle-based model into a voxel-based representation stored as a voxel grid (Eisemann and Décoret, 2008; Zhang et al., 2007; Dong et al., 2004). These can be classified in two categories: surface voxelization algorithms and solid voxelization algorithms.

Regarding surface voxelization, only voxels that intersect the models triangles are stored, thus creating a representation of the surface of the object. Solid voxelization demands a closed object since it also sets the voxels that are considered interior to the object (using a scanline fill algorithm for example). Since the voxelization process is introduced in this thesis in the context of global illumination algorithms, surface voxelization is preferred since light is reflected at the surface of the materials.

Other interesting variants of voxelization algorithms are called binary voxelization and conservative voxelization. These refer to the way voxel data is encoded in the resulting voxel grid. The voxelization process can store multiple values on the voxel grid (or grids), such as color and normal values of the voxelized model, or simply store an occupancy value for binary voxelization.

GPU voxelization algorithms use the rasterization pipeline to convert the triangles to voxels. During rasterization, each triangle generates multiple 2D fragments corresponding to the pixels intersected by the triangle. Then each pixel will affect one voxel.

As is, with this approach only the coverage of the pixel center is tested during the rasterization process. Therefore, some voxels that do intersect the surface are not set, resulting in a voxelization with holes (Figure 5).

In order to ensure a full voxelization, where every voxel that intersects the surface is set, every pixel touched by a triangle must generate a 2D fragment. One approach that can approximate such a solution is to use multisampling, generating a fragment when any of the sample locations in a pixel is covered by a triangle. However, this method still does not guarantee a full voxelization, since multisampling does not guarantee full coverage.

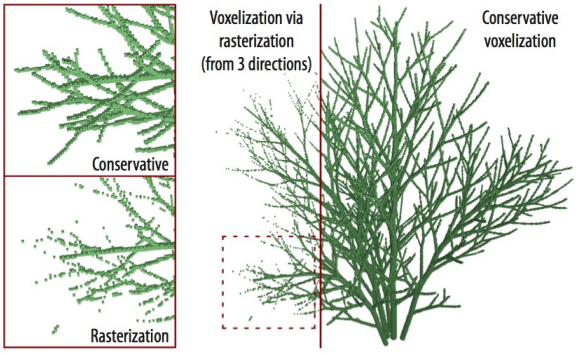


Figure 5: Conservative Voxelization. *Source:* Schwarz and Seidel (2010)

An accurate way to ensure full voxelization is to use a technique known as conservative rasterization. This process is based on the work of Hasselgren et al. (2005). The procedure is to shift the edges of the triangle outwards such that the triangle covers the center of every pixel that it touches. In order to avoid over rasterization when dealing with thin triangles a bounding box is also computed to discard pixels not touched by the original triangle (Figure 6).

NVIDIA has proposed an OpenGL extension to add conservative mode rasterization (NVIDIA, 2014);

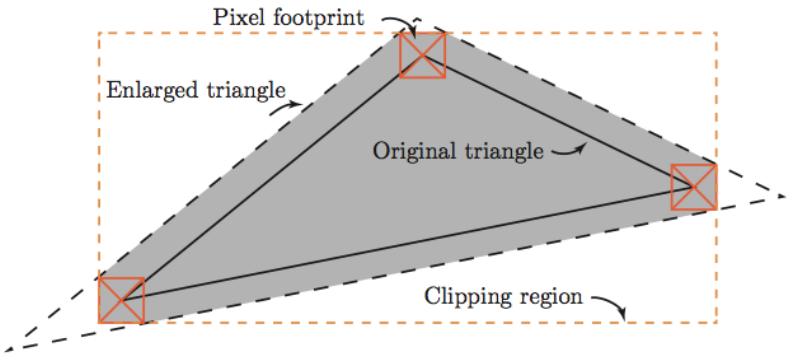


Figure 6: Triangle Expansion in Conservative Rasterization. *Source:* Crassin and Green (2012)

Since the main objective is to compute indirect illumination in real-time from a voxelized representation of the scene, achieving very fast voxelization of a triangle-based representation is critical under some circumstances, as shall be seen.

Multiple approaches of surface voxelization are presented next. Voxelization based in volume slicing and depth peeling are depth based approaches (2.1.1). On the other hand, thanks to the geometry shader, voxelization can be achieved with a single GPU pass. (2.1.2).

2.1.1 *Depth based Voxelization*

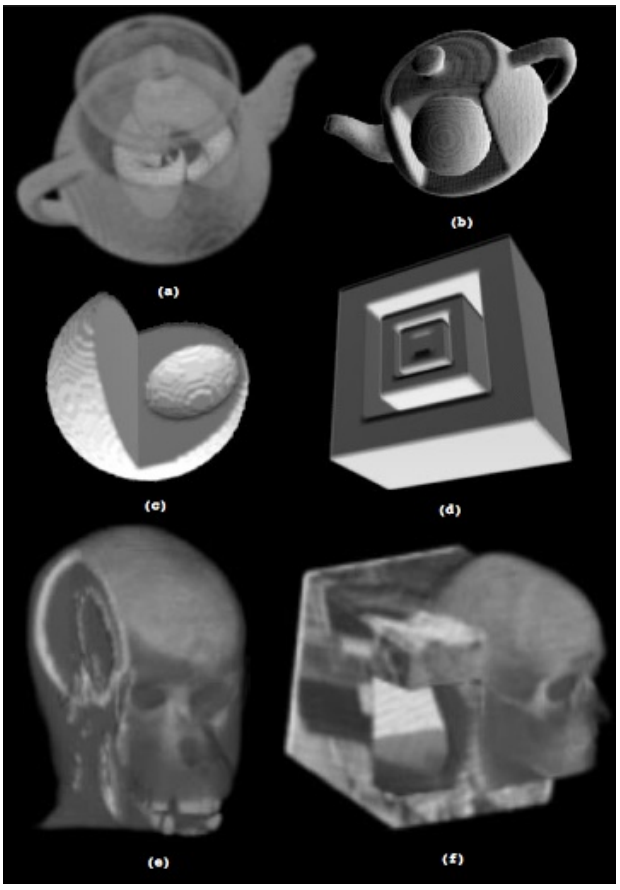


Figure 7: Examples of voxelization. *Source:* Fang et al. (2000)

Two of the earliest voxelization algorithms are based in volume slicing and depth peeling (Figure 7).

The idea in volume slicing (Fang et al., 2000) is to generate multiple slices for the voxelized volume. First, a bounding volume is chosen that defines the space to be voxelized. Then, the

algorithm performs multiple passes, setting the near and far planes such that for each rendering pass only the geometry contained between the two clip planes is rendered (Figure 8). This generates a number of 2D textures, each of them representing the geometry in the sliced volume at a certain depth. In this method, the number of passes performed is the same as the number of slices in which the volume it to be subdivided.

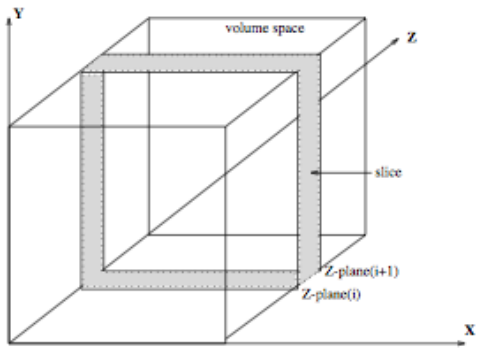


Figure 8: Volume Slicing. *Source:* Fang et al. (2000)

Depth peeling voxelization (Li et al., 2005) retrieves each depth layer of the scene in successive rendering passes (Figure 9). The scene is rendered normally, using the depth test to retrieve the nearest fragments. Each following pass renders the scene and compares the depth of the current fragment with a depth texture retrieved from the depth buffer of the previous pass. If the depth is greater than the depth in the depth texture, the fragment belongs to a new layer. The process stops when no fragment has a depth higher than the corresponding pixel in the depth texture.

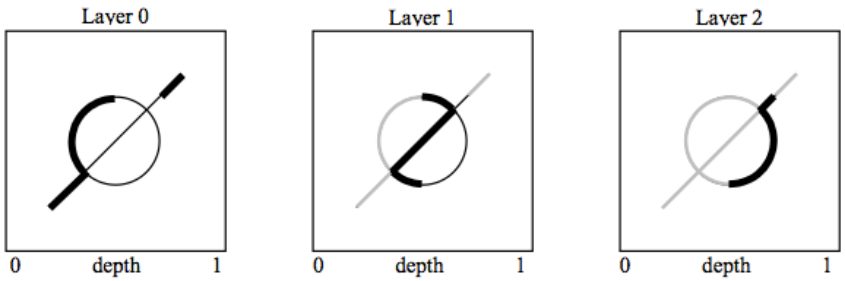


Figure 9: Depth peeling. Closest surfaces to the viewer are represented with bold black lines, hidden surfaces with thin black lines and peeled surfaces with light grey lines. *Source:* Everitt (2001)

With this approach, each 2D texture represents a different layer of the scene, and each pixel in those textures corresponds to a voxel. In order to avoid missing voxels, the peeling is performed

from three different orthogonal views, which results in voxels being rendered more than once (Figure 10). However, this approach usually requires much less rendering passes than the volume slicing voxelization method (Li et al., 2005).

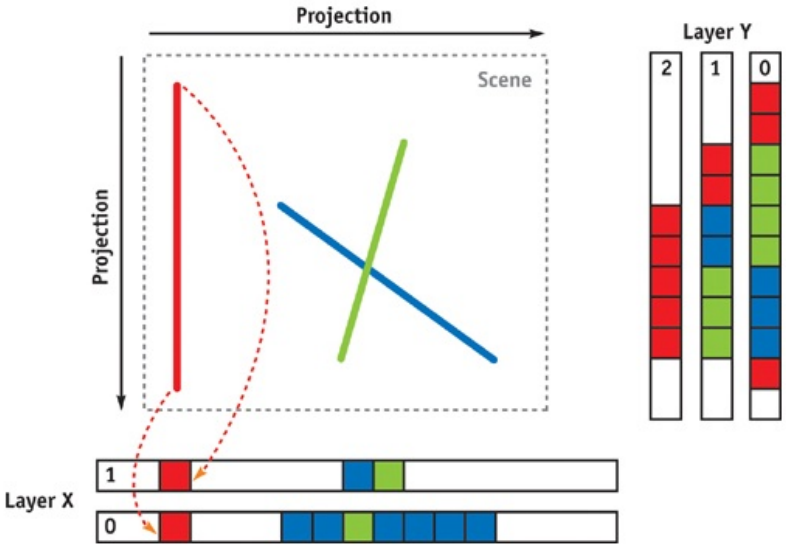


Figure 10: Projection along different orthogonal views in order to avoid missing voxels. *Source:* Li et al. (2005)

2.1.2 *Single Pass GPU Voxelization*

This surface voxelization algorithm is based on the observation made by Schwarz and Seidel (2010) that a thin surface voxelization of a triangle can be computed by testing if the triangle's plane intersects the voxel, and the 2D projection of the triangle along the dominant axis of its normal intersects the 2D projection of the voxel.

The dominant axis is the one from the three main axes of the scene that maximizes the surface of the projected triangle (Figure 11).

One way to ensure that a triangle is projected along the dominant axis of its normal is to render the object from the three main axis of the scene. However, this approach requires the scene to be rendered three times and for complex scenes this might be too computationally expensive.

Another approach is to use a geometry shader to determine the dominant axis and use the appropriate view matrix to transform the vertices to clip space, allowing the voxelization process to be executed in a single pass.

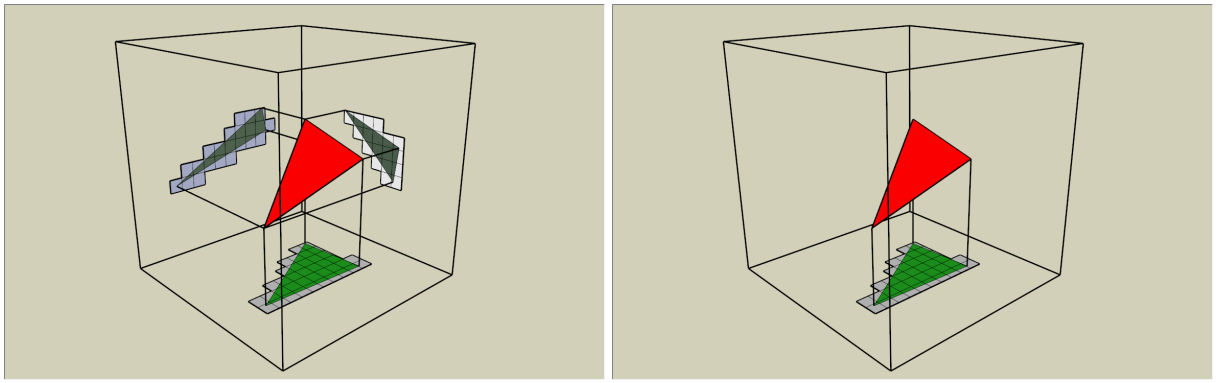


Figure 11: Projection of a triangle through the three main axis of the scene (left). The Y axis is chosen for the voxelization since it is the one that will generate maximum number of fragments during rasterization (right). *Source:* NVIDIA (2015)

In this manner, the voxelization process can be divided in several steps (Figure 12). First, the dominant axis of the triangle normal must be determined. This axis is one of the three main axes of the scene that maximizes the projected surface of the triangle, thus generating a larger quantity of fragments during rasterization. Since this choice needs to be done for every triangle, the geometry shader is used for this purpose, because the information about the three vertices of the triangle is available. The selected axis is the one that provides the maximum value for $\mathbf{l}_{\{x,y,z\}} = |\mathbf{n} \cdot \mathbf{v}_{\{x,y,z\}}|$ with $\mathbf{n}$ being the triangle normal and $\mathbf{v}_{\{x,y,z\}}$ the three main axis of the scene.

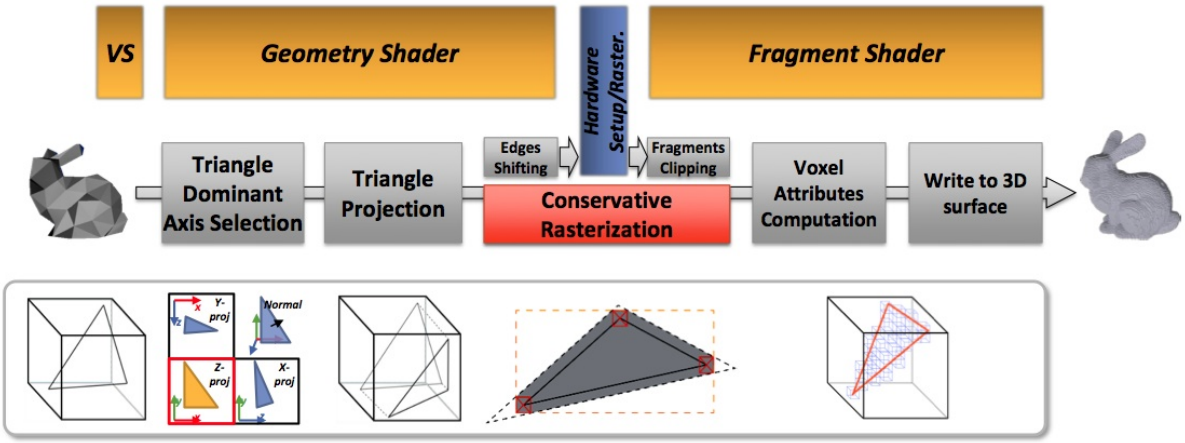


Figure 12: Voxelization Pipeline. *Source:* Crassin and Green (2012)

Once the dominant axis of the triangle normal has been selected, the triangle must be projected along this axis. The vertices of the triangle are swizzled in order to match the projection in

the geometry shader. The projection is an orthographic projection, setting its limits so that the frustum covers the entire volume to be voxelized. Rendering is performed with both backface culling and the depth test disabled in order to prevent early-z rejection and to render both sides of the triangle.

If a complete voxelization is required, the geometry shader can also be used to expand the triangles with conservative rasterization.

After passing through the geometry shader, the triangle is fed into the rasterization pipeline to perform 2D scan conversion (rasterization).

Since we want to generate fragments corresponding to the resolution of the voxel grid, the viewport resolution is set to match the lateral resolution of the voxel grid (e.g. 512×512 for a 512^3 grid). Also, all framebuffer operations can be disabled since image load/store is used to write the voxel data.

In the fragment shader voxel attributes are computed for each fragment. The 3D position inside the voxel grid must be determined in order to store these attributes in the correct voxel. These attributes are any useful attribute we would want to store per voxel. These can be simply storing a single value for the opacity of the voxel, or may include shading parameters such as albedo and normals.

The attributes can be stored directly in the voxel grid with image load/store operations. Since a voxel may contain many pixels, average values are commonly considered.

Another approach used by Crassin and Green (2012) is to store the fragments in a buffer generating a voxel fragment list. This voxel fragment list is a linear vector of entries stored inside a preallocated buffer object, which is treated as an array of structures, where each structure contains the attributes for a fragment voxel. To manage this list, a counter of the number of fragments of the list is maintained as a single value stored inside an atomic buffer object and updated with an atomic add operation in a linear list.

Using a voxel fragment list provides some advantages over merging the fragments directly since it allows to decouple the information of the scene geometry obtained from the voxelization pass from the lighting information. This is useful, especially for the creation of the sparse voxel octree (3.1.1). This comes however at the expense of some extra memory required to store every voxel fragment generated during the voxelization process.

2.2 REFLECTIVE SHADOW MAPS

Shadow Mapping, originally published in 1978 (Williams, 1978), is a very efficient method to add shadows to a 3D scene.

The basic shadow mapping algorithm consists in two separate passes. First, the scene is rendered from the light point of view where a texture storing the depth of the objects in the scene is created, i.e. the distance of the closest triangle to the light.

In a second step it is possible to test if a fragment is visible from the light. First it is necessary to compute the fragment coordinates in light coordinates. The coordinates are used to retrieve the depth value from the texture create in the first step, and this value is compared with the actual distance from the fragment to the light. Fragments which are further away than the recorded depth are in shadow, whereas fragments with the distance equal to the depth are lit.

Different techniques are employed for different types of lights: 2D textures are used for spotlights and cube maps for point lights.

Reflective shadow mapping (RSM - Dachsbacher and Stamminger (2005)) extends the shadow map concept by storing more information for each texel in the depth map. The information stored includes normals, world-space position and reflected flux. This information is stored in multiple textures using multiple render targets.

The information in this extended shadow map can be seen as a set of virtual point lights providing sources of indirect light. When computing the colour of a pixel a subset of these virtual point lights can be used to compute its first bounce of indirect light.

2.3 VOLUME RAY CASTING

In nature, a light source emits light rays that travel through space until they hit the surface of an object. When a photon hits a surface it can be absorbed, reflected or refracted, depending on the properties of the material.

Ray tracing is a technique that tries to mimic what happens in nature. However, instead of shooting rays from the light until they hit the view plane, which would need an enormous number of rays in order to produce a satisfying result, rays start from the view plane and are launched into the scene.

Ray tracing techniques have also been applied to the rendering of 3D volumetric data sets. One of the most commonly used volume rendering techniques is called Volume Ray Casting, or Ray Marching (Levoy, 1990).

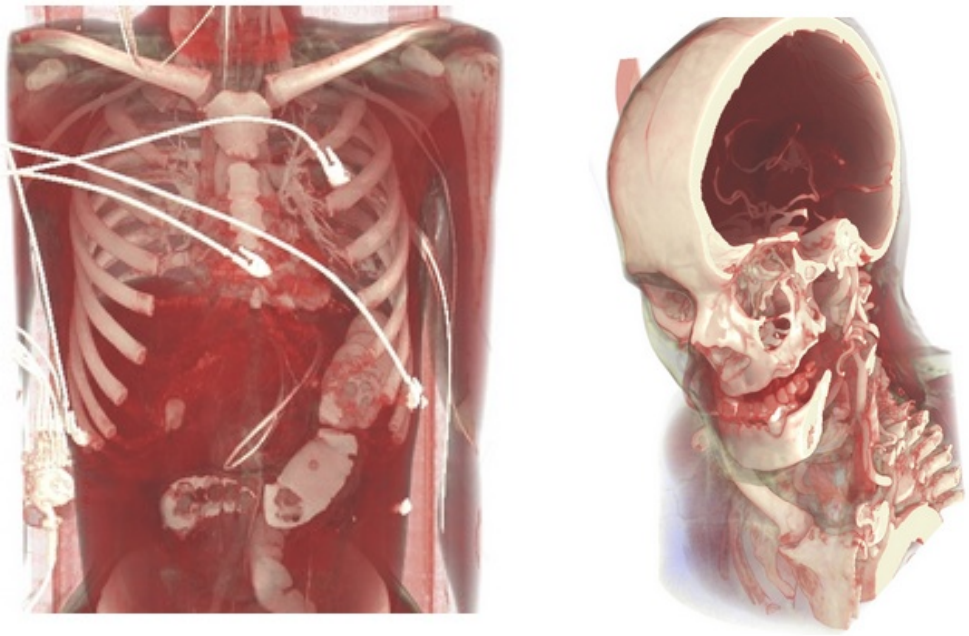


Figure 13: Volume Rendering. *Source:* Microsoft Volume Rendering SDK

This algorithm allows the production of a 2D image from a 3D grid made of voxels, in which each voxel contains an opacity and color value (Figure 13).

The idea is to launch rays from the view plane into the volume, sampling it at equally spaced intervals (Figure 14). The data is interpolated at each sampling point since the volume is usually not aligned with the camera (usually using trilinear interpolation) and the interpolated scalar values are mapped to optical properties by using a transfer function, forming an RGBA color value. A transfer function can be simply described as setting opacities for each voxel in such a way that both emission and absorption for a voxel can be computed.

The sampled values are accumulated along the ray, in a front-to-back compositing scheme since it allows to stop the evaluation when the accumulated transparency reaches zero:

$$C_{dst} \leftarrow C_{dst} + (1 - \alpha_{dst})C_{src}$$
$$\alpha_{dst} \leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src}$$

with C the color (C_{dst} for accumulated color and C_{src} for sampled color), α the opacity (α_{dst} for accumulated opacity and α_{src} for sampled opacity) defined as $\alpha = 1 - T$ and T the transparency.

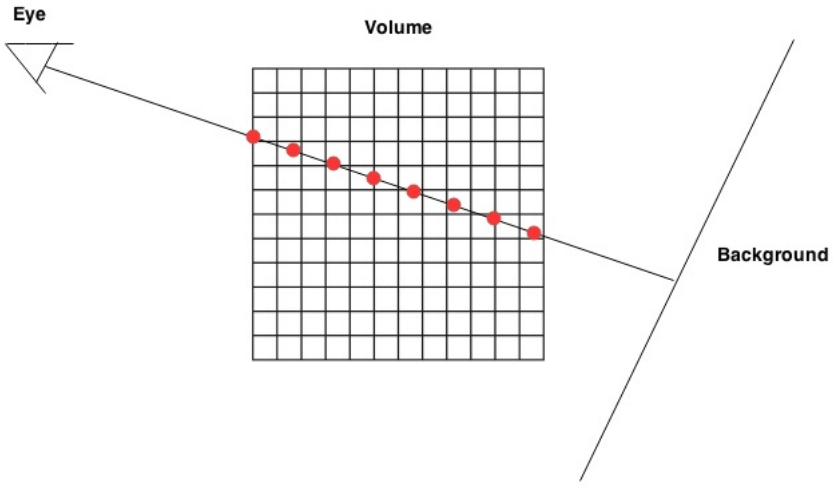


Figure 14: Volume Ray Casting

2.4 CONE TRACING

Apart from the rendering time, ray tracing approaches also suffer from problems related to aliasing and sampling. The problem is that shooting only one ray per pixel on the screen fails to capture enough information in order to produce an anti-aliased output. One common solution to this problem is using multisampling and sample each pixel multiple times with different offsets, instead of always shooting rays through the center of the pixels, at the expense of performance.

Cone Tracing (Amanatides, 1984) was proposed as a solution that allowed to perform anti-aliasing with only one ray per pixel. The main idea is to shoot cones instead of rays, by attaching the angle of spread and virtual origin of the ray to its previous definition, which only included its origin and direction. The angle of spread is defined as the angle between the center line of the cone and the cone boundary as measured at the apex of the cone, and the virtual origin is the distance from the apex of the cone to the origin.

The pixels on the screen are viewed as an area of the screen instead of a point, and setting the angle of spread of the cone such that it covers the entire pixel on the view plane will guarantee that no information is lost during the intersection process, producing an anti-aliased image. However, calculating the intersections between cones and objects is complex. The intersection test must not return only information about whether the cone has intersected any object, but also the fraction of the cone that is blocked by the object.

More recently, volume ray casting was used together with cone tracing in order to speed-up the cone intersection test (Crassin, 2011). The general idea is to launch cones instead of rays through a volume composed of voxels and accumulate the samples until the opacity saturates.

This technique is very flexible, allowing to generate multiple kinds of effects. It has been used in Crassin (2011) in order to perform direct volume rendering of voxel data, as well as to approximate global illumination, soft shadows and depth of field effects very efficiently.

By shooting a cone instead of a single ray towards the light source and accumulating the opacity along the cone it is possible to estimate how much of the light is occluded by objects. The cone starting from the camera intersects the object, generating an impact volume. A cone is then launched from the object to the light source with its radius equal to the impact volume. The traversal stops when the opacity value saturates, meaning that the object lies in shadow (Figure 15).

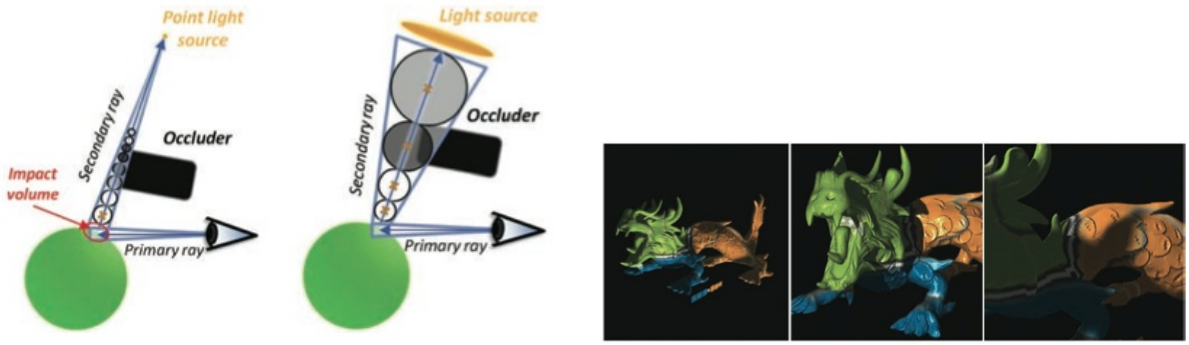


Figure 15: Estimating Soft Shadows through Voxel Cone Tracing. *Source:* Crassin (2011)

To approximate the depth of field blur effect, a similar technique is employed. The lens cone radius (the radius of the cones launched from the camera) is modified depending on the aperture of the lens and the focus plane and a double cone is used (Figure 16).

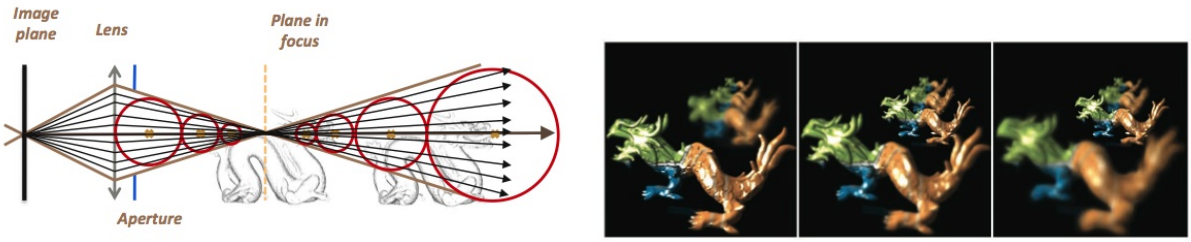


Figure 16: Estimating Depth of Field Effects through Voxel Cone Tracing. *Source:* Crassin (2011)

In section 4.2 an efficient implementation of this technique using a mipmapped voxel grid is employed in order to approximate global illumination.

2.5 VOLUMETRIC LIGHTS

Figure 17: The volumetric lighting effect is computed (left) and then added to the scene (right). *Source:* Vos (2014)

Voxels have also been applied to the creation of volumetric light effects (Figure 17). Vos (2014) proposes using a voxel grid to store information about the amount of scattering present in a volume in space in the scene, allowing to control the amount of scattering in a dynamic and more precise manner.

The idea is using particle effects to insert amounts of scattering in a 3D texture, which will be used later in a ray marching pass (2.3), together with the Henvey-Greenstein phase-function (Henyey and Greenstein, 1941), in order to compute the amount of light scattered in the direction of the camera (Figure 18). The Henvey-Greenstein phase-function (Equation 1) is an approximation to the Mie-scattering phase-funtion (Mie, 1908), which describes the way light is scattered by aerosols such as dust, mist or smoke.

$$f_{HG}(\theta) = \frac{(1-g)^2}{4\pi \cdot (1+g^2-2g \cdot cos(\theta))^{\frac{3}{2}}} \tag{1}$$

The algorithm starts by generating particles in order to simulate the effect pretended (for example, ascending particles for a smoke effect) and use their position to insert an amount of scattering into their corresponding position in the voxel grid.

A fullscreen quad is then rendered using deferred rendering and a ray is launched in the direction defined by the line segment from the camera to the fragment's position.

Ray marching is employed to advance through the ray and the voxel grid is sampled in order to recover the scattering amount stored in the previous pass. The sampling point is projected into a shadow map in order to find if it receives light and if it does, the Henvey-Greenstein

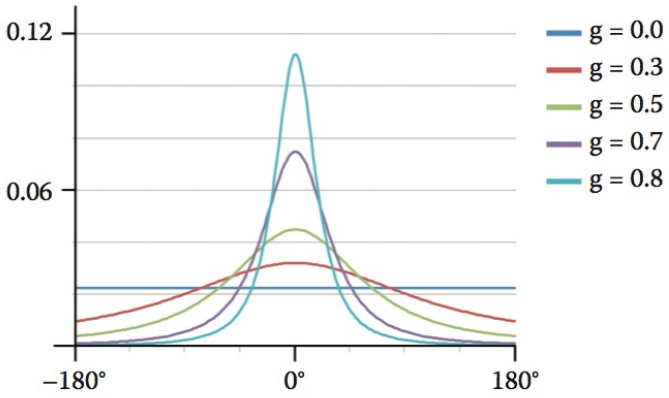


Figure 18: Results for the Henvey-Greenstein phase-funtion for different values of g. *Source:* Vos (2014)

phase-function is evaluated in order to calculate the amount of light reflected in the direction of the camera. Finally, all samples are combined and the effect produced is added to the scene (Figure 17).

Using ray-marching to compute the volumetric light effect can be very computationally expensive, since a large number of samples are needed in order to obtain a visually correct result Vos (2014).

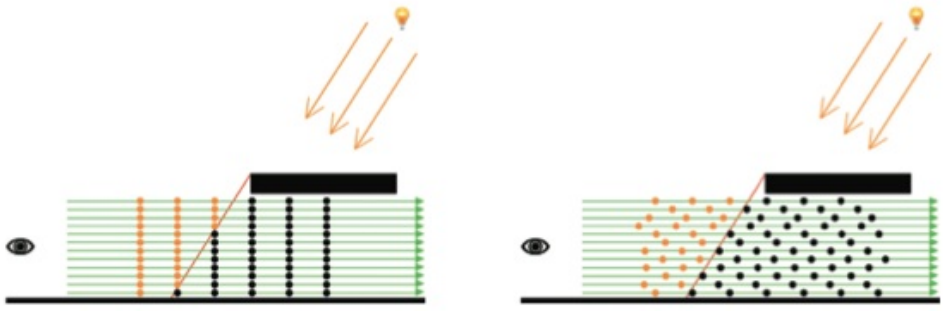


Figure 19: Left: Ray marching. Right: Dithered ray marching. *Source:* Vos (2014)

In order to reduce the number of steps performed by ray-marching, dithered ray-marching can be employed instead (Figure 19). The general idea is to use a dither pattern based on the Bayer matrix to advance the starting point of the ray by an offset. This causes the final image to contain a lot of noise (Figure 20), which can be solved by using a bilateral Gaussian blur (Pham and Vliet, 2005).

Another way to improve the performance of the algorithm is to render the volumetric effect at half the resolution of the window. However, since the result has to be added to the scene lately,

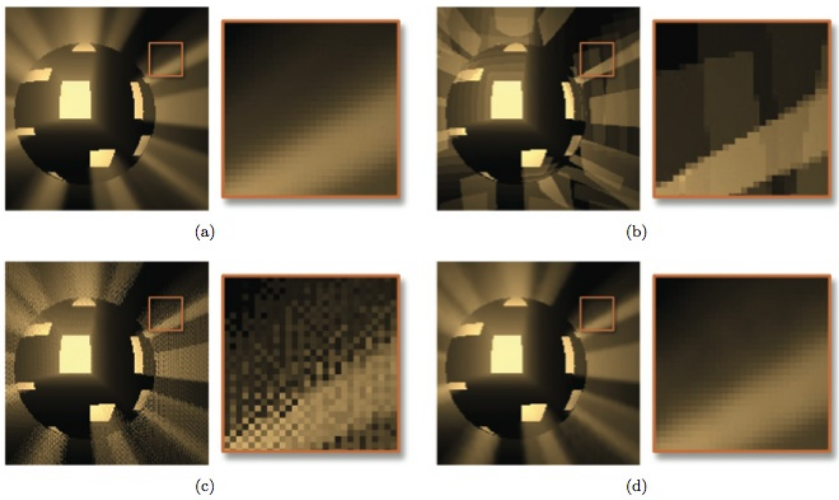


Figure 20: (a) ray-marching with lots of ray marching steps. (b) Reducing the number of ray marching steps causes banding artifacts. (c) Result from applying dithering to the ray marching samples. (d) Result from applying a bilateral Gaussian blur to the dithered result. *Source:* Vos (2014)

a bilateral upsampling (Shopf, 2009) is needed in order to avoid turning the edges pixelated or blurry if a regular bilinear filter is applied instead (Figure 21).

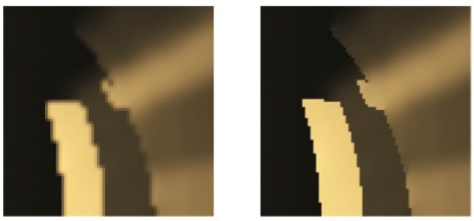


Figure 21: Using a bilinear filter causes blurry edges (Left). Using a bilateral filter preserves the edges (Right). *Source:* Vos (2014)

3

REAL-TIME VOXEL-BASED GLOBAL ILLUMINATION ALGORITHMS

Figure 22: Voxel-based Global Illumination. *Source:* Crassin et al. (2011)

Over the past few years, there has been an increasing interest in algorithms based on ray-tracing. With the rapidly increasing processing power of the graphics cards, these algorithms that required a long time to generate an image have started to be able to generate a few frames per second. But tracing polygons (in the classical sense, in which rays are intersected with triangles) is too expensive for real time applications.

Voxels have many benefits when compared to triangles, such as their ability to easily handle transparency, reflections and refraction by using volume ray casting (2.3) thanks to their volumetric representation. They are also cheaper to intersect than triangles, which makes them a good choice for ray tracing.

Voxels have been used for diverse applications, such as fluid simulation (Crane et al., 2007) and collision detection (Allard et al., 2010), but recently new algorithms for computing approximate global illumination in real time have been introduced (Figure 22).

These algorithms are very similar in their structure, as will be shown. These algorithms start by voxelizing the scene, storing voxel data into some data structure and then use this structure to

compute an approximation of the light interactions between the objects in the scene by utilizing a ray-tracing based approach.

There are several algorithms and data structures to perform each of these steps, each of them with advantages and disadvantages. This dissertation will focus on the most recent algorithms that have been presented for computing an approximation to global illumination in real time.

Each of the following sections will cover specific algorithms, providing an overview of the state of the art in this field. The chapter ends with a brief comparison and summary section.

3.1 INTERACTIVE INDIRECT ILLUMINATION USING VOXEL CONE TRACING

In order to maintain the performance, data storage, and rendering quality scalable with the complexity of the scene geometry, some way to pre-filter the appearance of the objects on the scene is needed. Pre-filtering not just the textures but the geometry as well will provide a scalable solution to compute approximate global illumination, only dependent on the rendering resolution, scaling with very complex scenes (Crassin et al., 2011).

Let us consider a volume in space containing multiple surfaces distributed more or less randomly. The overall light interactions inside the volume can be estimated by ignoring the exact positions of these surfaces inside the volume, and using instead an overall density distribution and an overall reflectance function to approximate the interaction of light within this volume (Figure 23). This observation was made by Perlin (Perlin and Hoffert, 1989) and Kajiya and Kay (Kajiya and Kay, 1989), allowing to pre-filter the information on the scene geometry into a volumetric representation.

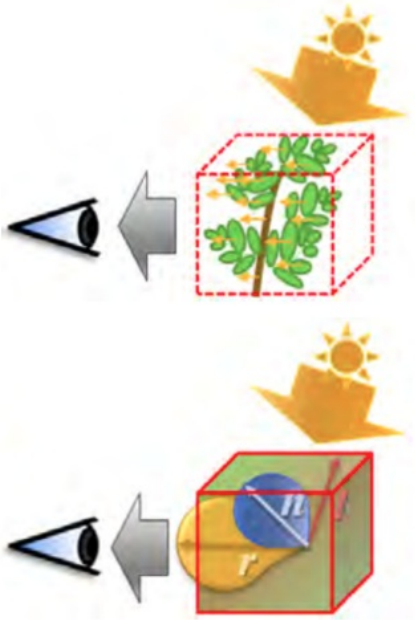


Figure 23: Voxel Lighting. *Source:* Crassin (2011)

Thus, when the geometry is pre-filtered in this way, the parameters used to compute global illumination can be represented volumetrically for a volume containing those surfaces, instead of using a simplified surface. With this kind of volumetric representation, the geometry of the scene can be represented by a density distribution associated with the parameters of the shading model describing the way light is reflected inside a volume (2.3).

One of the main advantages of transforming geometry in density distributions is that filtering this kind of distribution is turned into a linear operation (Neyret, 1998). This linear filtering is important since it allows to obtain a multiresolution representation of the voxel grid based on mipmapping, making it possible to automatically control the level of detail by sampling different mipmap levels of the voxel grid.

The general idea of this technique is to pre-filter the scene using a voxel representation (2.1) and store the values in an octree structure in order to get a hierarchical representation of the scene (3.1.1). The leaves of the octree will contain the data at maximum resolution and all the upper levels of the octree will mipmap the lower levels to generate data at different resolutions (3.1.2), thereby obtaining the basis for controlling the level of detail based on the distance from the camera.

After pre-filtering the scene, it is possible to compute an approximation to indirect illumination using Voxel Cone Tracing (Figure 24).

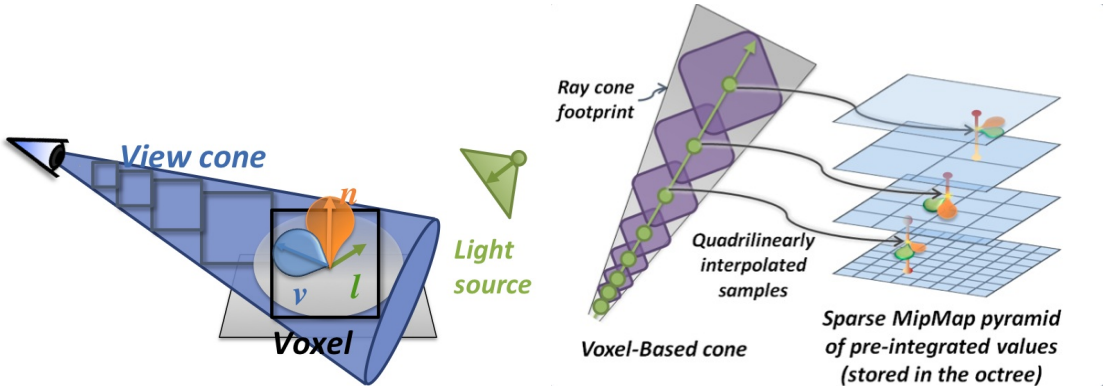


Figure 24: Voxel Cone Tracing. *Source:* Crassin et al. (2011); Crassin (2011)

However, this approach also has its problems. Besides the need for certain hardware features only available on the latest generation graphics cards, it is not practical for scenes with a large number of moving objects (Crassin, 2011). Updating the octree is a costly operation, so the sparse voxel octree cannot be rebuilt in every frame. Static objects only need to be pre-filtered once while dynamic objects need to be filtered in every frame. For a few moving objects it is possible to update the octree an keep the algorithm rendering at interactive frame rates, however increasing the number of dynamic objects will eventually turn this operation too computationally expensive, ruining the performance of the algorithm.

3.1.1 *Sparse Voxel Octree*

In order to build the sparse voxel octree, the scene must first be voxelized. The voxelization process is as detailed in Section 2.1.2. It uses conservative rasterization to generate fragments for all the voxels touched by the triangles in the scene and stores them into a voxel fragment list.

The voxel fragment list stores world positions, normals and material colors into a preallocated linear buffer in GPU memory, which will be needed for the construction of the sparse voxel octree data structure.

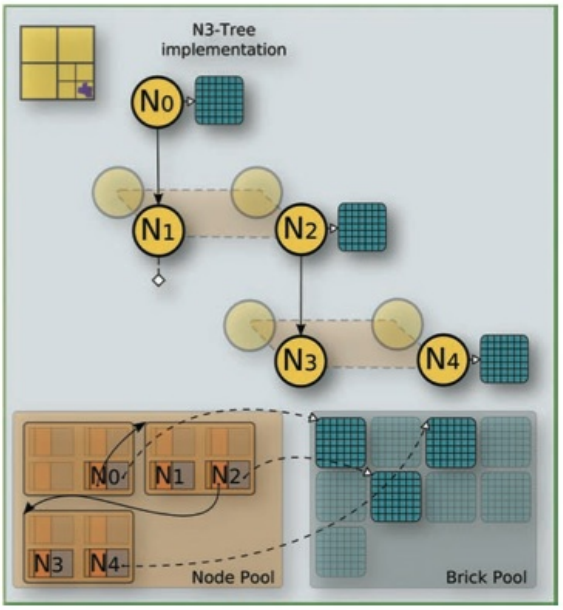


Figure 25: Sparse Voxel Octree Structure. *Source:* Crassin et al. (2010)

If the voxel fragments generated in the voxelization pass were stored in a regular 3D texture, every voxel would be stored, not just the ones intersected by the mesh triangles, thus producing a full grid and wasting a lot of memory with empty voxels. In order to handle large and complex scenes, there is a need to use an efficient data structure to handle the voxels.

In this method, the data structure chosen for this purpose is a Sparse Voxel Octree (Crassin et al., 2009; Laine and Karras, 2010), which has several benefits in this context, such as storing only the voxels that are intersected by mesh triangles and providing a hierarchical representation of the scene, which is very useful for the Level Of Detail (LOD) control mechanism.

The sparse voxel octree is a very compact pointer-based structure (Figure 25). The root node of the tree represents the entire scene and each of its children represents an eight of its volume.

Octree nodes are organized as $2 \times 2 \times 2$ node tiles stored on linear video memory.

In order to efficiently distribute the direct illumination over all levels of the octree afterwards, the structure also has neighbor pointers, allowing to rapidly visit neighboring nodes and the parent node.

Since the nodes are encoded in $2 \times 2 \times 2$ node tiles, some information needs to be duplicated in the borders of neighboring bricks to allow the use of hardware trilinear sampling in the brick boundaries.

If node centered voxels are used, a one voxel border needs to be added to the bricks. This would waste too much memory and introduce a lot of redundancy in the stored data (specially when small bricks are used, such as here).

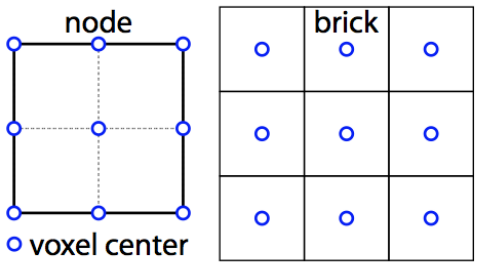


Figure 26: Voxel Brick. *Source:* Crassin et al. (2011)

Instead, voxel attributes are associated with the node tiles, stored as $3 \times 3 \times 3$ bricks in texture memory and assume that the voxel centers are located at the node corners instead of the node centers (Figure 26). This method allows to have all necessary data for a correct interpolation in the octree nodes without needing to store a one voxel border for neighboring voxels.

The sparse voxel octree is built from top to bottom by starting from the root node and subdividing non-empty nodes until the leaves are reached (Figure 27). After its creation, voxel fragments are written in the leaves and mipmapped into the interior nodes of the tree (3.1.2).

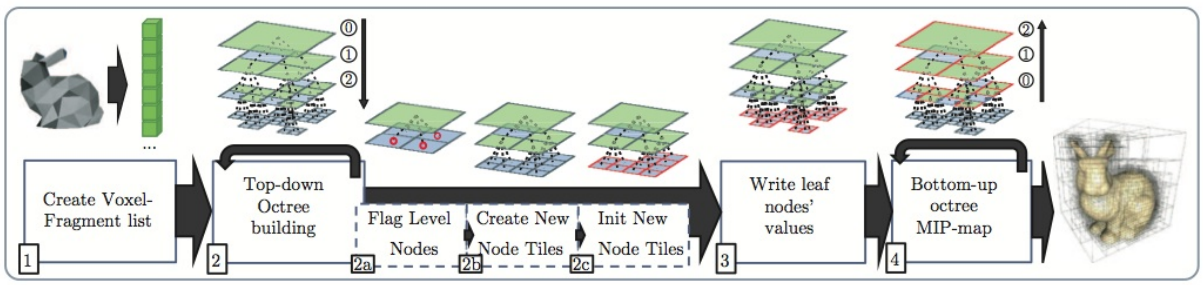


Figure 27: Steps for the creation of the sparse voxel octree structure. *Source:* Crassin and Green (2012)

If the voxel fragments were used directly at the end of the voxelization pass to subdivide the octree, traversing and subdividing the sparse voxel octree until the leaf nodes were reached, some synchronization between threads would have to be taken care of, since multiple threads could try to subdivide the same node at the same time (especially in the first levels of the octree).

A possible solution to avoid synchronization problems is to create the octree structure in multiple passes, one for each level of the octree, and voxelize the scene with a resolution matching the current octree level.

However, since the voxel fragments are written into a voxel fragment list at the end of the voxelization pass, it is possible to subdivide the building process of the octree structure in multiple passes, in such a way that no synchronization between threads is needed unlike if it was done in a single pass. Also, the scene does not need to be voxelized at multiple resolutions since the data stored in the voxel fragment list can be reused to create each level of the sparse voxel octree.

The subdivision of the octree is done one level at a time and the subdivision of each level is performed in three steps (Figure 28). First, the nodes that need to be subdivided are flagged using one thread per entry on the voxel fragment list. Each thread traverses the octree from top to bottom until it reaches the current level and flags the node in which the thread ended.

When a node is flagged, a new tile with $2 \times 2 \times 2$ subnodes needs to be allocated and linked to the node. In order to do so, one thread is launched per node on the current level of the octree and each of them checks the flag of its node, allocating a new tile and assigning its index to the childnode pointer of the current node if needed. The new nodes are also initialized to null child pointers. Since allocations can occur at the same time, they are controlled using a shared atomic counter.

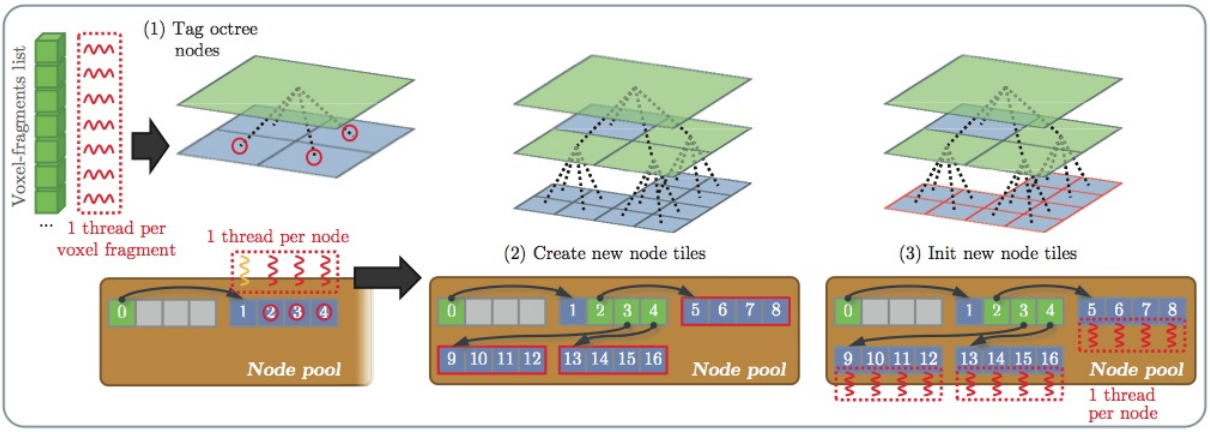


Figure 28: Node Subdivision and Creation. *Source:* Crassin and Green (2012)

Once the octree is built, the leaves of the octree need to be filled with the voxel fragments and this is achieved by using one thread per entry on the voxel fragment list. All values falling in the same destination voxel will be averaged and since multiple voxel fragments may try to write their attributes in the same destination, atomic operations are needed in order to avoid incorrect results. To do so, all values are first added using an atomic add operation, updating at the same

time a counter so that the summed value can be divided by the counter value in a subsequent pass.

After the sparse voxel octree has its leaves filled with the voxel fragments, these values need to be filtered (mipmapped) into the interior nodes of the sparse voxel octree (3.1.2).

Dynamic and static objects are both stored in the same sparse voxel octree structure for an easy traversal and unified filtering. Since fully dynamic objects need to be revoxelized every frame and static or semi-static objects only need to be revoxelized when needed, a time-stamp mechanism can be used in order to differentiate each type of object and prevent overwriting of static nodes and bricks (Crassin et al., 2011).

3.1.2 *Mipmapping*

In order to generate an hierarchical representation of the voxel grid, the leaves of the sparse voxel octree are mipmapped into the upper levels. The interior nodes of the sparse voxel octree structure are filled from bottom to top, in **n-1** steps for an octree with **n** levels. At each step, one thread is used to average the values contained in the eight subnodes of each non empty node in the current level.

Since each node contains a 3^3 vertex centered voxel brick, its boundary reappears in neighboring bricks. Consequently, when mipmapping the values, each voxel has to be weighted by the inverse of its multiplicity.

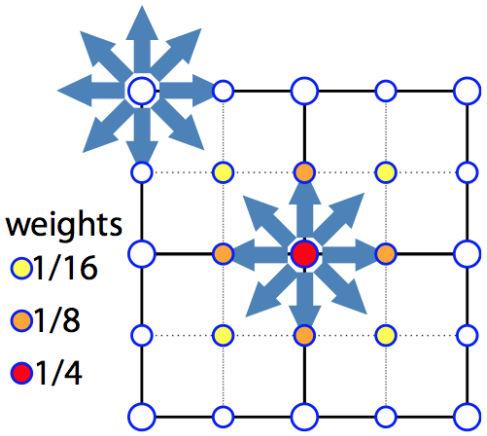


Figure 29: Mipmapping Weighting Kernel. *Source:* Crassin et al. (2011)

This results on a 3^3 Gaussian weighting kernel (Figure 29), which is an optimal reconstruction filter in this case (Crassin et al., 2011).

Each voxel at a given level has to represent the light behavior of the lower levels (and the volume it represents).

For this purpose, normals and light directions are encoded with distributions, since these are more accurate than single values (Han et al., 2007). However, to reduce the memory footprint, these distributions are not stored using spherical harmonics. Instead, Gaussian lobes characterized by an average vector D and a standard deviation σ are used. To ease the interpolation, the variance is encoded using the norm $|D|$ such that $\sigma^2 = \frac{1-|D|}{|D|}$ (Toksvig, 2005). For example, the

Normal Distribution Function (NDF) can be computed from the length of the averaged normal vector $|N|$ stored in the voxels and $\sigma_n^2 = \frac{1-|N|}{|N|}$.

The Normal Distribution Function describes the normals within a region, defined on the unit sphere (Figure 30). The NDF and the Bidirectional Reflectance Distribution Function (BRDF) are convolved, approximating the normals within a region accurately and turning the mipmapping of these functions into a linear operation, thereby providing a smooth filtering between mipmap levels.

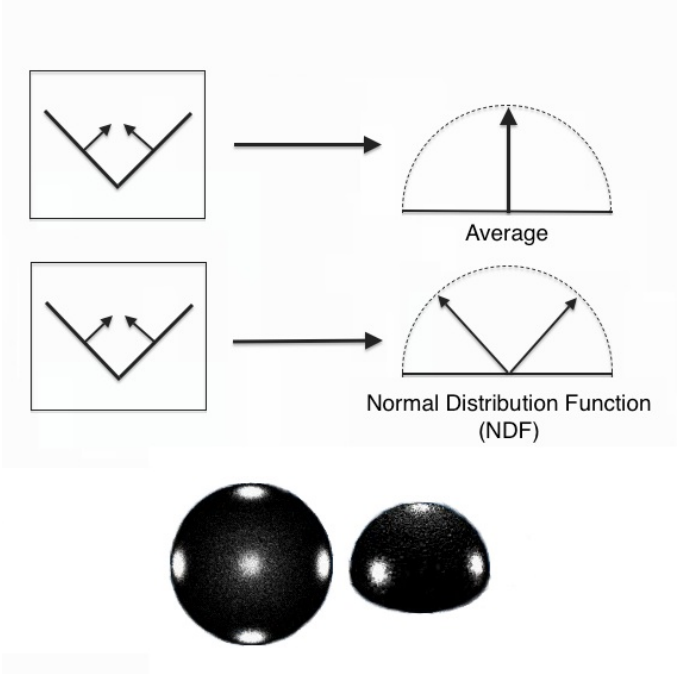


Figure 30: Normal Distribution Function (NDF).

Occlusion information is estimated in form of visibility (percentage of blocked rays) based simply on the transparency of all the intersecting objects. Only a single average value is stored to keep voxel data compact, which is a disadvantage for large thin objects since it causes a lack of view dependency (Figure 31).

Material color is encoded as an opacity weighted color value (alpha pre-multiplied) for better interpolation and integration during the rendering stage, as well as the normal information in order to properly account for its visibility.

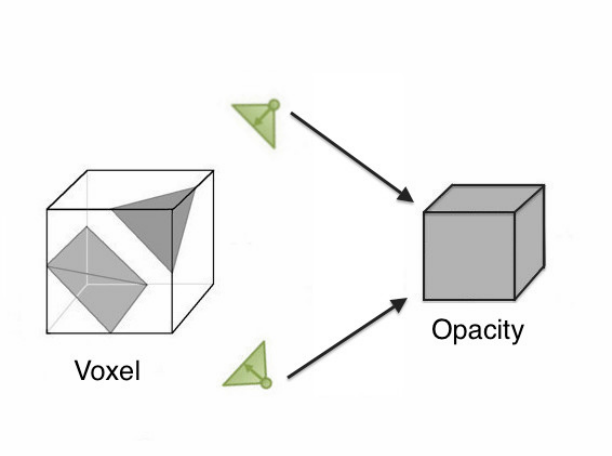


Figure 31: Opacity is stored as a single value inside a voxel, causing a lack of view dependency.

3.1.3 *Voxel Cone Tracing*

Before computing global illumination, information about the lighting in the scene needs to be added to the sparse voxel octree. For this purpose, the scene is rasterized from all light sources in order to determine the incoming radiance (energy and direction) for each visible surface fragment. This data is then stored in the leaves of the octree and mipmapped into the higher levels.

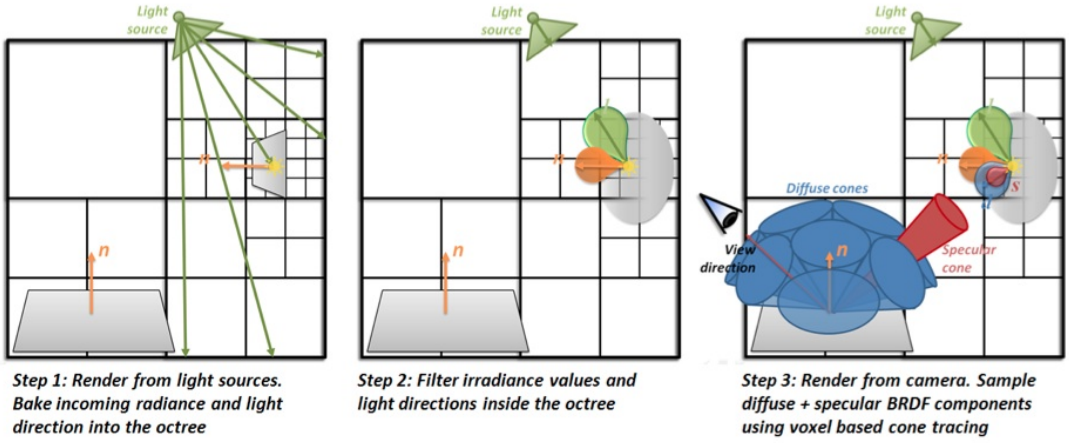


Figure 32: Direct lighting injection and indirect lighting computation. *Source:* Crassin et al. (2011)

Afterwards, the scene is rendered from the camera and for each visible surface fragment, multiple cones are launched along the hemisphere of the intersection point to estimate the diffuse contribution for the indirect illumination, by performing a final gathering (Jensen (1996)) and collecting illumination in the octree. To capture the specular contribution of the indirect illu-

mination (in the form of glossy reflections), a single cone is launched in the reflected direction. Finally, the approximation to global illumination is obtained by combining direct and indirect illumination (Figure 32).

This voxel cone tracing pass (Figure 33) is slightly different than the original cone tracing (2.4). The main idea is to step along the cone axis, retrieving the necessary data from the sparse voxel octree at the level corresponding to the cone radius and accumulating the lighting contributions using volume ray casting (2.3).

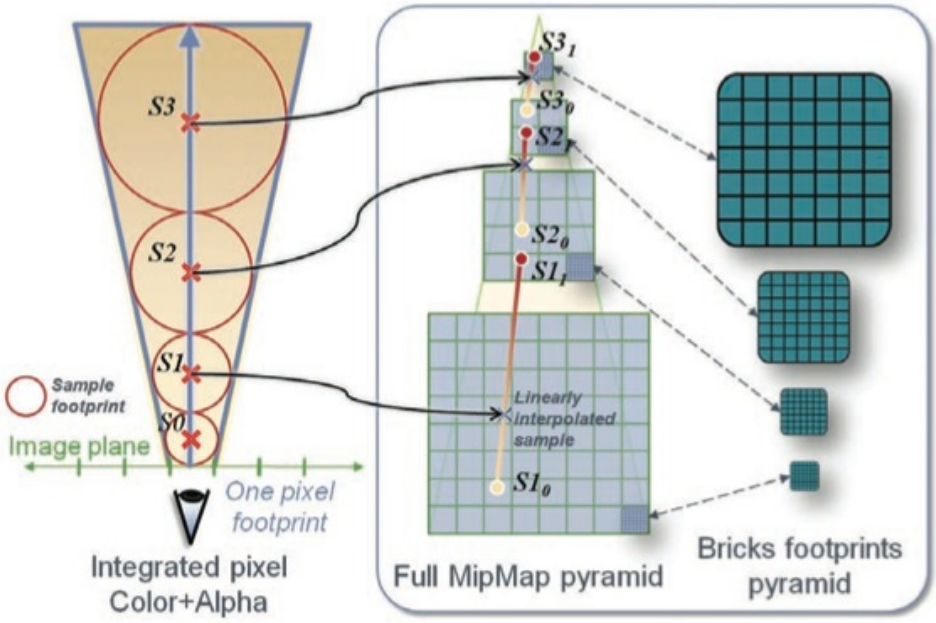


Figure 33: Voxel Cone Tracing. *Source:* Crassin et al. (2010)

Direct Illumination Injection

The scene is rendered from the light's view (using rasterization) and outputs a world position, generating a Reflective Shadow Map (2.2). Each pixel can be viewed as a photon that will bounce in the scene, and it will be stored in the sparse voxel octree as a direction distribution and an energy proportional to its angle with the light position.

These photons are stored at the leaves of the octree since they are located at the surface of the object. Since the octree has only collapsed empty voxels to produce the sparse representation, there is no risk to attempt to store data on a non existent leaf. Also, the resolution of the reflective shadow map is usually higher than the lowest level of the octree, so multiple photons might end up in the same voxel. These are combined by relying on an atomic add operation.

One of the main issues in this process is that voxels are repeated for neighboring bricks in order to allow using fast hardware filtering. One way to solve this is to scatter the lighting information in the voxels and its affected neighbors. However, this would lead to a lot of thread collisions. The approach selected to solve this problem is to perform 6 passes, two for each axis (Figure 34).

In the first x-axis pass, each thread will add voxel data from the current node to the corresponding voxels of the neighbor brick at its right. The next pass will simply copy data from the right to the left. After these two passes, values on the x-axis are coherent and the same will be done for the other y and z-axis. Since neighbor pointers have been added to the sparse voxel octree during its building phase, it is possible to access the neighbors efficiently, and thread collisions are avoided through this process, avoiding the need to use atomic operations.

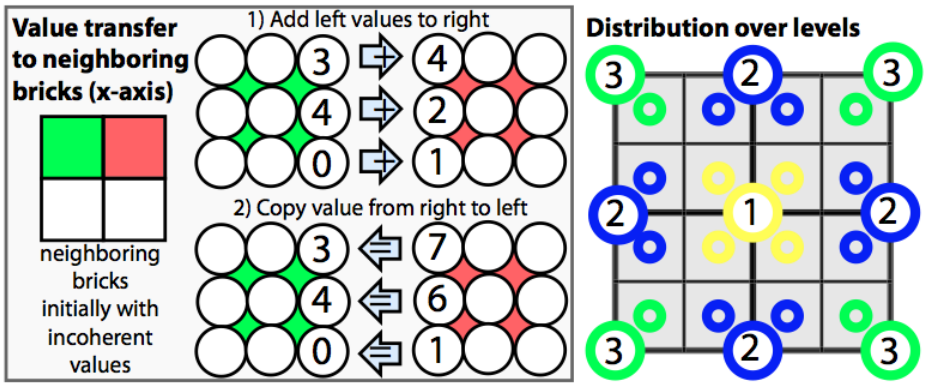


Figure 34: Data transfer between neighboring bricks and distribution over levels. *Source:* Crassin et al. (2011)

After this step, the lowest level of the sparse voxel octree has correct information and the values need to be mipmapped to the higher levels of the octree. In order to avoid unnecessary computations arising from the duplicated neighboring voxels, this step is performed in three separate passes, such that every thread has aproximately the same computational cost. The idea is to only compute the filtered results partially and take advantage of the transfer between bricks to complete the result (Figure 34).

The first pass computes the center voxel (yellow), the second pass computes half of the filtered value for the voxels in the center of the node's faces (blue), and the third pass computes a partial filering for the corner voxels (green).

After these three passes, the voxels on the higher levels of the octree are in the same situation as the leaves were after injecting the lighting information. Octree vertices might only contain a part of the result, but by applying the previously mentioned process to sum values across bricks, the correct result is obtained.

However, since direct light usually only affects a small part of the scene, launching one thread per leaf node would waste too many resources, filtering nodes that do not contain any photon and thus applying the filtering to zero values.

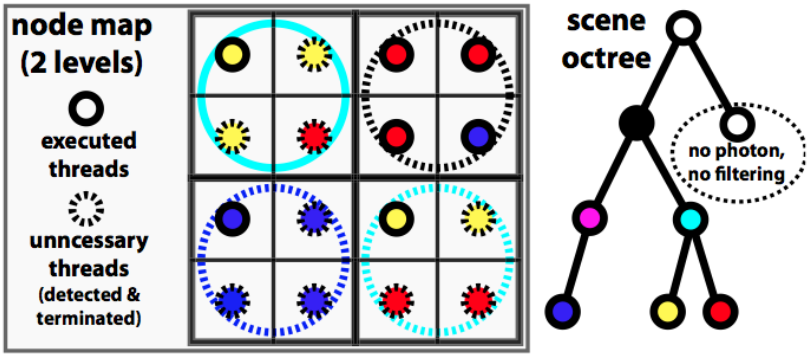


Figure 35: Node Map. *Source:* Crassin et al. (2011)

The approach used to reduce the number of threads and avoid filtering of zero values is to rely on a 2D node map, derived from the light view map (Figure 35).

The node map is a mipmap pyramid where the lowest level stores the indices of the 3D leaf nodes containing the corresponding photon of the light view map and the higher levels store the index of the lowest common ancestor for the preceding nodes of the previous level.

One thread is still launched for all pixels in the lowest node map but when a thread is descending the tree to find the node that it needs to compute the MIP-mapped value, it first checks the node map to verify if there is no common ancestor with another thread. If a common ancestor is found, it can assume that all threads passing through the same path afterwards will end up in the same voxel and thus the desired behavior is to terminate all threads except one. To achieve this, all threads that do not traverse the upper left pixel will be terminated and the remaining thread is in charge of computing the remaining filtered values.

Another problem in this representation is known as the two red-green wall problem. It is caused by averaging the values in the octree to a pre-integrated visibility value.

When two opaque voxels with very different values are averaged in the upper levels of the octree, the result can be different than what would be expected. For instance, two walls with different colors might end up as if they were semi-transparent. The same problem occurs for opacity, when a $2 \times 2 \times 2$ tile is half filled with opaque voxels and fully transparent ones, the resulting voxel would be half-transparent.

To counter this problem, an anisotropic voxel representation is used (Figure 36). It is built during the mipmapping process, when building or updating the sparse voxel octree with the lighting information.

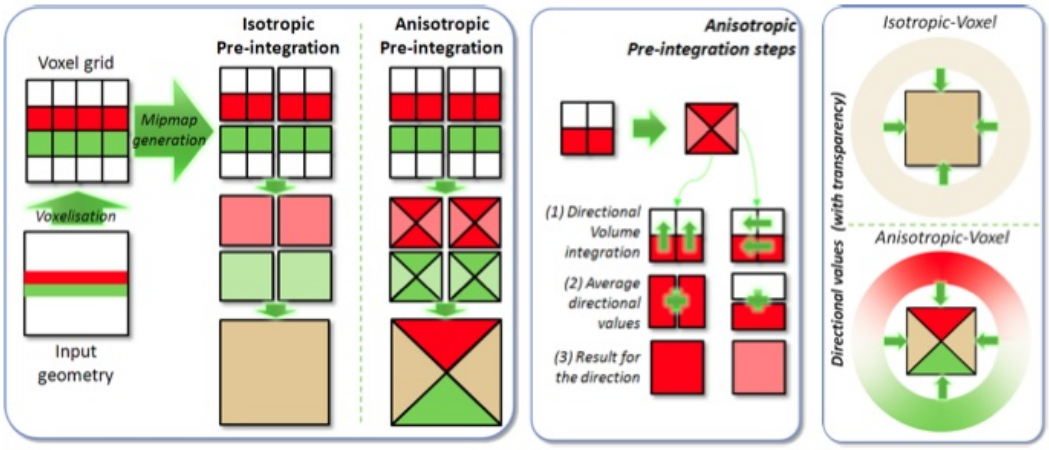


Figure 36: Anisotropic Voxel Representation. *Source:* Crassin et al. (2011)

Instead of storing a single channel of non-directional values, six channels of directional values are used, one for each major direction.

To generate the directional values, a first step of volumetric integration is performed in depth, followed by an average of the 4 directional values obtained. At render time, the voxel value is retrieved by finding the 3 closest directions to the view direction, and perform a linear interpolation between them.

Since fully opaque voxels can only turn transparent when their values are averaged in the lower levels of the sparse voxel octree, storing this directional representation for all the properties only needs to be accomplished for voxels that are not located on the leaves of the sparse voxel octree. In this manner, memory consumption is only increased by approximately 1.5x.

Indirect Illumination

For the indirect illumination computation, the shading of a voxel needs to be determined. In order to do this, the variations in the embedded directions and scalar attributes and the span of the cone that is currently accumulating the voxel need to be accounted for (Crassin et al., 2011).

The chosen approach is to translate the BRDF, the NDF and the span of the view cone into convolutions. These elements can be translated into convolutions, provided that they are represented as lobe shapes (Han et al., 2007; Fournier, 1992).

The Phong BRDF is considered, since its diffuse and specular lobes can be expressed as Gaussian lobes. The NDF can be computed from the length of the averaged normal vector that is stored in the voxels ($\sigma_n^2 = \frac{1-|N|}{|N|}$) (Toksvig, 2005). The distribution to the view cone is represented with a Gaussian lobe of standard deviation $\sigma_v = \cos(\psi)$, where ψ is the cone's aperture, by observing 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 going from the origin of the cone to the considered voxel (Figure 37).

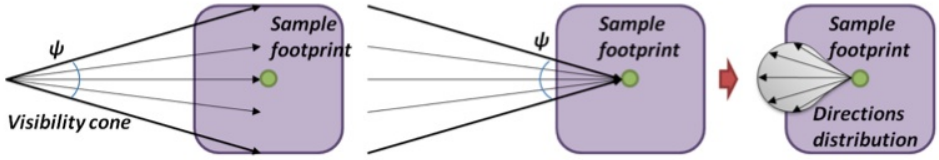


Figure 37: Directions distribution. *Source:* Crassin et al. (2011)

In order to determine efficiently in which surface points indirect illumination needs to be computed, deferred shading is employed. In each such surface point, a final gathering is performed by sending a few cones to query the illumination distributed in the octree.

3.2 REAL-TIME NEAR-FIELD GLOBAL ILLUMINATION BASED ON A VOXEL MODEL

The main idea of this method for calculating an approximation to global illumination in real-time (Thiedemann et al., 2011) is to generate a dynamic, view-independent voxel representation of the scene by relying on a texture atlas that provides visibility information of the objects in the scene.

This voxelized representation of the scene, in combination with reflective shadow maps, can then be used to compute one-bounce indirect illumination with correct occlusion at interactive frame-rates by shooting rays inside an hemisphere with a user-defined radius (near-field).

3.2.1 *Voxelization*

Thiedemann et al. (2011) propose a binary bitwise voxelization, being able to store the voxel grid inside a single 2D texture. By using texels with 128 bits, this extra dimension behaves as the third dimension in a conventional 3D grid.

Each bit represents a depth value along the voxelization arbitrary direction. In this way, each texel of a 2D texture represents a stack of voxels along the depth of the voxelization camera,

making it possible to encode a voxel grid as a 2D texture. Each bit encodes the presence of geometry at a certain depth along the voxelization direction.

Mipmapping creates a structure simliar to an octree. The mipmap is performed only in the x and y directions, preserving the depth dimension as shown in Figure 38.

Compared to a multi-value grid, this approach is very memory friendly.

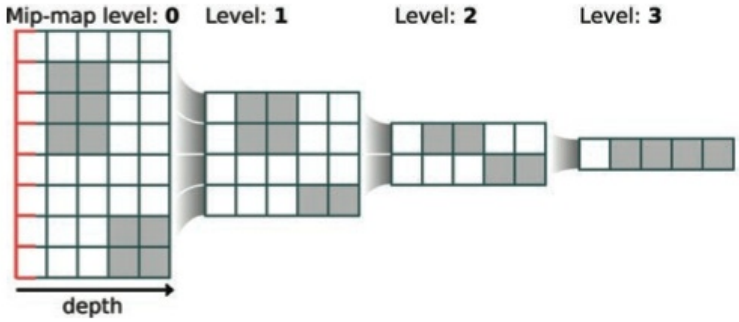


Figure 38: Mip-mapping. *Source:* Thiedemann et al. (2012)

3.2.2 *Visibility Queries*

In this work the grid behaves as an octree containing only visibility information. The grid is used only to compute visibility between two voxels.

In order to compute visibility, a ray-voxel intersection test is employed. A hierarchical binary voxelized scene is used to compute the intersection of a ray with the voxel grid (Figure 39).

Since the binary voxelization results in a hierarchical structure, it allows to decide on a coarse level if an intersection is to be expected in a region of the voxel grid or if the region can be skipped entirely.

This rendering method is based on the algorithm proposed by (Forest et al., 2009), but some improvements have been made in order to increase its performance and functionality.

The first step of the algorithm is to find if there is an intersection with the ray.

The traversal starts at the texel of the hierarchy that covers the area of the scene in which the starting point of the ray is located. To determine this texel, the starting point of the ray is projected onto the mip-map texture and used to select the appropriate texel at the current mip-map level. If the texel is found, a test is performed in order to determine if the ray hits any voxels inside the region it represents.

A bitmask is stored at each texel, representing a stack of voxels along the direction of depth. It is thus possible to use this bitmask to compute the bounding box covering the volume. The size of the bounding box depends on the current mip-map level.

After computing the bounding box corresponding to the current texel, the ray is intersected with it, generating two values: the depth where the ray enters the bounding box and the depth where it leaves the bounding box.

With these two values another bitmask can be generated, representing the voxels the ray intersects inside the bounding box. This bitmask (called ray bitmask) is compared with the bitmask stored in the texel of the mip-map hierarchy in order to determine if an intersection occurs and the node's children have to be traversed (Figure 39). If there is no intersection, the starting point of the ray is moved to the last intersection point with the bounding box and the mip-map level is increased. If an intersection occurs, the mip-map level is decreased to check if there is still an intersection on a finer resolution of the voxelization until the finest resolution is reached.

The algorithm stops if a hit is detected or if it surpasses the maximum length of the ray, which is defined by the user.

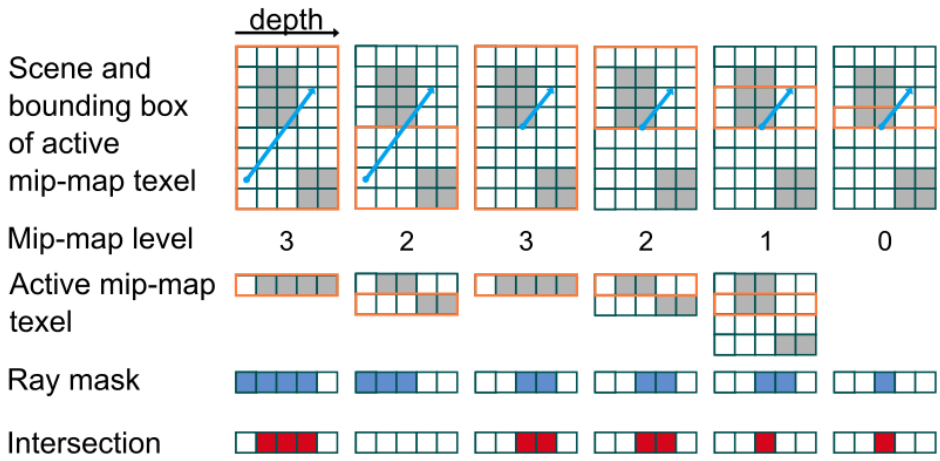


Figure 39: Hierarchy traversal in 2 dimensions. The blue arrow represents the current extent of the ray and in orange the bounding box of the current mip map levels is displayed. *Source:* Thiedemann et al. (2011)

3.2.3 *Rendering*

To compute indirect light the algorithm relies on the visibility queries described in Section 3.2.2, and on a reflective shadow map (Dachsbacher and Stamminger, 2005) containing direct light, position and normal for each pixel visible from the light position.

In order to compute the indirect light for a pixel in the camera view, a gathering approach is employed to compute one-bounce near-field illumination (Figure 40). **N** rays are cast using a cosine-weighted distribution, starting from the receiver position **x** with a maximum distance **r**.

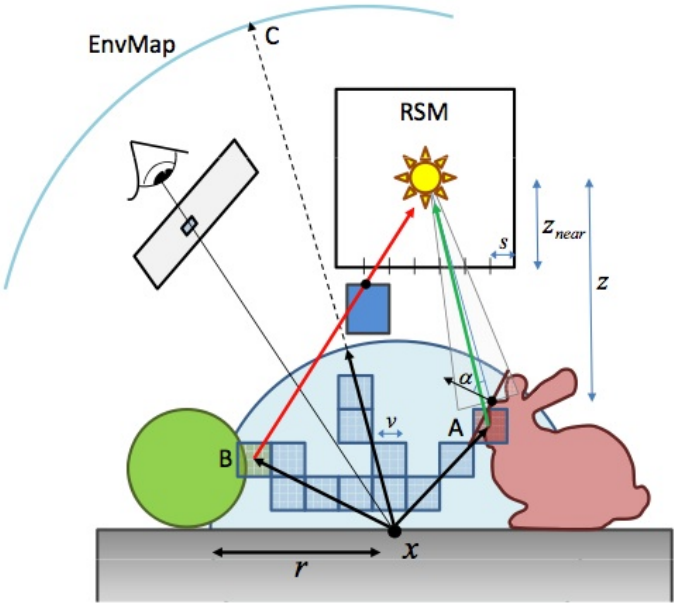


Figure 40: Near-field Indirect Illumination. *Source:* Thiedemann et al. (2011)

Intersection tests are performed for each ray to determine the first intersection point. If a non-empty voxel is hit along the ray, the direct radiance $\tilde{L}_i$ needs to be computed at the intersection point. This is performed by back-projecting the hit point to the reflective shadow map, allowing to read the direct radiance stored in the corresponding pixel of the reflective shadow map.

In case the distance between the 3D position of the hitpoint and the position stored in the pixel in the reflective shadow map is greater than a threshold ϵ, the direct radiance is invalid, and thus it is set to zero. The threshold ϵ has to be adjusted to the discretization **v**, the pixel size of the reflective shadow map **s**, the perspective projection and the normal orientation α: $\epsilon = max(v, \frac{s}{\cos\alpha} \cdot \frac{z}{z_{near}})$.

The radiance L_o at the receiver point **x** can be computed using Monte-Carlo integration with the formula (Thiedemann et al., 2012):

$$L_o(x) \approx \frac{\rho(x)/\pi}{N} \sum_{i=1}^{N} \tilde{L}_i(x, \omega_i)$$

where $\rho(x)/\pi$ is the diffuse BRDF at the receiver point, ω_i are **N** sample directions and $\tilde{L}_i(x, \omega_i)$ is the radiance that is visible at the hitpoint in sample direction ω_i, i.e., the value fetched from the RSM.

To provide some degree of filtering to the indirect light the result is downsampled with a geometry aware blur filter.

3.2.4 *Voxel Path Tracing*

It is possible to modify the rendering algorithm in order to extend its capabilities and create a path tracer based on the voxelized scene representation to better approximate global illumination, including computing glossy reflections.

Conventional path tracers are recursive algorithms that compute global illumination by launching several rays in multiple directions and using Monte-Carlo integration.

Some of the rays are launched in the direction of the light sources to compute direct illumination by verifying if the light is occluded by any object in the scene.

To compute indirect illumination, some rays are launched in randomly chosen directions in the hemisphere above the intersection point using importance sampling of the BRDF. When an intersection with another object occurs, the radiance is computed and the algorithm is restarted with the intersection point as the new starting point of the ray. Tracing stops when the ray reaches a certain length or a light source is intersected.

Voxel path tracing is an iterative version of this algorithm (Figure 41).

A reflective shadow map is created for each light source, storing wold positions, normals and radiance for each directly lit object in the scene.

The scene is rendered using deferred rendering into a geometry buffer storing world positions, normals and BRDF. For each fragment, a ray is launched in a random direction in the hemisphere above the world position using importance sampling of the BRDF.

Since the ray can intersect geometry not covered by the RSMs, a binary voxelization is not sufficient in this case. A 3D texture is used to create a voxel grid storing normals and BRDFs for the objects in the scene.

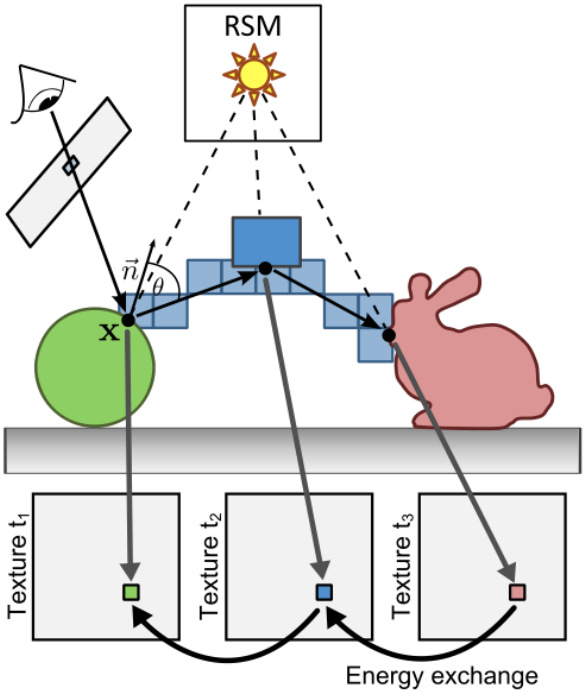


Figure 41: Voxel Path Tracing. At each intersection point, direct lighting, normal and BRDF are stored in a texture. Afterwards, the radiance is propagated backwards through the textures. *Source:* Thiedemann et al. (2011)

When an intersection occurs, direct illumination is computed by reprojecting the intersection point into the reflective shadow map. If the intersection point is present in the RSM (if he is not shadowed), radiance, normal and BRDF are obtained from the RSM and stored into a texture. If the intersection point is in shadow, the voxel grid is sampled in order to retrieve the normal and BRDF.

The intersection point is also stored into another texture in order to repeat the process and continue to trace the ray from the new position. The new randomly chosen direction in the hemisphere above the new position is computed by sampling the normal from the voxel grid.

After tracing the ray through the scene, the information stored in the texture at each intersection point is used to propagate the radiance from the last intersection point to the initial starting position. Light is transferred from the texture of the last intersection point to the previous intersection point and so forth until the final radiance value is computed.

Thiedemann et al. (2011) claims that the images obtained by voxel path tracing can visually compete with images obtained by conventional path tracing (Figure 42). However, overdarkening in indirect shadows of thin structures and visible discretized geometry present in

glossy reflections are still some issues, which are caused by voxelization. According to the results presented by Thiedemann et. al. the conventional path traced image took 28 minutes, while the voxel path tracing took only 2.2 seconds.

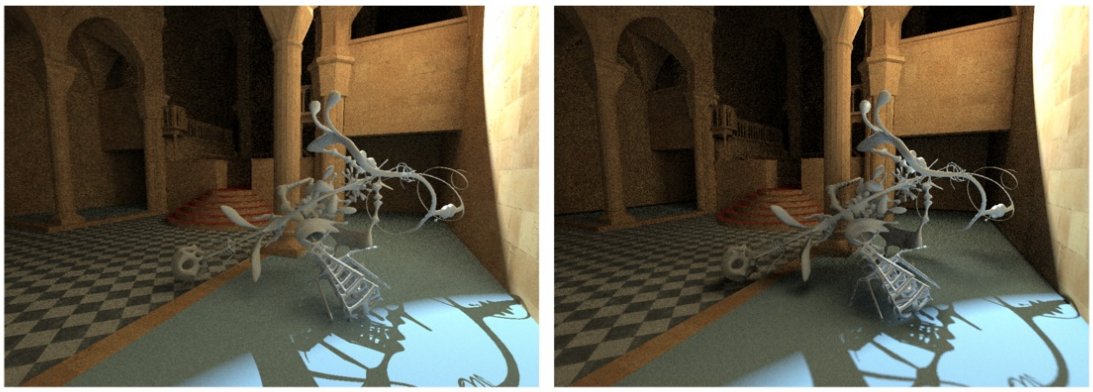


Figure 42: Left: Conventional Path Tracing. Right: Voxel Path Tracing. *Source:* Thiedemann et al. (2011)

3.3 LAYERED REFLECTIVE SHADOW MAPS FOR VOXEL-BASED INDIRECT ILLUMINATION

Sugihara et al. propose an algorithm that can be seen as the extension of near-field indirect illumination (3.2) to add cone tracing capabilities. This work maintains the separation between visibility information and direct light, but provides a multi level filtered version of the RSM, as opposed to the method described in section 3.1 where light and visibility were stored together (Figure 43).

One of the key features of this method (Figure 44) is to use the approach proposed in the variance shadow maps paper (Donnelly and Lauritzen, 2006). Instead of storing the world position in the RSM, both the depth and squared depth are stored. Filtering the texture will provide the mean and variance of the depth distribution. Considering a normal distribution for the depth values, the mean and variance are then used to compute the percentage of the surface within the cone being traced that receives direct light.

Similarly to section 3.2 the occlusion textures are mipmapped. However, Sugihara et al. also mipmaps the RSM.

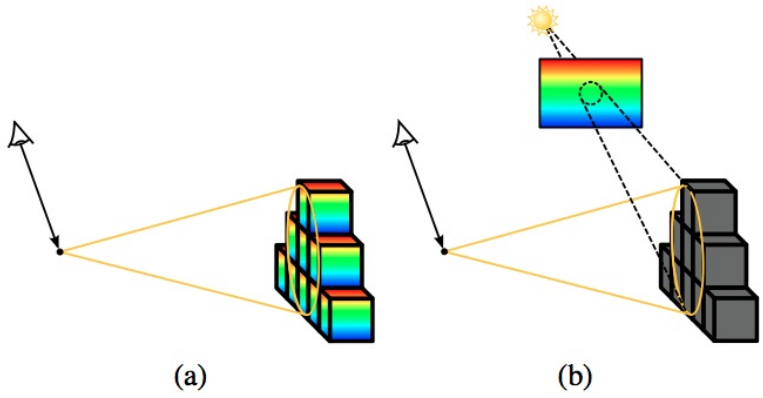


Figure 43: (a) represents tradicional voxel cone tracing where voxels encode multiple attributes. (b) stores only binary occupancy values in voxels and the rest of the attributes needed for voxel cone tracing are stored in the LRSM. *Source:* Sugihara et al.

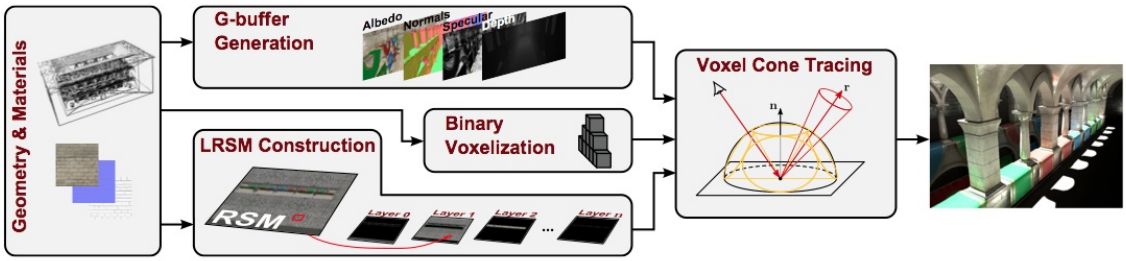


Figure 44: Pipeline of the Layered Reflective Shadow Maps for Voxel-based Indirect Illumination algorithm. *Source:* Sugihara et al.

The term layered in the methods title comes from the fact that in order to filter the RSM with mipmapping one assumes that there are no discontinuities both in depth as well as in normal values. Hence, to overcome this issue, the authors propose to partition the RSM based on a set of intervals for both depth and normal values.

When rendering, all layers are sampled, and those which both the depth range for the layer and normals fits the sample provide contributions to the outgoing radiance gathering.

The main contribution of this method is the small memory footprint when compared to algorithms using a full grid, and the simplicity of the data structures when compared to a sparse voxel octree.

On the downside, separating light from occlusion causes performance to be influenced by the number of lights during the cone tracing process.

3.4 GLOBAL ILLUMINATION USING VIRTUAL POINT LIGHTS

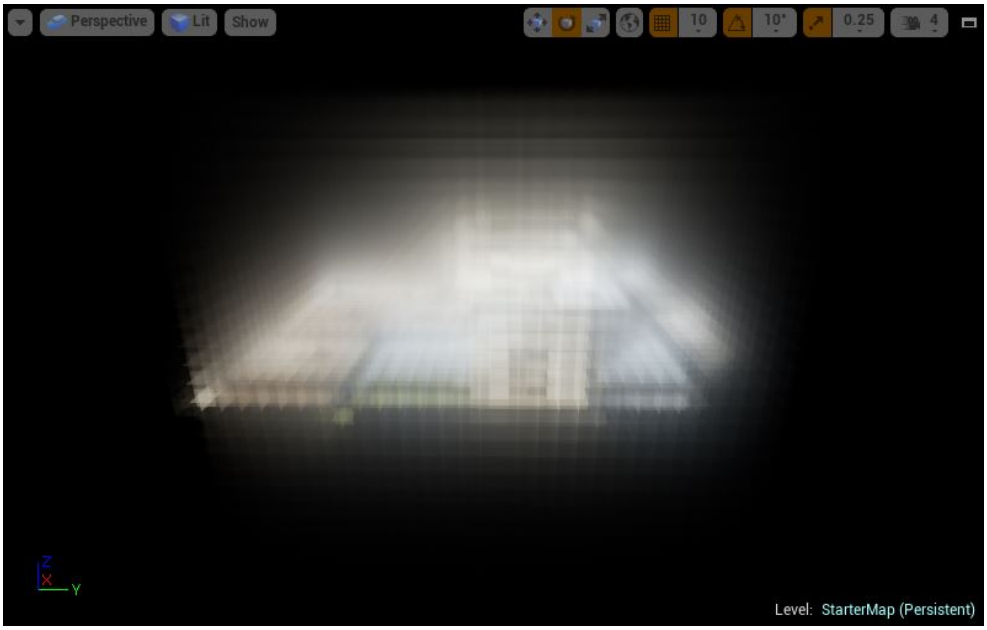


Figure 45: Light Propagation Volume visualization in the Unreal Engine. *Source:* UnrealEngine

Kaplanyan and Dachsbacher (2010) presented a technique called Light Propagation Volumes (LPV) that allows to approximate first-bounce indirect diffuse illumination in real time. The algorithm consists in four steps.

First, a Reflective Shadow Map (RSM) is generated and used to determine a set of secondary light sources, obtained from the objects in the scene (Figure 45). Basically, each object directly lit by a light source will reflect some of the direct lighting it receives, turning himself into a secondary light source, or Virtual Point Light (VPL).

These VPLs are then encoded into a distribution represented by Spherical Harmonics (SH) coefficients and injected into a 3D texture (the light propagation volume).

The next step consists in the propagation of the radiance through the LPV in multiple sequential iterations. Each iteration propagates the radiance from each of the VPLs to its six corresponding neighbors. In other words, each iteration corresponds to one propagation step of the lighting in the LPV space.

Finally, the radiance is extracted from the spherical harmonics coefficients stored in the light propagation volume and applied to the objects in the scene.

Doghramachi (2013) presented an alternative implementation of this algorithm. The main difference with the light propagation volumes technique is that it does not use a RSM to generate the VPLs. However, the algorithm has suffered some changes since the original work was published.

Initially, the algorithm only allowed the computation of an approximation of the diffuse indirect illumination. However, it has been later extended to be able to approximate glossy reflections. This comes at the cost of an extra higher resolution grid that is sampled using Voxel Cone Tracing to approximate specular reflections.

This changes not only the data structures needed for the execution of the algorithm, but also its structure, since some extra steps are needed during its execution.

First, a voxel grid representation for the scene is created using the hardware rasterizer. The voxelization algorithm is similar to the one previously presented in Section 2.1.2.

The scene is rendered and written into a buffer (the voxel grid) using atomic functions, creating a 3D grid representation of the scene. For diffuse indirect illumination, this grid contains the material color, occlusion and normal information of the geometry on the scene. For the higher resolution voxel grid, which shall be used for glossy reflections, only the material color and occlusion are stored. These grids are recreated each frame, allowing the scene to be fully dynamic and not relying on pre-calculations.

The lower resolution voxel grid used for the indirect diffuse illumination is kept at a relatively small size, thus some techniques have to be used in order to handle large environments. Several nested grids can be used, in which each grid has the same number of cells, but the size of the cells is increased (Figure 46). This allows to increase the detail of the diffuse indirect lighting near the viewer and use a coarser indirect lighting when far away from the viewer. Linear interpolation should be performed between the different grids to smooth the transitions between them.

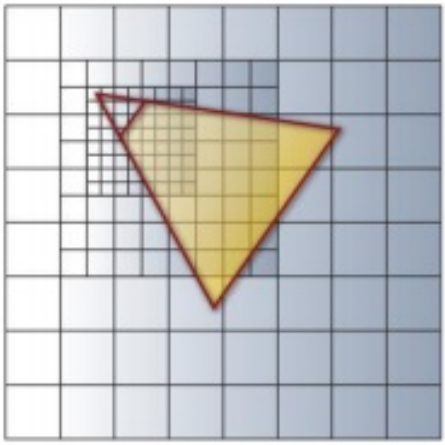


Figure 46: Nested Voxel Grids. *Source:* Kaplanyan and Dachsbacher (2010)

The lower resolution grids are then illuminated by each light source. The direct illumination is converted into virtual point lights stored as second-order spherical harmonics coefficients and the resulting coefficients are combined for each light source using the blending stage of the graphics hardware. Finally, the generated VPLs are propagated within the grid.

The higher resolution reflection grid is a hierarchical 3D grid created from the higher resolution voxel grid containing material color and occlusion and the lower resolution light propagation volume containing the propagated VPLs. The general idea is to extract ambient lighting from the SH coefficients and use this to light the samples of the voxel grid, storing the result into the reflection grid. In the end, the reflection grid is mipmapped in order to generate lighting information at multiple resolutions.

To generate approximate global illumination, the spherical harmonics coefficients are sampled and applied to the scene to obtain the approximation to indirect diffuse illumination. For the glossy reflections, the reflection grid is sampled using voxel cone tracing (3.1.3).

The algorithm can be subdivided into 5 distinct steps:

1. Voxelization
2. Direct Light Injection
3. Direct Light Propagation
4. Reflection Grid Creation and Mipmapping
5. Global Illumination Rendering

3.4.1 *Voxelization*

The objective of the voxelization pass is to generate a voxel grid encoding the attributes needed to compute the approximation to indirect illumination.

The voxelization scheme is the same described in Section 2.1.2. The main difference is that instead of having a single voxel grid, two nested grids are created for the indirect diffuse illumination by voxelizing the scene twice. These voxel grids have the same number of voxels, but one of the grids covers a volume eight times higher. To create the reflection grid afterwards, which is used to compute glossy reflections, a higher resolution voxel grid is generated that covers the same frustum as the finer indirect diffuse lighting grid (Figure 47).

The voxel grids are moved synchronously with the viewer camera and are snapped permanently to the grid cell boundaries to avoid flickering due to their discrete representation of the scene.

Since the voxel grid is a simplification of the actual scene, geometric information on the objects is lost during the voxelization pass. In order to amplify color bleeding for global illumination,

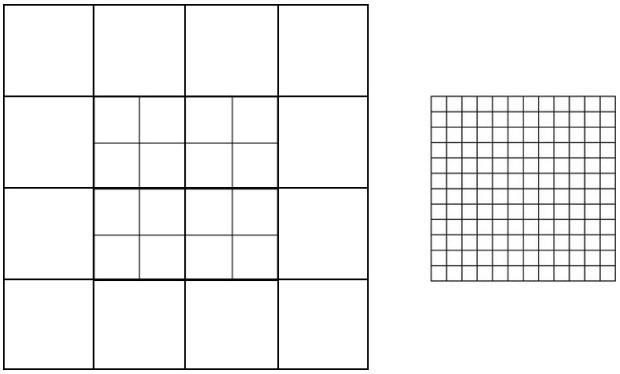


Figure 47: Illustration of the nested grids used for diffuse indirect lighting (left) and the reflection grid (right).

the color contrast value is calculated and used to write the material color of the fragments into the grid, thus giving preference to high contrast colors (colors with a high difference in their color channels). The contrast is computed with the formula:

$$\frac{|Color_{RRG} - Color_{GBB}|}{\sqrt{2} + Color_R + Color_G + Color_B}$$

Since normals can be opposite inside the same voxel, the closest face of a tetrahedron to which the current normal is closest is determined. A normal mask in generated in which each channel encodes a face of the tetrahedron. The normal is then written in the corresponding channel according to the tetrahedron face selected. Lately, this will allow to select the closest normal to the light vector when the voxels are illuminated, so that the best illumination can be computed. This leads however that sometimes the normal used is from a different geometry face than the color. However, since voxels condense information of the geometry inserted within its boundaries, this approximation does not have a significant negative impact on the result (Doghramachi, 2013).

3.4.2 *Direct Light Injection*

For each light source located within the grid boundaries, a quad with the size of the side of the voxel grid is rendered using hardware instancing.

Each instance corresponds to a depth value on the voxel grid, and all voxels that contain geometry information (or in other words that have the occlusion field set as opaque) are illuminated with the help of a shadow map, according to the type of the light source (Figure 48).

The radiance computed from the voxels is converted into a second-order spherical harmonic representation of virtual point lights and the second-order spherical harmonics coefficients for the three color channels are written into three 3D textures, one for each spherical harmonics channel. The results of all light sources are combined by using additive hardware blending.

In this way, virtual point lights that scale very well with an increasing number of light sources of different types are created entirely from the previously generated voxel grid (Doghramachi, 2013).

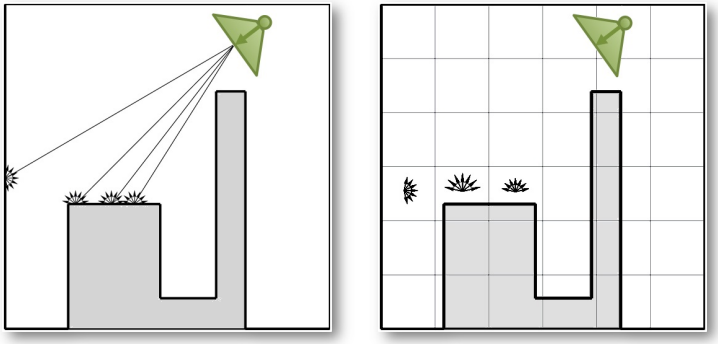


Figure 48: Lit surfaces are treated as secondary light sources and clustered into a voxel grid.

3.4.3 *Direct Light Propagation*

The propagation of the previously created virtual point lights across the grid is performed similarly to the light propagation volume technique proposed by Kaplanyan and Dachsbacher (2010).

In the original LPV technique, each virtual point light cell propagates its light to the surrounding six neighbor cells. However, in this implementation, a gathering is performed instead of a scattering in order to gather the radiance contribution from the six neighboring voxels.

During the propagation, the previously created voxel grid (3.4.1) is used to compute the occlusion of the light transport from the neighbor cells in order to avoid light leaking. The light contribution from each face of the neighboring voxels is then weighted by the occlusion and accumulated, storing the result in the grid. This step is then performed again using the results from the first propagation in an iterative manner until the light distribution is visually satisfying (Figure 49).

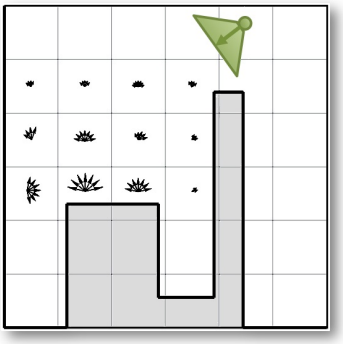


Figure 49: Virtual Point Light are propagated in the Voxel Grid.

3.4.4 *Reflection Grid Creation and Mipmapping*

For the creation of the reflection grid, a compute shader is used, launching one thread per voxel in the higher resolution grid created for this purpose during the voxelization pass.

First, each thread retrieves the material color and occlusion stored in the voxel grid. The voxel position is then reprojected into the finer grid used for indirect diffuse lighting and the spherical harmonics coefficients are sampled in order to extract the lighting component. This lighting is applied to the material color and the result is weighted according to the distance to the grid center before being stored into the finer level of the hierarchical reflection grid. Since the glossy reflections only cover a relatively small volume around the viewer, this ensures that by fading out the values lately during the voxel cone tracing pass, a smooth fade-out of the reflections is obtained as the camera moves through the scene. Finally, standard mipmapping is employed to filter the light through the coarser mipmap levels.

3.4.5 *Global Illumination Rendering*

A full-screen quad is rendered using deferred rendering and the world-space position, normal and direct lighting of each pixel is retrieved from a geometry buffer generated in a previous pass.

In order to apply diffuse indirect illumination to the scene, the previously generated grids containing the propagated virtual point lights are sampled according to the world position and surface normal of the fragment using linear hardware filtering. The results of both grids are linearly interpolated in order to obtain a smooth transition between the grids.

For the glossy reflections, voxel cone tracing (3.1.3) is used to launch a cone in the view direction, gathering the lighting information in the reflection grid at the corresponding mipmap

level. The result is weighted according to the distance to the grid center in order to ensure a smooth fade out of the reflections as the viewer moves.

Finally, direct and indirect diffuse lighting are combined together with the glossy reflections in order to generate an approximation to global illumination.

3.5 VOXEL GLOBAL ILLUMINATION

NVIDIA has recently presented a software library called VXGI that allows the computation of approximate global illumination in real-time (NVIDIA). The algorithm is very similar to the one presented by Crassin et al. (2011) and uses Voxel Cone Tracing (2.4) to accumulate light contributions in the scene.

The main difference with previous approaches that also used Voxel Cone Tracing is the use of a different data structure to encode voxel data. This data structure, called 3D clipmap, is a hierarchical data structure similar to a 3D texture mipmap but in which the finer levels are clipped such that they don't exceed a pre-defined size (Tanner et al., 1998).

The algorithm starts by encoding opacity and emittance information in the 3D clipmaps and then performs Voxel Cone Tracing to compute approximate global illumination in real time.

3.5.1 *3D Clipmap*

Clipmaps are multidimensional textures that were proposed as a solution to reduce the memory footprint of very large textures (Tanner et al., 1998), (Asirvatham and Hoppe, 2005).

When a texture is exceedingly large to fit in the memory of the GPU, one common approach is to subdivide the texture in multiple tiles that are loaded into the GPU memory as they are needed for the rendering process.

A clipmap is a data structure that allows to avoid having to manage the texture as a tiled resource by acting as a cache where the data necessary for the rendering of the current frame is loaded.

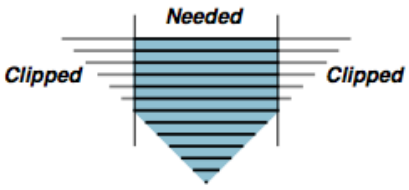


Figure 50: Side view of a clipmap within a mipmap structure. *Source:* Tanner et al. (1998)

A clipmap is a partial representation of a mipmap (Figure 50), in which the levels are clipped according to a user-defined maximum size. So, clipmaps only differ from mipmaps in the finer

levels of detail. The finer levels have the same spatial resolution as a mip-map texture, but multiple finer levels of the clipmap have the same number of voxels (Figure 51).

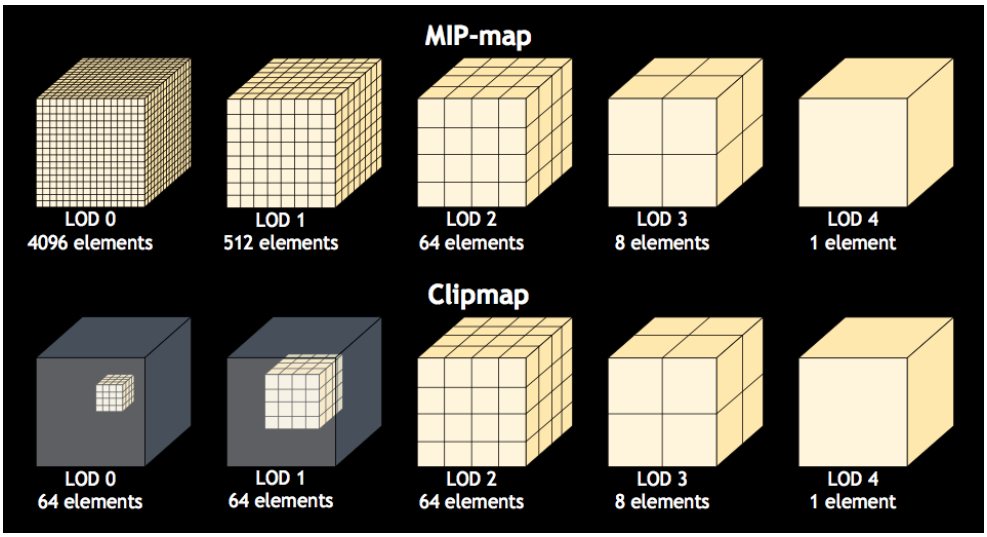


Figure 51: Comparison between mipmaps and clipmaps in their sizes and spatial resolution. *Source:* NVIDIA

By centering the clip-map around the viewer camera, the regions near the camera are represented with finer levels of detail, while regions far from the camera are represented by coarser levels of detail. This maps nicely to voxel cone tracing needs, since there is a higher spatial density of pixels near the camera due to the perspective projection used in 3D applications. The clipmap must however be updated when the camera moves in order to maintain all the information needed around the camera to render the current frame. In order to prevent having to update the clipmap completely each time the camera moves, it is possible to take advantage of the fact that usually only a small part of the clipmap needs to be updated (a great portion of the data is already present in the clipmap since it was needed in the previous frame). In order to update the clipmap incrementally with the needed data, toroidal addressing is used (Tanner et al., 1998). Toroidal addressing is a method that guarantees that a point in space always maps to the same point in the texture (Figure 52).

To do this, the clipmap size and the world space position of the point to be updated are needed. The clipmap address is then simply computed with:

$$ClipmapCoord.xyz = frac(WorldPosition.xyz / ClipmapSize.xyz)$$

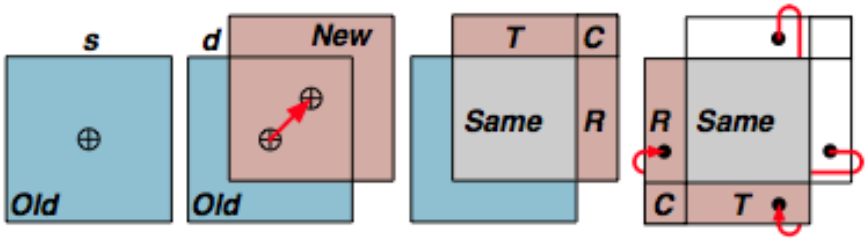
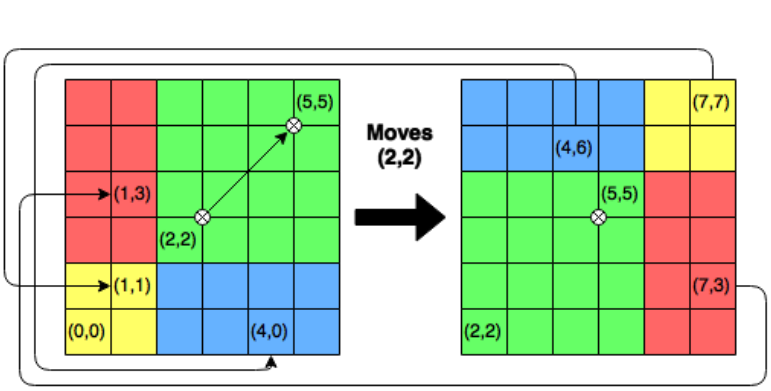


Figure 52: Toroidal Addressing. *Source:* Tanner et al. (1998)



World Position	Clipmap Position
0	0
1	0.16667
2	0.33333
3	0.5
4	0.66667
5	0.83333
6	0
7	0.16667

Figure 53: Example of toroidal addressing. The table on the right shows the mapping from the world coordinates to clipmap coordinates for a clipmap with size 6.

In Figure 53 an example is presented for a clipmap with a size 6×6. The clipmap is initially centered at (3,3). It moves by (2,2) and the new center becomes (5,5). The clipmap needs to be updated in such a way that the new data overwrites the data that is not necessary anymore. This is done by mapping the new data to the same position in the clipmap as the data that is unnecessary.

These features turn 3D clipmaps into a potential solution to the problem of the memory consumption and update time of the voxel data needed for the voxel cone tracing pass, particularly for very large scenes.

3.5.2 *Voxelization*

The main objective of the voxelization process remains the same: encode opacity and emittance information of the scene in order to be used in the voxel cone tracing pass.

However, the voxelization process used in this implementation is slightly different than the previously described approaches (section 2.1).

First of all, two voxelization passes are performed: opacity voxelization and emittance voxelization. The opacity voxelization pass computes approximate opacity of space, while emittance voxelization computes approximately the amount of light the geometry in the volume emits or reflects into all directions.

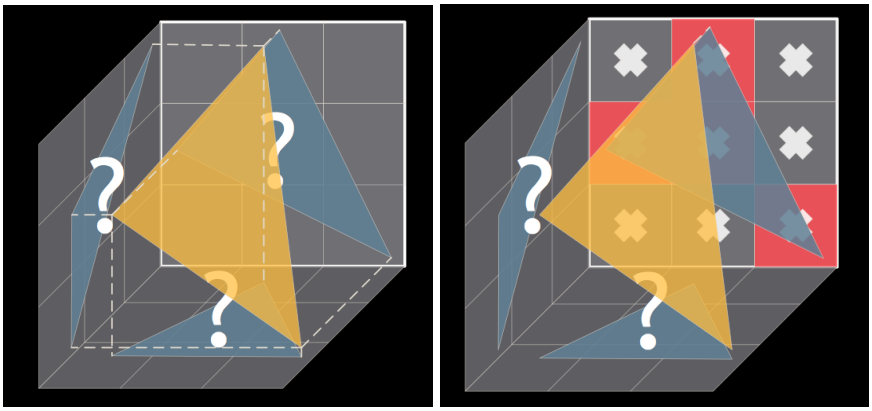


Figure 54: Voxelization using MSAA. *Source:* NVIDIA

The voxelization process starts by choosing the projection plane in order to maximize the number of fragments generated during rasterization. However, the rasterization is performed in Multisample Anti Aliasing mode (MSAA - 8 samples) in order to produce multiple samples for the rasterized triangles (Figure 54).

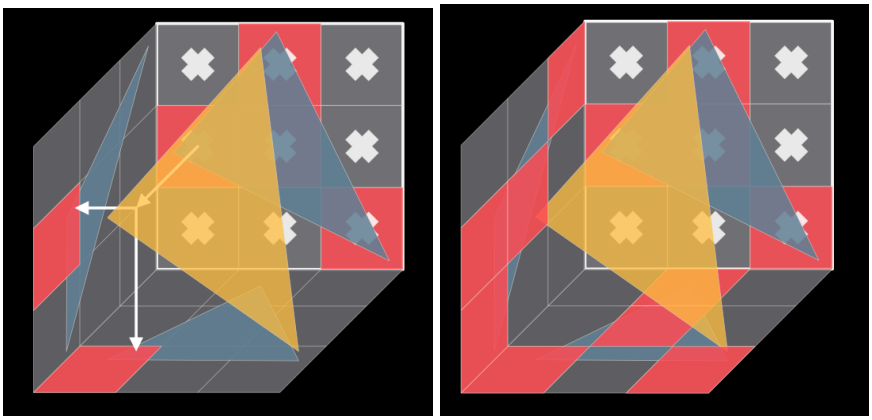


Figure 55: Reprojection of the MSAA samples into the other two projection planes. *Source:* NVIDIA

In the opacity voxelization pass, the depth coordinate is then computed for each sample and the samples are reprojected into the other two planes (Figure 55).

Finally, a bit count function is used in order to convert the coverage masks obtained by MSAA to an opacity value.

To make the object look thicker when viewed from other directions, in order to avoid flat objects like walls being viewed as transparent objects, the opacity is slightly augmented in the two reprojection planes.

For the emittance voxelization pass, simply using the covered samples and interpret their count as light intensity can cause some noticeable artifacts. Objects can change their brightness abruptly if they move, due to alterations to the number of covered samples (small objects or large voxels in the coarse clipmap levels)(Figure 56). In order to solve this, adaptive supersampling is used.

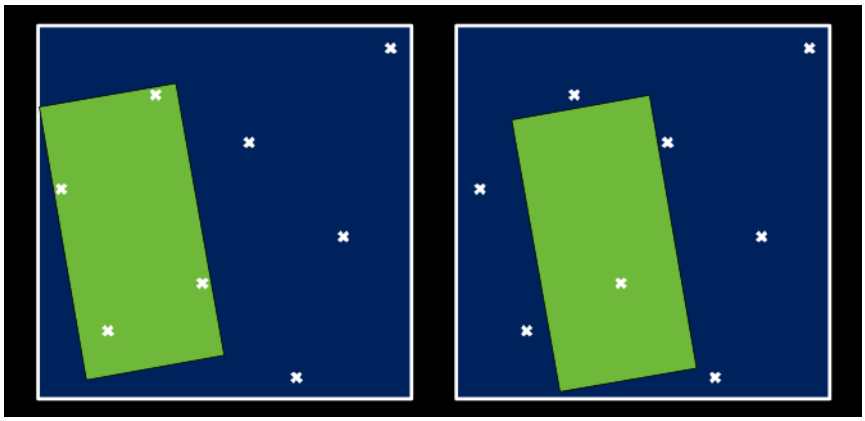


Figure 56: Movements from small objects or inside large voxels can cause changes in the number of covered samples. *Source:* NVIDIA

The coverage masks are then used to generate the brightness value of the fragment and the emittance is computed with the help of a shadow map generated previously.

To compute the levels of the clipmap, the scene can be voxelized using the resolution of the finer level and then downsampled in order to create the coarser levels. However, when using a clipmap the finer level does not possess all the information of the scene, thus being impossible to generate the coarser levels simply by downsampling the finer level.

The approach suggested is to rasterize the scene at multiple resolutions in a single pass by using Geometry Shader Instancing to rasterize each triangle multiple times with different viewport sizes.

This introduces a problem: when an object moves from one clip level to another its coarse representation changes, which can be visible as changes in the approximate indirect illumination.

To reduce this, the downsampled and voxelized representations are interpolated using a weight ramp that is computed based on the distance of the voxels to the camera position.

3.6 SUMMARY

All the algorithms described in this chapter share a similar structure.

They start by creating a representation of the scene based on voxels, using a voxelization technique (2.1). The direct lighting information is then used to compute the radiance of each voxel and a technique based on ray tracing or cone tracing is used to generate the approximation to inditect illumination.

The largest difference between these algorithms is the data structure used to store voxel data. If a single large regular structure (voxel grid) is used, a lot of memory is wasted with empty voxels. The problem is aggravated if voxel data at multiple resolutions is needed, such as for voxel cone tracing. If we want to use a data structure of this kind to approximate indirect illumination in large 3D scenes, multiple grids are required (with a relatively high resolution), which does not scale well considering the amount of memory provided by current GPUs.

In section 3.4, a solution is proposed that uses multiple hierarchical voxel grids centered around the camera. Multiple grids with low resolution are used to perform voxel cone tracing in a later pass to gather approximate indirect diffuse lighting, while a single grid is used for approximate indirect specular lighting (in the form of glossy reflections).

The low resolution of the diffuse grids turns this solution scalable (to a certain extent), but also a poor approximation to indirect diffuse lighting in the scene. Another problem is that glossy reflexions are only accounted for near the camera, since only a single grid is used.

In order to avoid wasting memory with empty voxels, section 3.1 proposes using a sparse voxel octree to store a hierarchical representation of the scene using voxels.

This solution separates the octree definition (node pool) from the data stored inside the octree (brick pool). Each set of 2^2 nodes is associated with a 3^3 brick for interpolation purposes. This causes that, for very dense scenes, it could be possible that the sparse voxel octree representation would need more memory than a regular voxel grid. However, since most 3D scenes are very sparse, this is not a major concern in the general case.

The biggest disadvantage of this kind of data structure, besides the complex implementation, is that it is very slow to create and update. This makes it suitable for scenes predominantly static, but not for scenes containing lots of dynamic objects.

Another disadvantage is that the sparse voxel octree needs to be traversed each time a value needs to be sampled, which turns the sampling of this data structure slower than sampling a regular grid with mipmapping.

Section 3.2 proposes using a 2D texture to encode information about the geometry in the scene, in which each texel represents a stack of values in the voxelization depth. Since the voxelization is binary, encoding only information about the geometry of the scene (opacity), the parameters needed to compute the shading of a voxel need to be retrieved from a reflective shadow map.

This approach is very efficient in terms of memory, but only allows the computation of an approximation of indirect diffuse lighting. It can be extended in order to compute approximate indirect specular lighting and multiple bounces of light by using a multivalued voxel grid and create a voxel path tracer, at the expense of a significant loss in performance.

Other disadvantages of the algorithm are that it needs a texture atlas in order to map the triangles to the 2D texture storing the voxelization of the scene, and that some computation is made in the CPU side in order to inject the result of the voxelization pass into the voxel grid.

An extension of this algorithm is described in section 3.3 that also allows the computation of glossy reflections in real time.

The main differences from the approach described in section 3.2 are that the RSM is subdivided in layers and voxel cone tracing is used to compute the approximation to indirect illumination.

This approach is very efficient in terms of memory, due to the binary voxelization that only needs to store information about occlusion in the scene. However, it is very dependent on the number of light present in the scene, due to the need to maintain and sample a layered reflective shadow map for each light source.

In section 3.5, a data structure named clipmap is proposed in order to store the information needed for the voxel cone tracing pass computing the approximation to global illumination.

This data structure is very similar to multiple hierarchical nested voxel grids. The memory consumption is however much lower than for nested voxel grids, offering at the same time fast sampling and updates through the use of toroidal addressing.

The scene needs however to be voxelized twice. Once to determine the occlusion in the scene, and once to determine the emission of each voxel (since they use a slightly different different process).

4

ANALYSIS AND IMPLEMENTATION

To perform an analysis of the proposed solutions for the computation of an approximation to global illumination using a voxel-based representation of the scene, some algorithms from the previous chapter 3 were implemented.

First, the technologies used for the implementation of the algorithms are mentioned (section 4.1).

An implementation of the Sparse Voxel Octree Global Illumination (SVOGI - 3.1) technique is described in (section 4.2).

The Light Propagation Volumes technique described in section 3.4 was also implemented and is detailed in section 4.3.

The implementation of the technique using a binary voxel representation of the scene described in section 3.2 is presented in section 4.4.

Finally, the results obtained by the implemented algorithms are analysed and discussed in section 4.6

4.1 TECHNOLOGICAL CHOICES

The most important choice for the implementation of the chosen algorithms is between the graphics programming interface to use.

OpenGL was chosen to handle the real-time 3D graphics of the applications. OpenGL is a cross-platorm graphics API for drawing 2D and 3D graphics. It is well documented and a wide quantity of books and examples are available for free through the internet. Altough it is multiplatform, it lacks some functionalities such as resource loading and window and input handling. There are however free libraries that offer these functionalities, turning this into a small issue.

Current GPUs offer the possibility to be used not only for graphical purposes, but also for more general computation. Since GPUs offer many unified cores, they are perfect for highly

parallelizable tasks. There are some platforms used for this purpose, such as CUDA or OpenCL. However, since we will use the most recent versions of OpenGL (and capable hardware), it is also possible to use the OpenGL Shading Language (GLSL) to create compute shaders to perform these operations. DirectX also offers this functionality in the name of High-Level Shader Language (HLSL).

Since OpenGL doesn't offer asset import, window management, or input handling, some libraries have to be used to counter these problems. There are a lot of candidates for these functions. However, there is a collection of libraries that simplify the interaction with OpenGL. It is called Very Simple * Libs (VSL - Lighthouse3d). It still depends on other libraries, but provides a wrapper to perform all the operations in a very simple manner. It also features a profiler for the CPU and GPU, which will come in handy to measure the performance of the implemented algorithms.

4.2 INTERACTIVE INDIRECT ILLUMINATION USING VOXEL CONE TRACING

Figure 57: Voxel representation of the scene and Voxel Cone Tracing result

In order to compute an approximation to global illumination using Voxel Cone Tracing, a hierarchical representation of the scene made of voxels is needed (Figure 57).

To achieve this, a voxel grid can be computed from a voxelization process (2.1) and the information of the voxels stored into a mipmapped 3D texture. But this approach has a problem: the voxel grid wastes a lot of memory encoding empty voxels.

The algorithm described in section 3.1 uses a Sparse Voxel Octree in order to reduce the memory usage needed to store the voxels after voxelization. However, the use of this kind of

data structure introduces a higher access time to the data during the cone tracing pass, since the sparse voxel octree has to be sampled recursively until the level desired is reached.

In order to better assess the trade-off between the usage of a full voxel grid and a sparse voxel octree, both versions of the algorithm have been implemented. Both are very similar in their structure, but the introduction of the sparse voxel octree increases the number of passes performed the algorithm, as well as the way voxel data is stored.

4.2.1 *Voxel Cone Tracing with a Full Voxel Grid*

The algorithm starts by voxelizing the scene in a single pass using the hardware rasterizer in order to generate voxel fragments (2.1.2), which contain the data needed to store the radiance into the voxel grid.

Then, a Shadow Map is generated. For fragments which are not in shadows, the radiance is computed and the result is stored in the highest definition level of the 3D texture containing the voxel grid.

The next step uses mipmapping to filter the 3D texture, obtaining the hierarchical representation of the scene needed for the voxel cone tracing pass.

The final pass of the algorithm uses Voxel Cone Tracing (3.1.3) to launch multiple cones through the scene to gather diffuse and specular indirect illumination and add the indirect lighting contribution to the direct lighting, resulting in approximate global illumination.

Thus, the algorithm can be subdivided in four passes:

1. Voxelization
2. Light Injection
3. Mipmapping
4. Voxel Cone Tracing

Voxelization

The objective of the voxelization pass is to convert the surface of the objects represented with triangles to a volumetric representation stored in a voxel grid. The first thing to do is to determine the volume that has to be voxelized.

To perform the voxelization an orthographic projection is defined in such a way that its frustum covers the volume to be voxelized. Since every triangle inside the orthographic volume has to

generate fragments in order to avoid missing information in the voxel grid, the depth test and face culling need to be disabled prior to the rendering call. Also, the resolution of the voxelization is controlled by altering the viewport before issuing the draw call.

This voxelization algorithm uses a vertex shader, geometry shader and fragment shader to produce voxel fragments containing world position, normal and material color.

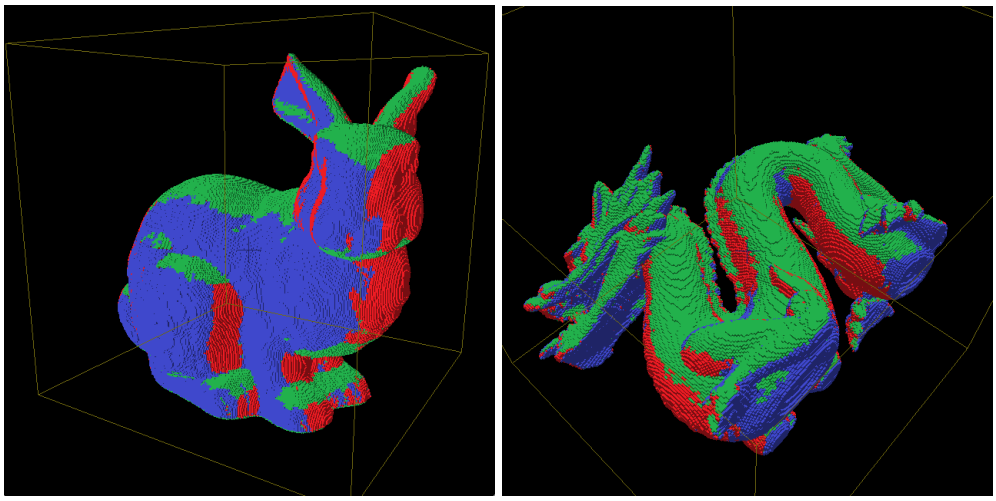


Figure 58: GPU Voxelization. Red: projection along x-axis. Green: projection along y-axis. Blue: projection along z-axis

The vertex shader simply outputs the world position, normal and texture coordinates to the geometry shader.

In order to produce the maximum number of fragments per triangle, each triangle must be projected along its dominant axis (Figure 58). The first step is to determine the normal of the triangle and finding which of the x,y,z components is greater. Since information about the three vertices of each triangle is needed to compute its normal, this is done in the geometry shader.

The next step is then to swizzle the vertices of the triangle, in such a way that it matches the orthographic projection during rasterization. Then, by multiplying the swizzled vertice coordinates by the projection matrix, the screen coordinates of the triangle are obtained (Listing 4.1).

```
vec4 aux[3];
if (dominantAxis == worldSpaceNormal.z)
{
  aux[0] = vec4(vPosition[0].xyz, 1.0);
  aux[1] = vec4(vPosition[1].xyz, 1.0);
  aux[2] = vec4(vPosition[2].xyz, 1.0);
}
else if (dominantAxis == worldSpaceNormal.y)
```

```
{
  aux[0] = vec4(vPosition[0].xzy, 1.0);
  aux[1] = vec4(vPosition[1].xzy, 1.0);
  aux[2] = vec4(vPosition[2].xzy, 1.0);
}
else if (dominantAxis == worldSpaceNormal.x)
{
  aux[0] = vec4(vPosition[0].zyx, 1.0);
  aux[1] = vec4(vPosition[1].zyx, 1.0);
  aux[2] = vec4(vPosition[2].zyx, 1.0);
}

vec4 screenPos[3];
screenPos[0] = projectionMatrix * aux[0];
screenPos[1] = projectionMatrix * aux[1];
screenPos[2] = projectionMatrix * aux[2];
```

Listing 4.1: Computation of screen coordinates with vertex swizzling

Since the 2D fragments generated after rasterization only take into account pixels that intersect triangles through their center, the triangles must be expanded so that every pixel touched by a triangle generates a fragment. This is done by shifting the screen coordinates of the triangle outwards by the size of a pixel's diagonal (Listing 4.2).

```
void expandTriangle(inout vec4 screenPos[3])
{
  vec2 side0N = normalize(screenPos[1].xy - screenPos[0].xy);
  vec2 side1N = normalize(screenPos[2].xy - screenPos[1].xy);
  vec2 side2N = normalize(screenPos[0].xy - screenPos[2].xy);
  screenPos[0].xy = screenPos[0].xy +
      normalize(-side0N+side2N)*pixelDiagonal;
  screenPos[1].xy = screenPos[1].xy +
      normalize(side0N-side1N)*pixelDiagonal;
  screenPos[2].xy = screenPos[2].xy +
      normalize(side1N-side2N)*pixelDiagonal;
}
```

Listing 4.2: Triangle expansion for conservative rasterization

This process, known as conservative rasterization implies the computation of a screen space bounding box of the triangle before shifting its vertices outwards (Listing 4.3). This bounding box will serve to discard extra fragments generated during rasterization in the fragment shader.

```
gBBox.xy = min(screenPos[0].xy, min(screenPos[1].xy, screenPos[2].xy));
gBBox.zw = max(screenPos[0].xy, max(screenPos[1].xy, screenPos[2].xy));
gBBox.xy -= vec2(pixelDiagonal);
gBBox.zw += vec2(pixelDiagonal);
```

Listing 4.3: Computation of the screenspace triangle's bounding box

Finally, the fragment shader is in charge of storing the voxel fragments and discard the excess fragments generated during conservative rasterization using the bounding box passed by the geometry shader.

The shadow map could be used to compute and write the radiance directly into the voxel grid. However, using a voxel fragment list (Listing 4.4) allows to separate the voxelization pass from the light injection pass, at the expense of some more memory used to store voxel data. The main advantage of separating voxelization from light injection is that the scene can be voxelized only once (if all objects are static) and the lighting kept dynamic.

```
// Voxel fragment
struct FragmentData
{
  uint position;
  uint color;
  uint normal;
};

// Voxel fragment list
layout(binding = 1, std430) buffer Fragments
{
  FragmentData fragmentList[];
};
```

Listing 4.4: Voxel Fragment List

The fragment is tested against the bounding box passed by the geometry shader. If the fragment is not inside the bounding box, it is discarded, and thus not appended to the voxel fragment list. Then, the voxel data is stored with the help of an atomic counter in order to avoid voxel fragments overwriting each other (Listing 4.5).

```
layout(binding = 1, offset = 0) uniform atomic_uint fragmentCounter;

void splat(in vec3 pos, in vec3 normal, in vec2 tc, in int fc)
{
  vec4 material;
  if (texCount == 0)
    material = diffuse;
  else
  {
    material.rgb = pow(texture(tex, tc).rgb, vec3(GAMMA));
    material.a = diffuse.a;
  }

  vec4 fragmentColor = material;
  vec4 fragmentNormal = vec4(normalize(normal) * 0.5 + 0.5, 1.0);
  vec4 fragmentCoord = vec4(pos * 0.5 + 0.5, 1.0);

  fragmentList[fc].position = fragmentCoord;
  fragmentList[fc].color = fragmentColor;
  fragmentList[fc].normal = fragmentNormal;
}

void main()
{
  vec2 bboxMin = floor((gBBox.xy * 0.5 + 0.5) * voxelResolution);
  vec2 bboxMax = ceil((gBBox.zw * 0.5 + 0.5) * voxelResolution);
  if (all(greaterThanEqual(gl_FragCoord.xy, bboxMin)) && all(lessThanEqual(
      gl_FragCoord.xy, bboxMax)))
  {
    int fragmentNumber = int(atomicCounterIncrement(fragmentCounter));
    splat(gPosition, gNormal, gTexCoord, fragmentNumber);
  }
  else discard;
}
```

Listing 4.5: Writing of the voxel fragments into the voxel fragment list

Light Injection

In order to fill the voxel grid with the voxel fragments, the attribute-less capability of the core profile in OpenGL can be used, allowing to issue a draw call with no vertex buffer attached in order to launch a certain number of threads.

The objective is to launch one thread for each fragment present in the voxel fragment list, so the number of fragments written in the previous pass needs to be determined. This information is already stored in the atomic counter used to store the voxel fragments in the voxel fragment list.

To avoid reading the data back to the CPU to launch the draw call, an indirect draw call can be used. The idea is that the draw call parameters are stored in a buffer that can be altered by a shader (Listing 4.6).

```
struct DrawArraysIndirectCommand
{
  GLuint  count;
  GLuint  primCount;
  GLuint  first;
  GLuint  baseInstance;
};

DrawArraysIndirectCommand command = { 0, 1, 0, 0 };
glGenBuffers(1, &mIndirectBuffer);
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, mIndirectBuffer);
glBufferData(GL_DRAW_INDIRECT_BUFFER, sizeof(DrawArraysIndirectCommand), &
    command, GL_STATIC_DRAW);
```

Listing 4.6: Indirect Draw Call Structure

The *count* and *first* parameters are used to draw a certain number of primitives, starting at *first* and ending at *first* + *count -1*. The *primCount* parameter is the count of the number of instances that are to be rendered and *baseInstance* is the offset to any instanced vertex attributes.

The primitive count is set initially at one since only one instance needs to be launched for the rendering call writing the fragments into the voxel grid. By binding the *count* field of the indirect draw call structure to the atomic counter buffer in the voxelization pass, all the information needed to issue the indirect draw call is available.

An indirect draw call is issued with attribute-less rendering, where the number of vertices are read from the indirect draw call buffer, launching in this way one thread per entry on the voxel fragment list (Listing 4.7).

```
glBindBuffer(GL_DRAW_INDIRECT_BUFFER, mIndirectBuffer);
glDrawArraysIndirect(GL_POINTS, 0);
```

Listing 4.7: Issuing an indirect draw call

In the vertex shader, each thread uses the *gl_VertexID* implicit input in order to access its corresponding entry in the voxel fragment list, retrieving the world position, color and normal of each fragment. The normal and color of the voxel fragment are used together with a shadow map and the light parameters to determine the diffuse shading of each fragment, according to the Lambertian reflection model (Listing 4.8).

```
FragmentData voxel = fragmentList[gl_VertexID];

vec4 fragmentPosition = voxel.position;
vec4 fragmentColor = voxel.color;
vec3 fragmentNormal = voxel.normal * 2.0 - 1.0;

vec3 N = normalize(fragmentNormal);
vec3 L = normalize(lightPosition);
float NdotL = max(dot(N,L), 0.0);

float shadowing = calcShadowing(fragmentPosition * 2.0 - 1.0);
vec3 incomingRadiance = shadowing * (fragmentColor.rgb * lightColor.rgb *
   NdotL);

imageAtomicRGBA8Avg(voxelGrid, ivec3(fragmentPosition.xyz * voxelResolution)
   , vec4(incomingRadiance, 1.0));
```

Listing 4.8: Light injection into the voxel grid

Now that the shading of each fragment is known, the fragments need to be stored in the corresponding voxel in the voxel grid, averaging the values that fall into the same voxel. The voxel grid is 3D texture with a size matching the voxelization resolution and with a RGBA8 texture format.

Since multiple fragments can try to store data into the same voxel, atomic operations have to be used. However, image atomic operations have severe limitations in OpenGL: image atomic operations can only be used on integer images, either signed or unsigned, with the GL_R32I/r32i or GL_R32UI/r32ui formats.

In order to surpass this limitation, it is possible to emulate an atomic average in RGBA8 images using the *imageAtomicCompSwap* function (Crassin and Green, 2012), (Listing 4.9).

The idea of the algorithm is to loop on each write, exchanging the value stored in the voxel grid with the moving average. The loop stops when the value stored in the voxel grid has not been changed by another thread. The moving average is computed using the alpha component of the RGBA format as a counter of the number of fragments that have been joined together.

```
vec4 convRGBA8ToVec4(uint val)
{
  return vec4(float((val & 0x000000FF)), float((val & 0x0000FF00) >> 8U),
     float((val & 0x00FF0000) >> 16U), float((val & 0xFF000000) >> 24U));
}

uint convVec4ToRGBA8(vec4 val)
{
  return (uint(val.w) & 0x000000FF) << 24U | (uint(val.z) & 0x000000FF) <<
     16U | (uint(val.y) & 0x000000FF) << 8U | (uint(val.x) & 0x000000FF);
}

void imageAtomicRGBA8Avg(layout(r32ui) coherent volatile uimage3D grid,
    ivec3 coords, vec4 value)
{
  value.rgb *= 255.0;
  uint newVal = convVec4ToRGBA8(value);
  uint prevStoredVal = 0;
  uint curStoredVal;

  while((curStoredVal = imageAtomicCompSwap(grid, coords, prevStoredVal,
     newVal)) != prevStoredVal)
  {
    prevStoredVal = curStoredVal;
    vec4 rval = convRGBA8ToVec4(curStoredVal);
    rval.rgb = (rval.rgb * rval.a); // Denormalize
    vec4 curValF = rval + value;  // Add
    curValF.rgb /= curValF.a; // Renormalize
    newVal = convVec4ToRGBA8(curValF);
  }
}
```

Listing 4.9: RGBA8 Image Atomic Average Function

A problem may arise from using the alpha channel of an RGBA8 texture as a counter for the number of merged samples. Only 8 bits are available so if more than 256 voxel fragments try to merge their values in the same voxel, overflow may occur. The fragments are generated using a viewport matching the voxelization resolution, so overflow would only happen if more than 256 triangles occupied the same pixel, at the same depth.

Another problem due to the use of the alpha channel as a counter is that the final alpha value stored in the grid, which should represent the opacity of that voxel, is not correct.

To correct the opacity value, a thread must be launched for each voxel of the voxel grid. Each thread will simply access its corresponding voxel and if it is not empty, modify the alpha value so that the voxel is considered fully opaque (Listing 4.10).

```
layout (local_size_x = 8, local_size_y = 8, local_size_z = 8) in;

layout(binding = 0) uniform sampler3D srcMip;
layout(binding = 1, rgba8) writeonly uniform image3D dstMip;

void main()
{
  ivec3 pos = ivec3(gl_GlobalInvocationID);
  vec4 srcColor = texelFetch(srcMip, pos, 0);

  if(srcColor != vec4(0))
  {
    srcColor.rgb = srcColor.rgb;
    srcColor.a = 1.0; // Set voxel fully opaque
    imageStore(dstMip, pos, srcColor);
  }
}
```

Listing 4.10: Correction of the occlusion value in the finer level of the voxel grid

Since the number of threads necessary is known beforehand, the correction of the alpha values of the voxel grid is performed simply by launching a compute shader with the number of threads matching the resolution of the voxel grid.

Since this operation is considered computationally expensive (since it must be done for every voxel of the voxel grid), some tests were performed in order to compare the performance of compute shaders against vertex shaders using attribute-less rendering (which is the method used in Crassin et al. (2011)). The results are presented in Table 1, whcich shows that using a compute shader offers some advantages against using a vertex shader with attribute-less rendering.

The increase in performance is particularly noticeable when using anisotropic grids, since the operations must be performed in multiple grids.

	Isotropic (1 Grid)		Anisotropic (6 Grids)	
	Occlusion Correction	Mipmap	Occlusion Correction	Mipmap
Attribute-less	2.87	3.58	17.29	21.86
Compute	2.11	1.11	11.31	7.91

Table 1: Performance measurements (in ms) of the occlusion correction and mipmap passes, comparing attribute-less rendering and compute. The voxel grids were stored in 3D textures with a size 256^3

Mipmapping

Now that the 3D texture has data about the lighting on the scene at maximum resolution, it is necessary to create the lower mipmap levels in order to have an hierarchical representation of the lighting in the scene to use during the voxel cone tracing pass.

This is done level by level, launching a compute shader with a number of threads equal to the resolution of the mipmap level to be filled. Each thread accesses the next higher mipmap level and samples the eight voxels that correspond to the voxel to be filled, averaging their values and storing the result using an image store operation. By using hardware trilinear interpolation, a single texture sampling instruction can be used in the shader to retrieve the average of the eight voxels. Since the mipmapping is performed level by level, no atomic operations are needed in order to ensure that the results stay coherent.

Note that although OpenGL has an instruction to perform mipmapping for a given texture, it's implementation with NVIDIA drivers 353.30 causes a memory error when considering 3D textures with 512^3 pixels, even on a graphics card with 3GB of RAM.

Voxel Cone Tracing

The voxel grid encodes information about the geometry of the scene (all voxels containing geometry have an occlusion higher than zero) and the direct lighting information at multiple resolutions, providing an hierarchical representation of the scene.

With this information it is possible to compute an approximation to indirect illumination by launching cones and sampling the voxel grid at different resolutions, according to the cone aperture.

This pass requires a previous pass using a deferred rendering approach (Figure 59). A full-screen quad is rendered and the fragment shader simply samples the geometry buffer in order

to retrieve the positions, normals, and material colors. Direct lighting is also sampled from a texture computed previously with the help of a shadow map.

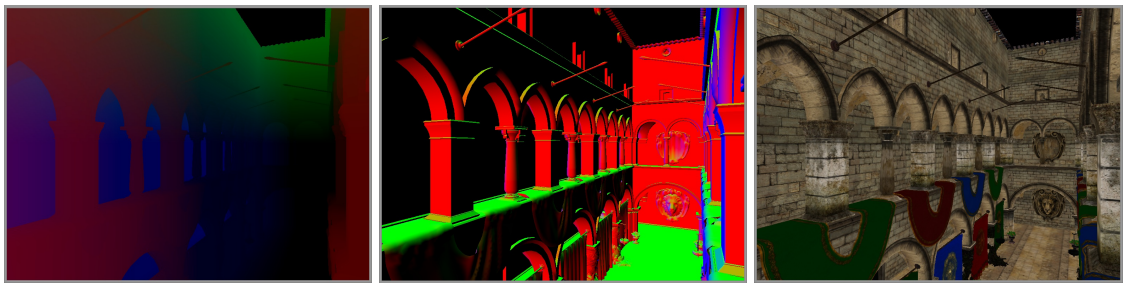


Figure 59: Geometry buffer storing world positions (left), normals (middle) and material colors (left)

With this information it is possible to compute the direction of the reflected ray and a single cone with a small aperture is launched in order to capture the specular contribution to indirect illumination (Listing 4.11). The aperture of the specular cone affects the roughness of the specular reflection. The wider the cone, the smoother the specular reflection (Figure 60).

```
specular = voxelConeTrace(position, R, coneRatio, maxDist);
```

Listing 4.11: Launching a cone in the reflected direction to compute the glossy reflection

For the diffuse indirect illumination, 5 cones with a large aperture are launched in the hemisphere around the normal. A cone is launched in the direction of the normal, while the other four are launched in different directions, making an angle of 45 degrees with the normal, with the help of the tangent and bitangent vectors. The cones are weighted according to the angle made with the normal (Listing 4.12).

```
// (dot(normal, normal) == 1)
indirectDiffuse += voxelConeTrace(position, N, coneRatio, maxDist);

// vectors at 45 degrees dot == 0.707
indirectDiffuse += 0.707 * voxelConeTrace(position, normalize(normal +
    tangent), coneRatio, maxDist);
indirectDiffuse += 0.707 * voxelConeTrace(position, normalize(normal -
    tangent), coneRatio, maxDist);
indirectDiffuse += 0.707 * voxelConeTrace(position, normalize(normal +
    bitangent), coneRatio, maxDist);
indirectDiffuse += 0.707 * voxelConeTrace(position, normalize(normal -
    bitangent), coneRatio, maxDist);
```

Listing 4.12: Launching 5 cones to gather the diffuse indirect illumination

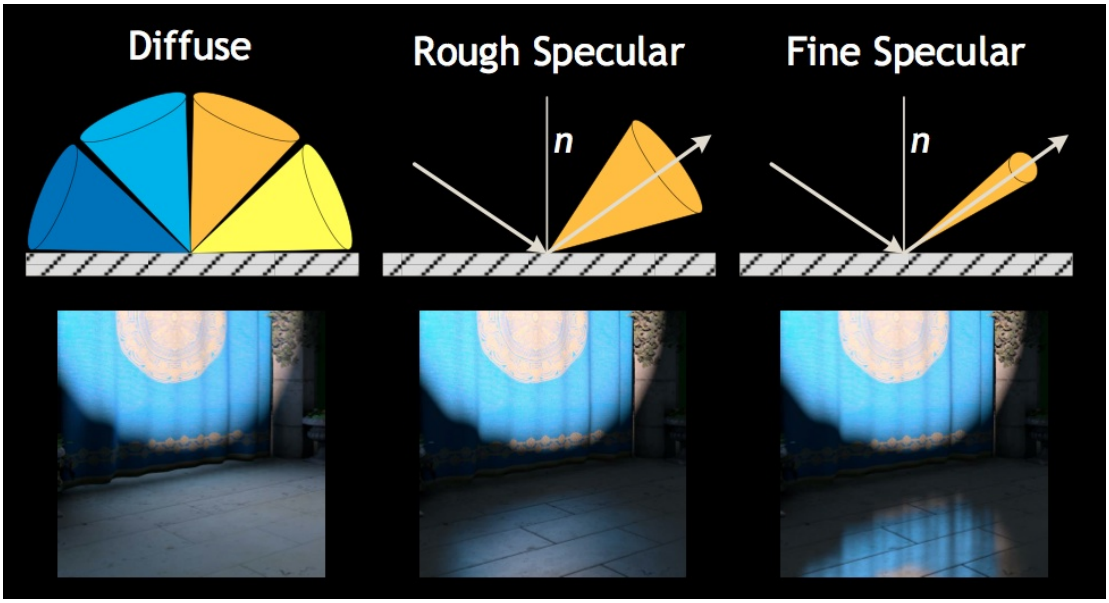


Figure 60: Diffuse and specular cones for the voxel cone tracing pass. The wider the cone, the rougher the reflection. *Source:* NVIDIA

Tracing cones through the voxel grid is very similar to using volume ray casting (section 2.3). The difference is that instead of shooting rays and sampling them at equally spaced intervals, the rays have a thickness that increases along the tracing path and the distance between samples increases as the sampling position gets farthest from the cone apex.

```
// origin, dir, and maxDist are in texture space
// dir should be normalized
// coneRatio is the cone diameter to height ratio
vec4 voxelConeTrace(vec3 origin, vec3 dir, float coneRatio, float maxDist)
{
  vec3 samplePos = origin;
  vec4 accum = vec4(0.0);

  // define the starting sample diameter
  // a little bit more than the actual voxel diameter
  float minDiameter = 2.0 / voxelResolution;

  // advance the starting point to avoid self-intersection
  float startDist = minDiameter;

  float dist = startDist;
  float sampleLOD = 0;
  float sampleInitial = max(minDiameter, coneRatio * dist);
```

```
  while (dist <= maxDist && accum.a < 1.0)
  {

    // ensure the stepping size is always at least minDiameter
    // to avoid lots of overlapped samples
    float sampleDiameter = max(minDiameter, coneRatio * dist);

    // convert diameter to LOD using the sample size
    // for example:
    // log2(1/256 * 256) = 0
    // log2(1/128 * 256) = 1
    // log2(1/64 * 256) = 2
    sampleLOD = log2(sampleDiameter * voxelResolution);

    vec3 samplePos = origin + dir * dist;
    vec4 sampleValue = voxelFetch(samplePos, sampleLOD);

    sampleValue.a = 1.0 - pow(1.0 - sampleValue.a, (sampleDiameter/2.0)/
        sampleInitial);
    float sampleWeigth = (1.0 - accum.a);
    vec3 accumVal = (accum.rgb * accum.a);
    accum.rgb = accumVal + sampleWeigth * (sampleValue.a * sampleValue.rgb);
    accum.a += sampleValue.a * sampleWeigth;

    // advance in smaller steps and take twice
    // the samples to avoid banding artifacts
    dist += sampleDiameter/2.0;
  }
  return vec4(accum.rgb, 1.0);
}
```

Listing 4.13: Voxel Cone Tracing through a voxel grid

The information needed to trace a cone is: its starting point, its direction, its aperture and the maximum distance that will be traveled until the tracing is stopped (Listing 4.13).

The first thing to do is set the starting point. The fragment position that has been retrieved from the geometry buffer corresponds to the world position of the surface of an object to which indirect illumination has to be computed. However, that surface also has had its lighting contribution added to the voxel grid. So, in order to avoid self intersection, the starting point of the cone has to be advanced by at least the size of the diameter of a voxel on the last level of the voxel grid.

The maximum distance and the aperture are set by the user, but the aperture is actually represented as the the cone diameter to height ratio. The idea is to sample the voxel grid from the starting position along the direction of the cone, accumulating the samples using front-to-back alpha blending, until the maximum distance set by the user is surpassed or the alpha value containing the accumulated occlusion saturates.

The distance to the next sample is increased by the cone's diameter of the current sample and by associating the volume that a voxel represents in each sample with the diameter of the cone, the volume increases between each sample. This makes the sample resemble a sphere and the proper mipmap level of the voxel grid is chosen according to the size of that spherical volume since each voxel corresponds to a different volume in space in different mipmap levels. This is done by using $mipLevel = \log_2(d)$, where d is the diameter of the sphere.

Since the full grid is stored in a 3D texture, it is possible to sample the voxels using hardware quadrilinear filtering (Listing 4.14).

```
vec4 voxelFetch(vec3 voxelPos, float sampleLOD)
{
  vec4 filteredIrradiance = textureLod(voxelGridIrradiance, voxelPos,
     sampleLOD);
  return filteredIrradiance;
}
```

Listing 4.14: Fetching voxel data from a 3D texture grid using hardware quadrilinear interpolation

This algorithm already allows to compute a very good approximation to indirect illumination (Figure 61). However, some artifacts are noticeable (Figure 62).

Since the start position and stepping size between each sample during the cone tracing is the same for every cone with the same aperture, some banding artifacts can appear when tracing glossy reflections. This is a well-known problem on volume rendering approaches (Crane et al., 2007), and common ways to solve the problem are to increase the number of samples taken for each cone by reducing the stepping size, and jittering the start position of the cone. The approach taken was to reduce the stepping size by half during the traversal, thus doubling the number of samples taken. To account for the smaller step size, the weight of the sample must be corrected using $\alpha_s' = 1 - (1 - \alpha_s)^{\frac{d'}{d}}$, where α_s' is the new alpha value for the sample, α_s is the sampled value from the voxel grid, d' the new sampling step and d the original sampling step. This approach had no visible impact on the performance of the algorithm, and it was sufficient to remove visible banding on the scene (Figure 63).

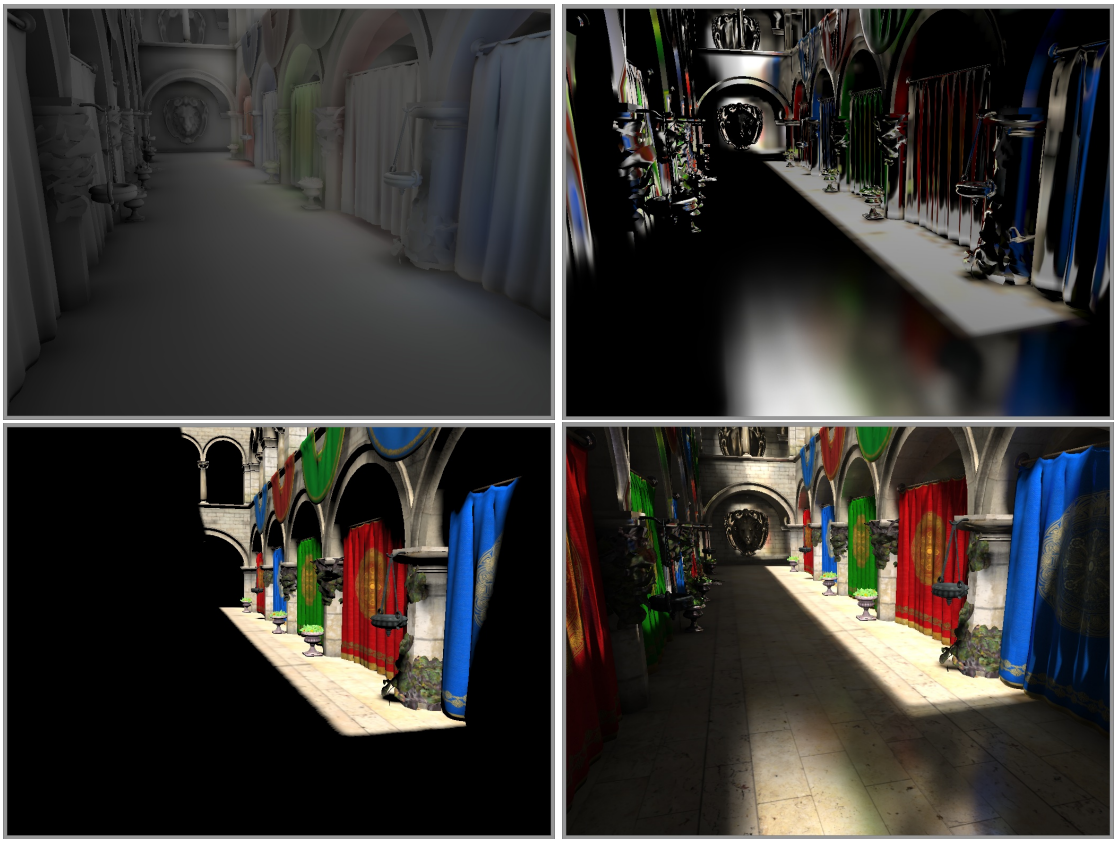


Figure 61: Indirect diffuse lighting (upper left), indirect specular lighting (upper right) and final result (lower right) obtained by combining direct (lower left) and indirect illumination.

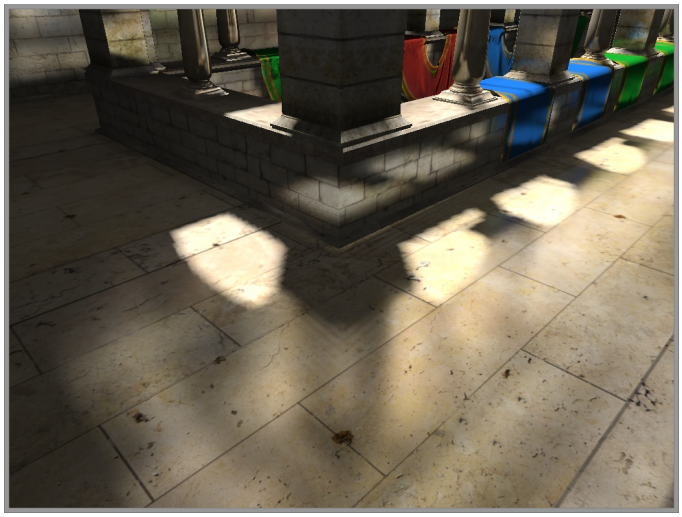


Figure 62: Banding on the Voxel Cone Tracing result

Figure 63: Doubling the number of samples during the voxel cone tracing pass has removed visible banding.

Another problem is that since voxels only store color, when tracing cones it is possible to accumulate some lighting contributions from objects that are occluded from the view point, thus leading to light leaking. To reduce these artifacts, anisotropic voxels are required.

Some approaches use four normal values, one for each face of a tetrahedron, others use six, one for each face of a cube, resulting in five more voxel grids, in order to be able to filter the irradiance anisotropically during the mipmap pass. In here we are going to follow Crassin et al. (2011) approach and use six directional voxel grids.

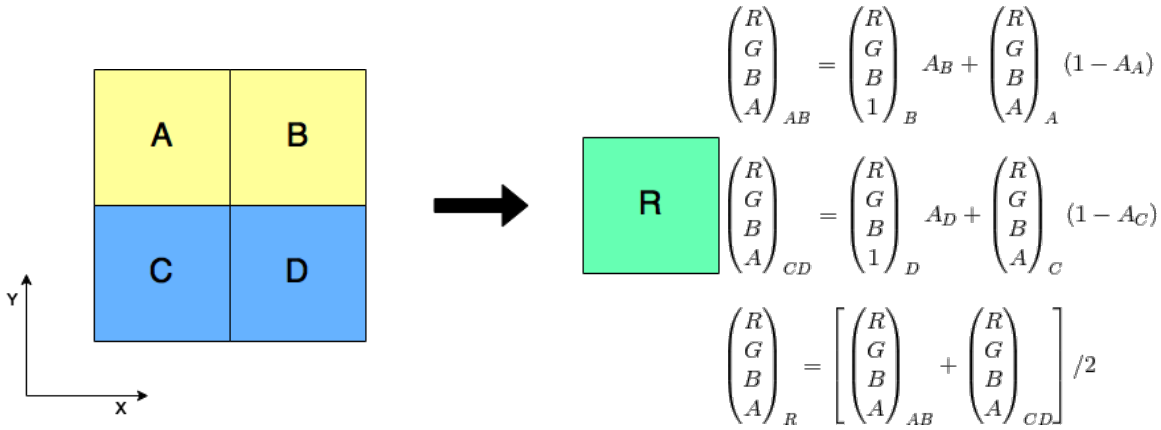


Figure 64: Example of anisotropic mipmapping of voxels in the X direction.

The mipmap pass is altered in order to perform alpha blending in one of the six directions, followed by the averaging of the resulting four values and the result is stored in the corresponding voxel grid (Figure 64).

Then in the cone tracing pass, instead of sampling only one grid, three samples are taken from three of the six directional 3D textures and weighted by the cone direction Listing 4.15.

```
vec4 voxelFetch(vec3 voxelPos, float sampleLOD, vec3 dir)
{
  vec4 sampleX = dir.x < 0.0 ? textureLod(voxelGrid[0], voxelPos, sampleLOD)
       : textureLod(voxelGrid[1], voxelPos, sampleLOD);
  vec4 sampleY = dir.y < 0.0 ? textureLod(voxelGrid[2], voxelPos, sampleLOD)
       : textureLod(voxelGrid[3], voxelPos, sampleLOD);
  vec4 sampleZ = dir.z < 0.0 ? textureLod(voxelGrid[4], voxelPos, sampleLOD)
       : textureLod(voxelGrid[5], voxelPos, sampleLOD);

  vec3 sampleWeights = abs(dir);
  float invSampleMag = 1.0 / (sampleWeights.x + sampleWeights.y +
     sampleWeights.z);
  sampleWeights *= invSampleMag;

  vec4 filtered = (sampleX * sampleWeights.x) + (sampleY * sampleWeights.y)
     + (sampleZ * sampleWeights.z);

  return filtered;
}
```

Listing 4.15: Anisotropic sampling of the voxel grid

Although the light leaking is greatly reduced (Figure 65), this approach has some drawbacks: the memory consumption of the voxel grid is increased (by at least 1.5 times) and the cone tracing pass has to do more texture calls (three per sample instead of just one), thus increasing the rendering time.

Another problem is that the glossy reflections are only single bounce. This means that the specular reflections do not take into account multiple reflections and only directly lit surfaces are shown. In order to view every object in the scene, not just the ones directly lit, it is possible to add some ambient lighting to the scene during voxelization. Nevertheless, multiple reflections are still ignored (Figure 66).

Figure 65: Anisotropic voxels reduce greatly light leaking in the scene.

Figure 66: Adding some ambient lighting during the light injection pass allows to capture some details omitted previously.

4.2.2 *Voxel Cone Tracing with a Sparse Voxel Octree*

This algorithm extends the previously described algorithm by using a different data structure to store the voxel grid. Instead of a full 3D texture, a sparse voxel octree is created that allows to collapse empty voxels in order to reduce the memory usage. However, the reduced memory consumption comes at the cost of an extra step: the creation of the sparse voxel octree structure. Also, the octree has to be traversed before retrieving the desired information stored in the voxels.

The steps performed by the algorithm are very similar to the previously described approach (4.2.1). However, some of these passes are different, due to the change of the data structure storing the voxel grid.

The algorithm starts by voxelizing the scene, using exactly the same process described in section 4.2.1. At the end of this pass, a fragment list containing voxel fragments is available (Listing 4.4), with the information needed for the light injection pass and for the creation of the sparse voxel octree.

Next, the sparse voxel octree structure is created by using the world positions stored in the fragment list previously created, writing the material colors and normals into the leaves of the octree.

After the leaves of the octree are filled with the voxel data, they are mipmapped through the octree structure until the root node is reached, generating data at multiple resolutions.

A Reflective shadow map is created in order to compute irradiance and store the result into the leaves of the octree structure. Once again, the data is mipmapped in the octree.

Finally, multiple cones are launched in the scene to gather indirect illumination. The process is very similar to the one detailed in section 4.2.1. The main difference is that the octree structure must be traversed to retrieve the data, instead of directly accessing the required mipmap level. As previously, the indirect illumination is added to the direct lighting to generate approximate illumination for the scene.

The algorithm is then subdivided in five different steps:

1. Voxelization
2. Sparse Voxel Octree Creation
3. Mipmapping
4. Light Injection
5. Voxel Cone Tracing

Sparse Voxel Octree Creation

The sparse voxel octree structure is a data structure composed by two components: the node pool and the brick pool.

Each node from the sparse voxel octree stores multiple pointers to access data at different levels during the traversal of the sparse octree structure (Listing 4.16). More precisely, it stores the address to its eight children (which are grouped in node tiles, allowing to access each of the eight children with a single address to the first child), the address of its corresponding brick in the brick pool, as well as the three x, y, z neighbor nodes which will come in handy during the light injection and mipmapping passes (3.1.3).

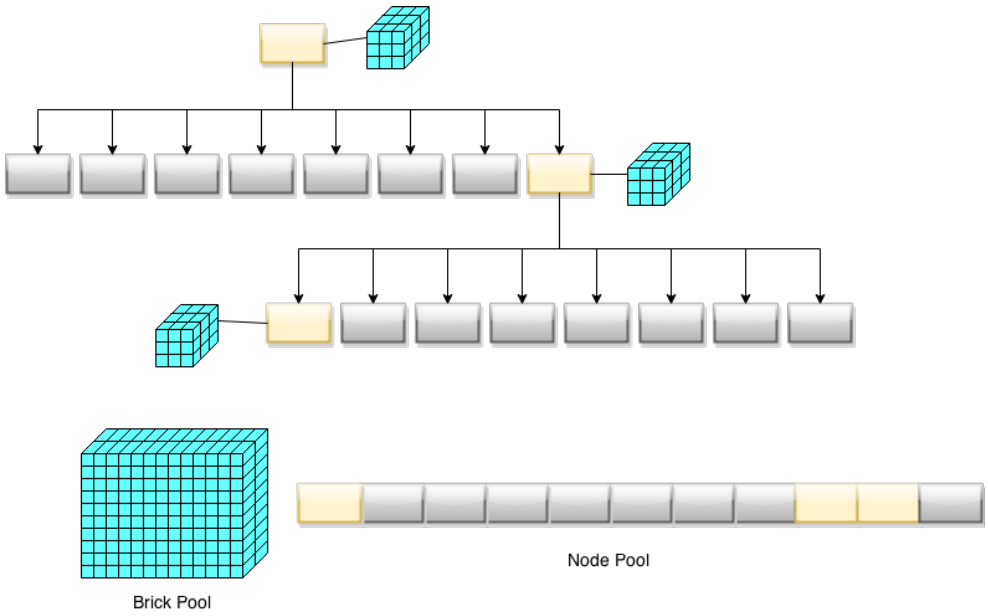


Figure 67: Octree and Octree Pools.

The brick pool is a 3D texture with an RGBA8 data format, storing the voxel data in bricks composed by 3^3 voxels in order to allow to use hardware trilinear filtering when sampling the voxels during the cone tracing pass (Figure 26). Each brick stores voxel data corresponding to each node tile of the sparse voxel octree.

```
struct OctNode
{
  uint nodePtr; // pointer to array of children
  uint brickPtr;

  uint neighborX;
  uint neighborY;
  uint neighborZ;
};

layout(binding = 1, std430) buffer Octree
{
  OctNode octree[];
};
```

Listing 4.16: Sparse Voxel Octree Structure

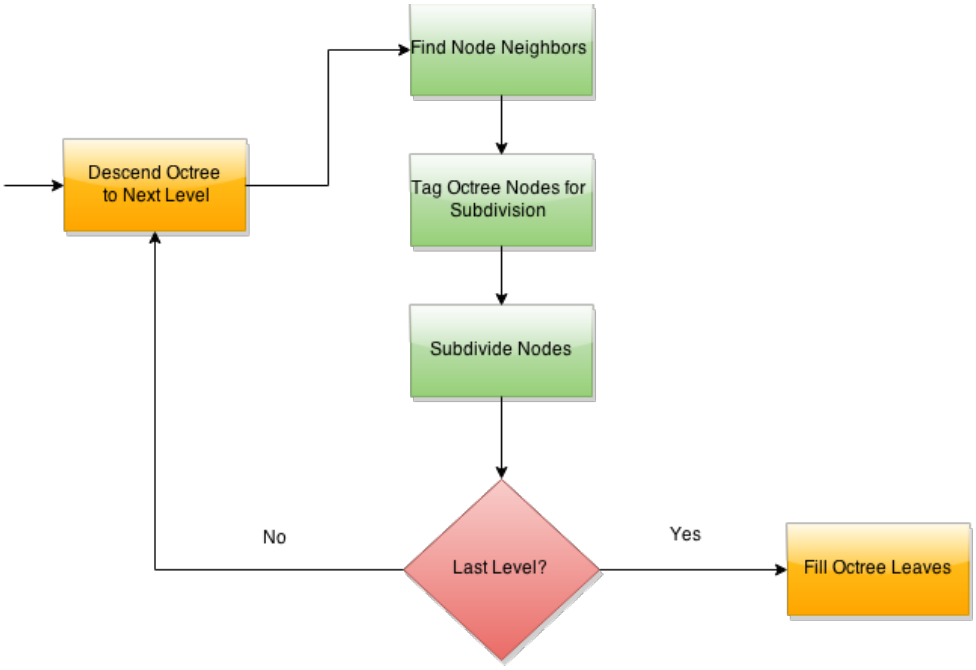


Figure 68: Octree Subdivision.

The creation of the octree is performed in multiple passes using attribute-less rendering. Level by level, starting from the root node, the octree is subdivided until the leaves are reached and the voxel fragments are written into their corresponding voxel bricks (Figure 68). The subdivision of the octree is performed in three passes:

1. Neighbors finding
2. Octree tagging
3. Octree subdivision

The first pass of the octree creation is in charge of finding the three x, y, z neighbors of a voxel and storing their addresses in the octree node. To do that, one thread must be launched for each fragment in the voxel fragment list.

Similarly to 4.2.1, the total number of fragments is stored in the fragment counter at the end of the voxelization pass. However, different passes of the octree creation might need a different number of threads to be launched. To avoid losing the information of the total number of fragments present in the fragment list, a separate buffer is bound to the atomic counter during the voxelization pass (instead of binding directly the atomic counter to the *count* field of the indirect draw structure). Another possibility would be to extend the indirect draw call structure to store the total number of fragments in another field appended at the end of the structure.

Then, similarly to Crassin and Green (2012), a draw call is issued with a single thread and using attribute-less rendering in order to modify the indirect draw structure parameters. For this pass, the thread simply reads the atomic fragment counter value and writes it in the *count* field of the indirect draw call structure (Listing 4.21).

Now that the indirect draw call structure has been updated, an indirect draw call is issued in order to find the neighbors of each node in the octree. Each thread reads the world position from the voxel fragment list and traverses the octree using the kd-restart algorithm (Listing 4.17).

```
vec3 convXYZToVec3(uint val)
{
  return vec3(float((val & 0x3FF00000) >> 20U), float((val & 0x000FFC00) >>
     10U), float(val & 0x000003FF));
}

(...)

vec3 pos = convXYZToVec3(fragmentList[gl_VertexID].position);

uint nodeAddress = 0;
vec3 boxMin = vec3(0);
vec3 boxDim = vec3(voxelResolution);
int depth = 1;

// Traverse octree until undivided node is found
while (octree[nodeAddress].nodePtr != NULL_NODE && octree[nodeAddress].
    nodePtr != SUBDIVIDE_NODE)
{
  boxDim *= 0.5;
  vec3 childIndex = step(boxMin + boxDim, fragmentPos);
  nodeAddress = octree[nodeAddress].nodePtr + uint(dot(childIndex, vec3(1,
     2, 4))); // get next pos and sum corresponding offset
  boxMin += (boxDim * childIndex);
  depth++;
}
```

Listing 4.17: Sparse voxel octree traversal

The traversal starts from the root node, computing the volume dimensions for the current node and comparing it to the world position retrieved from the voxel fragment list in order to find out to which child the traversal must be continued. Traversal stops when an empty node is found.

Now that the current leaf node is found, the world position is increased in each axis separately by the size of the current node's volume and the traversal is restarted with each neighbors world position until the same depth is reached (Listing 4.18). The neighbor node address is then written into the corresponding field in the octree node structure.

```
if(pos.x + boxDim.x < voxelResolution)
    octree[nodeAddress].neighborX = findNeighbors(pos + vec3(boxDim.x, 0, 0)
        , depth);

  if(pos.y + boxDim.y < voxelResolution)
    octree[nodeAddress].neighborY = findNeighbors(pos + vec3(0, boxDim.y, 0)
        , depth);

  if(pos.z + boxDim.z < voxelResolution)
    octree[nodeAddress].neighborZ = findNeighbors(pos + vec3(0, 0, boxDim.z)
        , depth);
```

Listing 4.18: Increasing the fragment's position by the node's volume in order to find the neighbors

The second pass tags the octree nodes in order to distinguish which ones should be subdivided. To do that a draw call is issued, launching one thread for each entry in the voxel fragment list (the indirect draw buffer does not need to be updated). Each thread in the vertex shader accesses its corresponding world position in the voxel fragment list and uses it to traverse the octree until an empty or tagged node is found (Listing 4.17). The node pointer of the octree node is then tagged in order to mark the node for subdivision (Listing 4.19).

```
if(depth == maxDepth)
    // Mark node as leaf
    octree[nodeAddress].nodePtr = LEAF_NODE;
else
    // Mark node for subdivision
    octree[nodeAddress].nodePtr = SUBDIVIDE_NODE;
```

Listing 4.19: Octree tagging

Since the octree is sparse, only some nodes have been marked for subdivision and only those nodes have to subdivided. In order to be able to launch one thread per node on each level of the octree, the nodes per level have to be kept on each step of the octree subdivision. The draw indirect structure has been extended in order to store an array of unsigned integers corresponding to the number of nodes on each level of the octree (Listing 4.20).

```
layout (std430, binding = 0) buffer IndirectBuffer
{
    uint count;
    uint primCount;
    uint first;
    uint baseInstance;
    uint nodesPerLevel[];
} DrawArraysCommand;
```

Listing 4.20: Indirect draw structure extended to store the nodes for each level of the octree

So, before launching the octree subdivision pass, a draw call with one single thread is issued for the purpose of altering the indirect draw structure parameters so that the next indirect draw call will start from the address of the first node on the current level and with a number of threads equal to to the number of nodes in the current octree level.

```
layout(binding = 1, offset = 0) uniform atomic_uint fragmentCounter;
layout(binding = 2, offset = 0) uniform atomic_uint nextNodeAddress;

// Indirection buffer
layout (std430, binding = 3) buffer IndirectBuffer
{
    uint count;
    uint primCount;
    uint first;
    uint baseInstance;
  uint nodesPerLevel[];
} DrawArraysCommand;

uniform int index;

// Update parameters of the indirect draw structure
void main()
{
    // One thread per entry on the fragment list
  if(index == 0)
  {
    DrawArraysCommand.count = atomicCounter(fragmentCounter);
    DrawArraysCommand.primCount = 1;
    DrawArraysCommand.first = 0;
    DrawArraysCommand.baseInstance = 0;
```

```
  }
  // Compute number of nodes on current level (index)
  // Update count and store in the nodesPerLevel array
  if(index > 0)
  {
    uint offset = 0;
    for(int i = 0; i < index-1; i++)
    {
      offset += DrawArraysCommand.nodesPerLevel[i];
    }

    DrawArraysCommand.first = offset;
    DrawArraysCommand.count = ((atomicCounter(nextNodeAddress) * 8) + 1) -
       offset;
    DrawArraysCommand.primCount = 1;
    DrawArraysCommand.baseInstance = 0;
    DrawArraysCommand.nodesPerLevel[index-1] = DrawArraysCommand.count;
  }
  // One thread per node on the current level (abs(index))
  if(index < 0)
  {
    int newIndex = abs(index);
    uint offset = 0;
    for(int i = 0; i < newIndex-1; i++)
    {
      offset += DrawArraysCommand.nodesPerLevel[i];
    }

    DrawArraysCommand.first = offset;
    DrawArraysCommand.count = DrawArraysCommand.nodesPerLevel[newIndex-1];
    DrawArraysCommand.primCount = 1;
    DrawArraysCommand.baseInstance = 0;
  }
}
```

Listing 4.21: Updating the indirect draw structure

Now that the indirect draw structure has the correct values, an indirect draw call is launched with the purpose of subdividing the nodes of the octree in the current level. Each thread checks the node pointer to its children and if the node has been marked for subdivision, an address is computed with the help of an atomic counter. The value returned by this atomic counter is also

used in order to compute the address to a brick in the brick pool, and the values are written into the octree node. In case the node currently being subdivided belongs to the last level of the octree, the atomic counter is only used to compute a brick address, since leaf nodes do not have children.

```
uint convVec3ToXYZ(vec3 val)
{
  return (uint(val.x) & 0x000003FF) << 20U | (uint(val.y) & 0x000003FF) <<
     10U | (uint(val.z) & 0x000003FF);
}

void main()
{
    if (octree[gl_VertexID].nodePtr == SUBDIVIDE_NODE)
    {
        // Retrieve and update the address of the next free memory slot to
           store the child nodes and the bricks
        uint numNodes = atomicCounterIncrement(nextNodeAddress);
        uint childNodesOffset = 1 + numNodes * 8;

        uint brickOffset = numNodes;
        uint zDirection = brickOffset % numBricks;
        uint yDirection = (brickOffset / numBricks) % numBricks;
        uint xDirection = brickOffset / (numBricks * numBricks);

        uint brickAddr = convVec3ToXYZ(vec3(xDirection, yDirection,
           zDirection));

        // Subdivide the current node by writing the child nodes and bricks
           address
        octree[gl_VertexID].nodePtr = childNodesOffset;
        octree[gl_VertexID].brickPtr = brickAddr;
    }

  if (octree[gl_VertexID].nodePtr == LEAF_NODE)
    {
        uint brickOffset = atomicCounterIncrement(nextNodeAddress);
        uint zDirection = brickOffset % numBricks;
        uint yDirection = (brickOffset / numBricks) % numBricks;
        uint xDirection = brickOffset / (numBricks * numBricks);

        uint brickAddr = convVec3ToXYZ(vec3(xDirection, yDirection,
           zDirection));
```

```
        octree[gl_VertexID].brickPtr = brickAddr;
    }
}
```

Listing 4.22: Octree subdivision

Now that the sparse voxel octree structure has been created, the bricks must be filled. The voxel fragment list contains normal and material information for each fragment that need to be inserted into the leaves of the octree and then mipmapped into the upper levels.

An indirect draw call is issued with a number of threads equal to the number of fragments in the voxel fragment list (the indirect draw structure is altered in the same way as before). Each thread retrieves world position from its corresponding entry in the voxel fragment list and uses it to traverse the octree until the last level is reached (Listing 4.17). The brick address is then retrieved and used to store color and normal information from the voxel fragment into the color and normal brick pools, respectively, using an RGBA8 image atomic average operation (Listing 4.9). Since each brick contains 3^3 voxels that represent 2^2 octree nodes (and some information from their neighbors, in order to use hardware trilinear filtering), the voxels are actually stored and averaged into the corner voxels of the brick (Listing 4.23).

```
// Get brick address
uint brickAddr = octree[nodeAddress].brickPtr;
// Compute address of node in the brick
boxDim *= 0.5f;
childIndex = step(boxMin + boxDim, pos);
vec3 coords = (convXYZToVec3(brickAddr) * 3) + (childIndex * 2);  // Node
    corners

imageAtomicRGBA8Avg(brickPoolColor, ivec3(coords), convRGBA8ToVec4(voxel.
    color));
imageAtomicRGBA8Avg(brickPoolNormal, ivec3(coords), convRGBA8ToVec4(voxel.
    normal));
```

Listing 4.23: Indirect draw structure extended to store the nodes for each level of the octree

In a following pass, one thread is launched for each of the leaf nodes using an indirect draw call. Each thread retrieves the brick address from the node and samples the corner voxels. The occlusion (stored in the alpha channel) is then corrected since the RGBA8 image atomic average uses the alpha value and the corner values are spread through the whole voxel brick (Figure 69).

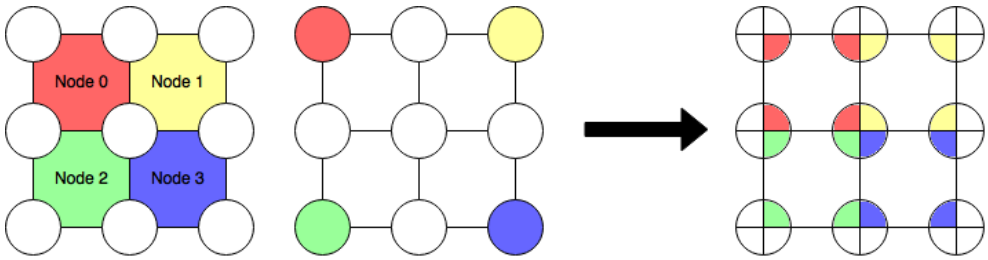


Figure 69: In the leaves of the octree, values are stored in the brick corners and then spread through the whole brick in a following pass.

Now the only thing missing from the bricks in leaf nodes of the octree is the neighbor information. The neighbor transfer consists of three passes, one on each direction (x, y, z) and uses the neighbor addresses stored into the octree nodes in order to rapidly access the neighbor nodes in the octree. One thread per leaf node is issued and each thread samples the node address and one of the neighbors of its corresponding leaf, checking if either of them is empty. If both of the leaf nodes exist, the brick address and neighbor brick address are retrieved and used to average voxels in the direction desired (Figure 34). The average value is then stored in both bricks.

Mipmapping

Since the octree stores its values in bricks, which repeat voxels between neighboring bricks in order to allow using hardware trilinear filtering when sampling them, the mipmapping needs to be done manually. The mipmapping of the sparse voxel octree is done level by level, in **n-1** steps for an octree with **n** levels. On each step, multiple passes are performed (Figure 70).

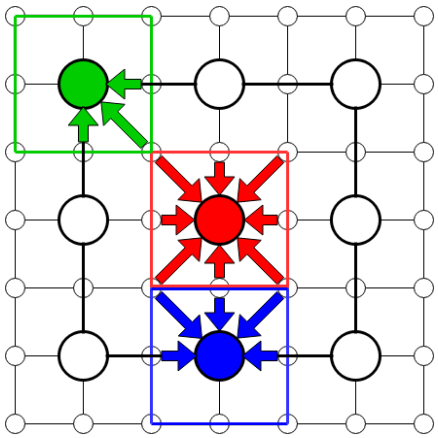


Figure 71: Partial result computed using the mipmapping process.

First of all, the indirect draw call parameters are altered using the number of nodes stored previously at the end of the indirect draw call structure, in order to launch indirect draw calls with a number of threads matching the number of nodes on the current level (Listing 4.21).

Each pass samples the corresponding octree node, retrieving the address to the brick and its children. The address of the children nodes is then used to retrieve the brick address of each child in order to sample the higher resolution bricks. However, some information could be missing from the children's bricks. In fact, only the center voxel has access to all the voxels it needs to compute the averaged

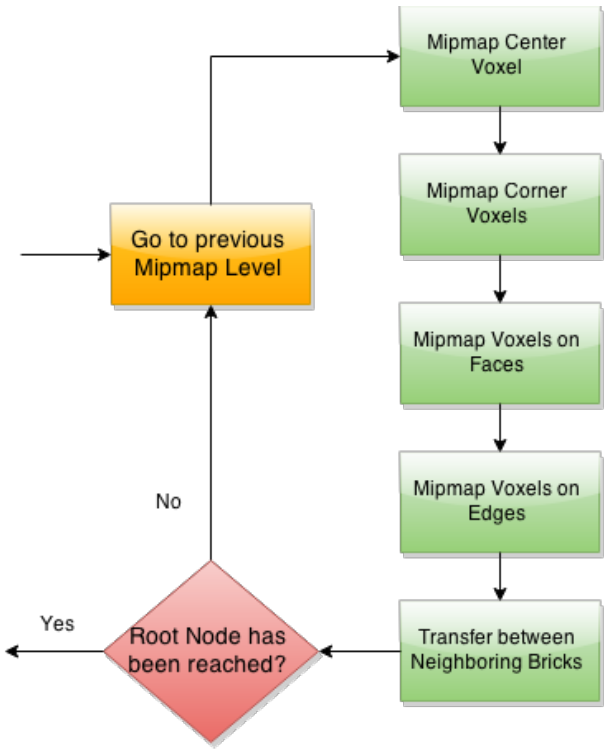


Figure 70: Octree Mipmapping.

voxel. For the rest of the voxels, some information has to come from neighboring bricks in order to complete the result.

The solution to this problem is to compute only a partial averaged value using the children bricks, and then complete the result using the same neighbor transfer scheme described previously to complete the result on the leaves of the sparse voxel octree (Figure 71). Corners, edges, sides and center voxels are mipmapped in separate passes, since they need information about a different number of voxels from the higher resolution bricks. Also, because voxels sampled from the children bricks reappear in neighboring bricks, the sampled values have to be weighted in accordance to their multiplicity in order to generate a correct result (3.1.2).

Light Injection

Since the color, opacity and normal of the fragments have already been averaged into two brick pools, there is no need to compute the irradiance directly from the fragments and storing them in the voxel grid.

Instead, a reflective shadow map is generated (section 2.2) in order to splat the direct lighting in the voxel grid. To compute the RSM a full-screen quad with a viewport corresponding to

its size is rendered. The RSM stores the world positions of the fragments seen from the light's perspective (Figure 72).

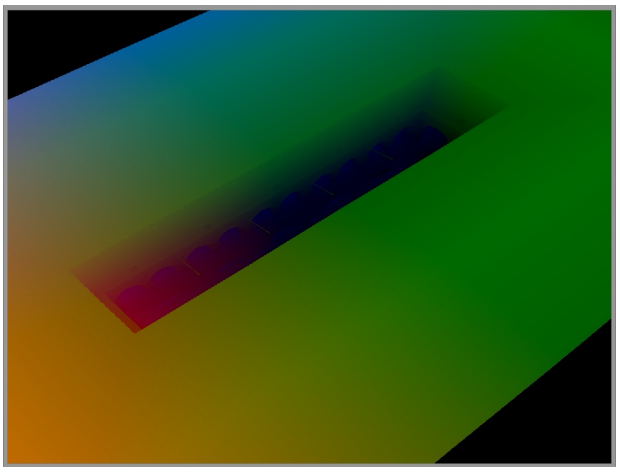


Figure 72: Reflective shadow map storing world positions.

The world position is sampled in the fragment shader and converted to voxel grid coordinates and the sparse voxel octree is then traversed until the leaf node is encountered (Listing 4.17), retrieving the address of one of the four corners from the corresponding brick in the brick pool (according to the node in which lighting needs to be injected).

Now that the brick address where the light has to be injected has been found, the averaged color and normal can be sampled from their corresponding brick pools and used to compute the irradiance using the Lambertian reflection model, storing the result in the irradiance brick pool (Listing 4.24).

```
// Get brick address
uint brickAddr = octree[nodeAddress].brickPtr;
// Compute address of node in the brick
boxDim *= 0.5f;
childIndex = step(boxMin + boxDim, pos);
vec3 injectionCoords = (convXYZToVec3(brickAddr) * 3) + (childIndex * 2); //
    Node corners

vec3 voxelNormal = texelFetch(brickPoolNormal, ivec3(injectionCoords), 0).
   xyz * 2.0 - 1.0; // unpack normal
vec4 voxelColor = texelFetch(brickPoolColor, ivec3(injectionCoords), 0);

vec3 L = normalize(lightDirection); // Directional Light
vec3 N = normalize(voxelNormal);
float NdotL = max(dot(N, L), 0.0);
```

```
vec4 incomingRadiance = vec4(voxelColor.rgb * NdotL * lightColor.rgb, 1.0);

imageAtomicRGBA8Avg(brickPoolIrradiance, ivec3(injectionCoords),
    incomingRadiance);
```

Listing 4.24: Injecting direct lighting into the brick cornes of the bricks in the octree leaves

After filling the leaf nodes of the sparse voxel octree with the direct lighting information, the irradiance brick pool is completed with the lower mipmap levels by performing a mipmap pass in the same way as before for the color and normal brick pools.

Voxel Cone Tracing

The voxel cone tracing pass is essentially the same described in Section 4.2.1. The difference lies in the way voxels are sampled during the traversal of the cone.

For the full voxel grid, the 3D texture was sampled using hardware quadrilinear filtering, using simply the voxel grid coordinate and the textureLOD.

Now, the sparse voxel octree has to be traversed up to the desired mipmap level, requiring as many samples as the depth of the level, retrieving the brick address in order to perform sampling of the desired voxel in the brick pool. Since voxels are stored in bricks, it is not possible to simply use the quadrilinear filtering offered by the hardware. To achieve quadrilinear filtering, hardware trilinear filtering is used within the brick, and the filtering between mipmap levels is performed manually (Listing 4.25).

```
vec4 voxelFetch(vec3 voxelPos, float sampleLOD)
{
  uint nodeAddress = 0;
  vec3 boxMin = vec3(0.0);
    vec3 boxDim = vec3(voxelResolution);
  vec3 boxMin2 = vec3(0.0);

  vec3 childIndex = vec3(0.0);
  int depth = 1;

  // Traverse octree until undivided node is found
  uint nodePointer = octree[nodeAddress].nodePtr;
  uint previousNodeAddress = 0;
  while (nodePointer != NULL_NODE && nodePointer != LEAF_NODE && depth <
     sampleLOD)
```

```
{
  previousNodeAddress = nodeAddress;
  boxDim *= 0.5;
  childIndex = step(boxMin + boxDim, voxelPos);
  // get next pos and sum corresponding leaf offset
  nodeAddress = octree[nodeAddress].nodePtr + uint(dot(childIndex, vec3
      (1.0, 2.0, 4.0)));
  // store data from previous level for quadrilinear
  boxMin2 = boxMin;
  boxMin += (boxDim * childIndex);
  nodePointer = octree[nodeAddress].nodePtr;
  depth++;
}

if(nodePointer == NULL_NODE) return vec4(0);

// Compute sampling address from previous mip level
uint brickAddr1 = octree[previousNodeAddress].brickPtr;
vec3 previousChildIndex = step(boxMin2 + boxDim, voxelPos);
// Offset for trilinear filtering
vec3 previousOffset = vec3(0.5) + (previousChildIndex + fract(voxelPos/
    boxDim));
vec3 previousCoords = (convXYZToVec3(brickAddr1) * 3) + previousOffset;
ivec3 size = textureSize(brickPoolColor, 0);
previousCoords /= size;
// retrieve irradiance
vec4 irradiance1 = texture(brickPoolIrradiance, previousCoords);
// retrieve color and occlusion
vec4 color1 = texture(brickPoolColor, previousCoords);

// Compute sampling address from current mip level
uint brickAddr0 = octree[nodeAddress].brickPtr;
boxDim *= 0.5;
childIndex = step(boxMin + boxDim, voxelPos);
// Offset for trilinear filtering
vec3 offset = vec3(0.5) + childIndex + fract(voxelPos/boxDim);
vec3 coords = (convXYZToVec3(brickAddr0) * 3) + offset;
//ivec3 size = textureSize(brickPoolColor, 0);
coords /= size;
// retrieve irradiance
vec4 irradiance0 = texture(brickPoolIrradiance, coords);
```

```
  // retrieve color and occlusion
  vec4 color0 = texture(brickPoolColor, coords);

  // perform quadrilinear filtering according to LOD value
  float f = fract(sampleLOD); // get the decimal part
  vec4 result = vec4(mix(irradiance0.rgb, irradiance1.rgb, vec3(f)), mix(
      color0.a, color1.a, f));

  return result;
}
```

Listing 4.25: Fetching the voxel data from the sparse voxel octree for the cone tracing pass and manual quadrilinear filtering

4.3 GLOBAL ILLUMINATION USING VIRTUAL POINT LIGHTS

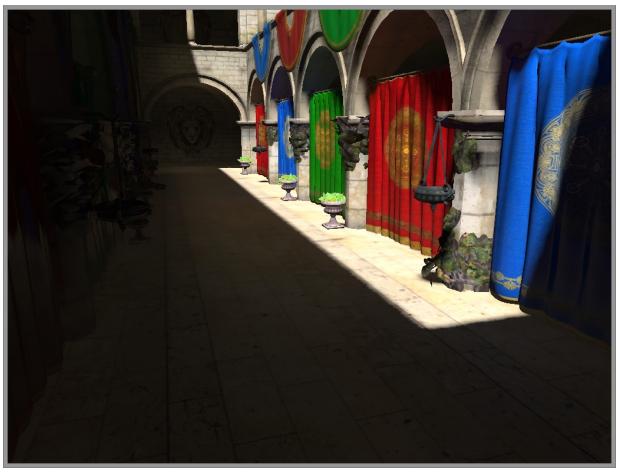


Figure 73: Result obtained from applying the VPLs for indirect diffuse lighting and Voxel Cone Tracing for indirect specular lighting

This algorithm uses Virtual Point Lights (VPL) stored and propagated in a Light Propagation Volume (LPV) to approximate diffuse indirect illumination in the scene (Figure 73). The glossy reflections in the scene are computed by using Voxel Cone Tracing (3.1.3) to trace a cone in the reflected direction and gathering samples through a hierarchical representation of the scene.

As mentioned before (section 3.4) the algorithm uses two grids for indirect diffuse lighting. Both grids share the same resolution voxel wise (the authors used 32^3), but one occupies 8 times

more volume. Another grid with a higher resolution is used for the specular reflections and covers the same frustum as the finer diffuse grid.

The grids move along with the camera, defining a volume around it. For the grids to move along with the camera without introducing artifacts, some care must be taken during the voxelization pass.

Since each voxel represents a small volume in space, moving the frustum by an amount not multiple of the voxel size may cause different fragments to be joined together, causing flickering each time the camera moves. To avoid this problem, the frustum must be adjusted so that it moves only the size of a voxel at a time.

Initially, the algorithm starts by voxelizing the scene once for each voxel grid, storing material color and occlusion in the voxel grids. The smaller grids used for diffuse lighting also store the normals.

The diffuse grids are then used together with a shadow map to compute VPLs storing the reflected radiance of the voxels, the result being encoded in three textures (one for each color channel) using spherical harmonics (SH) coefficients. These VPLs are then propagated through the grid in order to distribute lighting across the scene.

The reflection grid, to allow the computation of specular lighting, is a higher resolution grid (256^3) that covers the same frustum as the smallest grid of the indirect diffuse illumination. It is created by sampling the finer indirect diffuse textures containing the VPLs and fetching the intensity value in the finer mipmap level. The reflection grid is then mipmapped in order to fill the coarser mipmap levels.

Finally, indirect diffuse lighting is obtained by retrieving the values from the voxel grids, according to the normals of the object surfaces, and glossy reflections are computed by tracing a cone in the reflected direction.

Thus, the algorithm can be subdivided into 5 distinct steps:

1. Voxelization
2. Direct Light Injection
3. Direct Light Propagation
4. Reflection Grid Creation and Mipmapping
5. Global Illumination Rendering

4.3.1 *Voxelization*

For this implementation, two grids are used for diffuse indirect illumination. They both have the same size (32^3), but one covers a frustum 8 times larger.

The grids are created as a linear buffer storing an array of structures. These structures are defined as an unsigned integer that encodes the color and occlusion of that voxel, and a vector of four elements that encodes the normal information (Listing 4.26).

```
struct Voxel
{
  uint colorOcclusionMask;
  uvec4 normalMasks;
};
```

Listing 4.26: struct definition of a voxel in the voxel grid

Another higher resolution grid (256^3) that shares the frustum with the smallest grid of the indirect diffuse illumination is created. It is defined as a 3D texture that only stores a single unsigned integer encoding color and occlusion and it will be used to create the reflection grid in a subsequent pass.

The voxelization pass is actually very similar to the one previously explained in section 4.2. It is performed once for each grid (low resolution fine and coarse grids, and higher resolution grid).

As before, depth testing, the color mask and the depth mask are disabled, and an orthographic projection is used. The difference now is that the frustum of the orthographic projection is different between the fine/high resolution grids and the coarse grid.

The geometry shader, using conservative rasterization, is the same detailed in section 4.2.1.

The main difference is in the fragment shader. The RGB material color is retrieved and encoded in linear space into the last 24 bits of an unsigned integer. The higher 7 bits are used to encode a contrast value computed from the retrieved color in linear space, and a single bit records the occlusion. Since the values are written into the buffer/3D texture using atomicMax/imageAtomicMax operations, colors with a higher contrast value will automatically dominate Listing 4.27.

```
// get surface color
vec3 base = texture(tex, TexCoords).rgb;

// encode color in linear space into unsigned integer
vec3 baseLinear = diffuse.rgb / 3.14159265359 * pow(base, vec3(GAMMA));
```

```
uvec3 iColor = uvec3(baseLinear * 255.0f);
uint colorOcclusionMask = (iColor.r << 16U) | (iColor.g << 8U) | iColor.b;

// By writing the contrast value (0-255) in the highest 8 bit of
// the color-mask, automatically colors with high contrast will
// dominate, since we write the results with an atomicMax into
// the voxel-grids. The contrast value is calculated in SRGB space.
float contrast = length(base.rrg - base.gbb) / (sqrt(2.0f) + base.r + base.g
    + base.b);
uint iContrast = uint(contrast * 127.0);
colorOcclusionMask |= iContrast << 24U;

// encode occlusion into highest bit
colorOcclusionMask |= 1 << 31U;
```

Listing 4.27: Encoding the color, occlusion and contrast values in an unsigned int

For the diffuse lighting grids, normal information also needs to be computed, encoded and written in the buffer. Since fragments in the same voxel can have opposite normals, special care has to be given when writing normal values.

First, the normal is encoded into an unsigned integer. 9 bits are used for each channel of the normal (8 bit for the channel value, 1 bit for the sign) and the remaining 5 bits are used to encode the dot product between the normal and a face of a tetrahedron (Listing 4.28).

```
// encode normal into lowest 27 bits
// 9 bits per channel (8 for value, 1 for sign)
ivec3 iNormal = ivec3(normal*255.0f);
uvec3 iNormalSigns;
iNormalSigns.x = (iNormal.x>>5) & 0x04000000;
iNormalSigns.y = (iNormal.y>>14) & 0x00020000;
iNormalSigns.z = (iNormal.z>>23) & 0x00000100;
iNormal = abs(iNormal);
uint normalMask = iNormalSigns.x | (iNormal.x<<18) | iNormalSigns.y | (
   iNormal.y<<9) | iNormalSigns.z | iNormal.z;

// normalized directions of 4 faces of a regular tetrahedron
mat4x3 faceMatrix;
faceMatrix[0] = vec3(0.0f, -0.57735026f, 0.81649661f);
faceMatrix[1] = vec3(0.0f, -0.57735026f, -0.81649661f);
faceMatrix[2] = vec3(-0.81649661f, 0.57735026f, 0.0f);
faceMatrix[3] = vec3(0.81649661f, 0.57735026f, 0.0f);
```

```
// Determine to which tetrahedron face the normal is closest to
vec4 dotProducts = normal * faceMatrix;
float maximum = max(max(dotProducts.x, dotProducts.y), max(dotProducts.z,
    dotProducts.w));
uint index;
if(maximum == dotProducts.x)
  index = 0;
else if(maximum == dotProducts.y)
  index = 1;
else if(maximum == dotProducts.z)
  index = 2;
else
  index = 3;

// By writing the corresponding dotProduct value in the highest
// 5 bit of the normal-mask, automatically the closest normal to
// the determined tetrahedron face will be selected, since we
// write the results with an atomicMax into the voxel-grids.
float dotProduct = dotProducts[index];

uint iDotProduct = uint(clamp(dotProduct, 0.0, 1.0) * 31.0f);
normalMask |= iDotProduct<<27U;
uint normalIndex = index;
```

Listing 4.28: Encoding the normal information in an unsigned integer

Finally, according to the normal index, the normal mask is written into the corresponding channel of the vector in the buffer using an atomicMax operation. Since the dot product between the normal and the face of the tetrahedron is encoded in the most relevant bits, the closest normal to the tetrahedron will automatically dominate (Listing 4.29).

```
// get index into the voxel-grid
int gridIndex = ((voxelPos.z * 1024) + (voxelPos.y * 32) + voxelPos.x);

// output color/ occlusion
atomicMax(gridBuffer[gridIndex].colorOcclusionMask, colorOcclusionMask);

// output normal according to normal index
if(normalIndex == 0)
  atomicMax(gridBuffer[gridIndex].normalMasks.x, normalMask);
```

```
if(normalIndex == 1)
  atomicMax(gridBuffer[gridIndex].normalMasks.y, normalMask);
if(normalIndex == 2)
  atomicMax(gridBuffer[gridIndex].normalMasks.z, normalMask);
if(normalIndex == 3)
  atomicMax(gridBuffer[gridIndex].normalMasks.w, normalMask);
```

Listing 4.29: Storing the color/occlusion and the normal information into the corresponding channel in the voxel grid

4.3.2 *Direct Light Injection*

Since lighting will be encoded as virtual point lights using a spherical harmonics representation, three buffers (one for each color channel) with the same size as the voxel grids (32^3) are needed. 3D textures are used with an RGBA16F data format to encode the SH coefficients in order to allow the use of hardware trilinear filtering when applying the indirect diffuse lighting to the scene.

For the light injection, $32 \times 32 \times 32$ threads need to be launched. Although Doghramachi (2013) used a fullscreen quad (with a viewport matching the grid side) and instanced rendering to generate the threads, the same result can be obtained by launching a compute shader with the required number of threads.

This is performed for the fine and coarse grid only, since the higher resolution grid will be used to inject lighting in the reflection grid using a different process, in a subsequent pass.

The corresponding voxel is retrieved from the attached buffer and the color and normal are decoded. The world-space position is computed using the voxel grid coordinate.

With this information, a shadow map (generated previously) is used to compute diffuse direct illumination depending on the type of the light (Listing 4.30).

```
// Voxel data
struct Voxel
{
  uint colorOcclusionMask;
  uvec4 normalMasks;
};

// Voxel Grid
layout(binding = 1, std430) buffer GridBuffer
```

```
{
  Voxel gridBuffer[];
};

// Decode specified mask into a float3 color (range 0.0f-1.0f).
vec3 DecodeColor(in uint colorMask)
{
  vec3 color;
  color.r = (colorMask>>16U) & 0x000000FF;
  color.g = (colorMask>>8U) & 0x000000FF;
  color.b = colorMask & 0x000000FF;
  color /= 255.0f;
  return color;
}

// Decode specified mask into a vec3 normal (normalized).
vec3 DecodeNormal(in uint normalMask)
{
  ivec3 iNormal;
  iNormal.x = int((normalMask>>18) & 0x000000FF);
  iNormal.y = int((normalMask>>9) & 0x000000FF);
  iNormal.z = int(normalMask & 0x000000FF);

  ivec3 iNormalSigns;
  iNormalSigns.x = int((normalMask>>25) & 0x00000002);
  iNormalSigns.y = int((normalMask>>16) & 0x00000002);
  iNormalSigns.z = int((normalMask>>7) & 0x00000002);
  iNormalSigns = 1-iNormalSigns;

  vec3 normal = vec3(iNormal)/255.0f;
  normal *= iNormalSigns;
  return normal;
}

(...)

// get index of current voxel
ivec3 voxelPos = ivec3(gl_GlobalInvocationID);
int gridIndex = ((voxelPos.z * 1024) + (voxelPos.y * 32) + voxelPos.x);

// get voxel data
```

```
Voxel voxel = gridBuffer[gridIndex];

// decode color of voxel
vec3 albedo = DecodeColor(voxel.colorOcclusionMask);

// get normal of voxel that is closest to the light-direction
float nDotL;
mat4x3 normalMatrix;
normalMatrix[0] = DecodeNormal(voxel.normalMasks.x);
normalMatrix[1] = DecodeNormal(voxel.normalMasks.y);
normalMatrix[2] = DecodeNormal(voxel.normalMasks.z);
normalMatrix[3] = DecodeNormal(voxel.normalMasks.w);
vec4 dotProducts = lightDir * normalMatrix;

float maximum = max(max(dotProducts.x, dotProducts.y), max(dotProducts.z,
    dotProducts.w));
uint index;
if(maximum == dotProducts.x)
  index = 0;
else if(maximum == dotProducts.y)
  index = 1;
else if(maximum == dotProducts.z)
  index = 2;
else
  index = 3;

nDotL = dotProducts[index];
vec3 normal = normalMatrix[index];

// compute shadowTerm by using shadowMap
float shadowTerm = ComputeShadowTerm(vec4(position, 1.0f));

// compute diffuse direct illumination
vec3 vDiffuse = albedo * lightColor * clamp(nDotL, 0.0f, 1.0f) *
    lightMultiplier * shadowTerm;
```

Listing 4.30: Direct lighting computation from the voxel grid

The last step is to encode the diffuse albedo into a virtual point light using a second order spherical harmonics representation and write each channel into the corresponding 3D texture.

For this, a clamped cosine lobe function oriented in the Z direction is encoded as spherical harmonics. Since it possesses rotational symmetry around the Z axis, the spherical harmonic projection results in zonal harmonics, which are simpler to rotate than general spherical harmonics. This way, each channel of the diffuse albedo is multiplied by zonal harmonics, rotated into the direction of the voxel normal and stored into the corresponding 3D texture using image store operations (Listing 4.31).

```
// Returns zonal harmonics, rotated into the specified direction
vec4 ClampedCosineCoeffs(in vec3 dir)
{
  vec4 coeffs;
  coeffs.x = 0.8862269262f;          // PI/(2*sqrt(PI))
  coeffs.y = -1.0233267079f;         // -((2.0f*PI)/3.0f)*sqrt(3/(4*PI))
  coeffs.z = 1.0233267079f;          // ((2.0f*PI)/3.0f)*sqrt(3/(4*PI))
  coeffs.w = -1.0233267079f;         // -((2.0f*PI)/3.0f)*sqrt(3/(4*PI))
  coeffs.wyz *= dir;
  return coeffs;
}

// turn illuminated voxel into virtual point light
// represented by second order spherical harmonics coeffs
vec4 coeffs = ClampedCosineCoeffs(normal);
vec4 redSHCoeffs = coeffs * vDiffuse.r;
vec4 greenSHCoeffs = coeffs * vDiffuse.g;
vec4 blueSHCoeffs = coeffs * vDiffuse.b;

// output red/ green/ blue SH-coeffs
imageStore(redSH, voxelPos, redSHCoeffs);
imageStore(greenSH, voxelPos, greenSHCoeffs);
imageStore(blueSH, voxelPos, blueSHCoeffs);
```

Listing 4.31: Converting illuminated voxel into VPL and storing in the corresponding texture

4.3.3 *Direct Light Propagation*

Now that the virtual point light have been injected into the grid using SH, the light must be propagated through the grid in order to add their contribution to neighbor voxels.

To do this, a compute shader is used, launching one thread per voxel in the grid. Each thread identifies its corresponding voxel in the voxel grid by querying the gl_GlobalInvocationID input variable.

The contributions from the six neighboring voxels are computed and added to the current voxel. In order to do this, the direction from the neighbor voxel to the face of the current cell and its corresponding solid angle are determined (Figure 74).

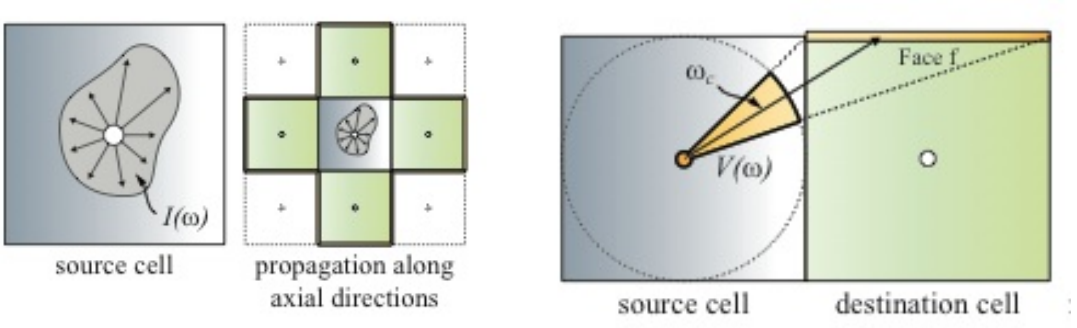


Figure 74: Illustration of the scattering approach for the VPL propagation. *Source:* Kaplanyan and Dachsbacher (2010)

The spherical harmonics coefficients for that direction are computed and used to retrieve the flux from the neighbor cell to the face of the current voxel by weighting them by the solid angle (Listing 4.32).

```
// solid angles (normalized), subtended by the face onto the neighbor cell
    center
#define SOLID_ANGLE_A 0.0318842778f // (22.95668f/(4*180.0f))
#define SOLID_ANGLE_B 0.0336955972f // (24.26083f/(4*180.0f))

// offsets to 6 neighbor cell centers
const ivec3 offsets[6] =
{
  ivec3(0, 0, 1),
  ivec3(1, 0, 0),
  ivec3(0, 0, -1),
  ivec3(-1, 0, 0),
  ivec3(0, 1, 0),
  ivec3(0, -1, 0)
};

// SH-coeffs for six faces
const vec4 faceCoeffs[6] =
```

```
{
  vec4(0.8862269521f, 0.0f, 1.0233267546f, 0.0f),  // ClampedCosineCoeffs(
      vec3(offsets[0]))
  vec4(0.8862269521f, 0.0f, 0.0f, -1.0233267546f), // ClampedCosineCoeffs(
      vec3(offsets[1]))
  vec4(0.8862269521f, 0.0f, -1.0233267546f, 0.0f), // ClampedCosineCoeffs(
      vec3(offsets[2]))
  vec4(0.8862269521f, 0.0f, 0.0f, 1.0233267546f),  // ClampedCosineCoeffs(
      vec3(offsets[3]))
  vec4(0.8862269521f, -1.0233267546f, 0.0f, 0.0f), // ClampedCosineCoeffs(
      vec3(offsets[4]))
  vec4(0.8862269521f, 1.0233267546, 0.0f, 0.0f)    // ClampedCosineCoeffs(
      vec3(offsets[5]))
};

(...)

// get grid-position of current cell
ivec3 elementPos = ivec3(gl_GlobalInvocationID);

// initialize SH-coeffs with values from current cell
vec4 sumRedSHCoeffs = texelFetch(redSH, elementPos, 0);
vec4 sumGreenSHCoeffs = texelFetch(greenSH, elementPos, 0);
vec4 sumBlueSHCoeffs = texelFetch(blueSH, elementPos, 0);

// for each neighboring cell
for(uint i = 0; i < 6; i++)
{
  // get grid-position of 6 neighbor cells
  ivec3 samplePos = elementPos + offsets[i];

  // continue, if cell out of bounds
  if((samplePos.x < 0) || (samplePos.x > 31) || (samplePos.y < 0) || (
      samplePos.y > 31) || (samplePos.z < 0) || (samplePos.z > 31))
    continue;

  // load SH-coeffs for neighbor cell
  vec4 redSHCoeffs = texelFetch(redSH, samplePos, 0);
  vec4 greenSHCoeffs = texelFetch(greenSH, samplePos, 0);
  vec4 blueSHCoeffs = texelFetch(blueSH, samplePos, 0);
```

```
  // for each face of the current cell
  for(uint j = 0; j < 6; j++)
  {
    // get direction from current neighbor cell center to face of current
       cell
    vec3 neighborCellCenter = vec3(offsets[i]);
    vec3 facePosition = vec3(offsets[j]) * 0.5f;
    vec3 dir = facePosition - neighborCellCenter;
    float fLength = length(dir);
    dir /= fLength;

    // get corresponding solid angle
    float solidAngle = 0.0f;
    if(fLength > 0.5f)
      solidAngle = (fLength>=1.5f) ? SOLID_ANGLE_A : SOLID_ANGLE_B;

    // get SH-coeffs for direction
    vec4 dirSH;
    dirSH.x = 0.2820947918f;          // 1/(2*sqrt(PI))
    dirSH.y = -0.4886025119f;         // -sqrt(3/(4*PI))*y
    dirSH.z = 0.4886025119f;          // sqrt(3/(4*PI))*z
    dirSH.w = -0.4886025119f;         // -sqrt(3/(4*PI))*x
    dirSH.wyz *= dir;

    // calculate flux from neigbor cell to face of current cell
    vec3 flux;
    flux.r = dot(redSHCoeffs, dirSH);
    flux.g = dot(greenSHCoeffs, dirSH);
    flux.b = dot(blueSHCoeffs, dirSH);
    flux = max(vec3(0), flux) * solidAngle * fluxAmplifier;

    // add contribution to SH-coeffs sums
    vec4 coeffs = faceCoeffs[j];
    sumRedSHCoeffs += coeffs * flux.r;
    sumGreenSHCoeffs += coeffs * flux.g;
    sumBlueSHCoeffs += coeffs * flux.b;
  }
}

// write out generated red/ green/ blue SH-coeffs
imageStore(redOutputTexture, elementPos, sumRedSHCoeffs);
```

```
imageStore(greenOutputTexture, elementPos, sumGreenSHCoeffs);
imageStore(blueOutputTexture, elementPos, sumBlueSHCoeffs);
```

Listing 4.32: Computation of the flux of each neighbor voxel to the face of the current voxel

However, in order to perform the light propagation accurately, the occlusion also needs to be accounted for.

The grid buffer that contains the color/occlusion mask and the normal masks is accessed to obtain the information of the neighboring voxel. The normals are decoded and the closest normal to the direction from the neighbor to the current voxel is calculated. By using the same zonal harmonics function used in the previous pass, the SH coefficients for the normal are computed. This provides occlusion coefficients by which the flux from each neighbor to the face of the current cell can be weighted (Listing 4.33).

```
// Determine which of the 4 specified normals (encoded as
// normalMasks) is closest to  the specified direction.
// The function returns the closest normal and as output
// parameter the corresponding dot-product.
vec3 GetClosestNormal(in uvec4 normalMasks, in vec3 direction, out float
    dotProduct)
{
  mat4x3 normalMatrix;
  normalMatrix[0] = DecodeNormal(normalMasks.x);
  normalMatrix[1] = DecodeNormal(normalMasks.y);
  normalMatrix[2] = DecodeNormal(normalMasks.z);
  normalMatrix[3] = DecodeNormal(normalMasks.w);
  vec4 dotProducts = direction * normalMatrix;

  float maximum = max(max(dotProducts.x, dotProducts.y), max(dotProducts.z,
      dotProducts.w));
  uint index;
  if(maximum == dotProducts.x)
    index = 0;
  else if(maximum == dotProducts.y)
    index = 1;
  else if(maximum == dotProducts.z)
    index = 2;
  else
    index = 3;

  dotProduct = dotProducts[index];
```

```
  return normalMatrix[index];
}

(...)

// get index of corresponding voxel
int gridIndex = (samplePos.z * 1024) + (samplePos.y * 32) + samplePos.x);
Voxel voxel = gridBuffer[gridIndex];

// If voxel contains geometry info, find closest normal to current direction
// In this way the highest occlusion can be generated.
// Then get SH-coeffs for retrieved normal.
if((voxel.colorOcclusionMask & (1<<31U)) != 0)
{
  float dotProduct;
  vec3 occlusionNormal = GetClosestNormal(voxel.normalMasks, -directions[i],
      dotProduct);
  occlusionCoeffs = ClampedCosineCoeffs(occlusionNormal);
}

(...)

// apply occlusion
float occlusion = 1.0f - clamp(occlusionAmplifier * dot(occlusionCoeffs,
   dirSH), 0.0f, 1.0f);
flux *= occlusion;
```

Listing 4.33: Computation of the occlusion coefficients of the neighboring cells. The flux is then weighted accordingly.

Summing all the contributions from the neighboring voxels correctly weighted by the occlusion coefficients provides the lighting contribution of the neighboring virtual point lights. This contribution is added to the initial value and each channel is stored in the corresponding 3D texture using image store operations (Listing 4.32). This process is repeated multiple times to allow the lighting to propagate through the grid.

4.3.4 *Reflection Grid Creation and Mipmapping*

The creation of the reflection grid is done by launching a compute shader with $256 \times 256 \times 256$ threads, since the 3D texture for this reflection grid has a 256^3 resolution.

Each thread retrieves the color/occlusion from the higher resolution grid previously generated during the voxelization pass. Then, the spherical harmonics coefficients of the finest lower resolution voxel grid are sampled (since the reflection grid has the same frustum) and lighting is extracted from the spherical harmonics coefficients of corresponding voxel (Listing 4.34).

```
ivec3 voxelPos = ivec3(gl_GlobalInvocationID);
uint iColorOcclusion = imageLoad(highResGrid, voxelPos);

// decode color/ occlusion
vec4 colorOcclusion = vec4(DecodeColor(iColorOcclusion), iColorOcclusion >>
    31U);

// get texCoords into 3D texture of propagated virtual point lights
vec3 texCoords = vec3(voxelPos) + vec3(0.5f, 0.5f, 0.0f);
texCoords /= 256.0f;

// Sample red/ green/ blue SH-coeffs trilinearly from the 3D textures and
    extract light.
vec3 ambient;
ambient.r = texture(redSH, texCoords).r;
ambient.g = texture(greenSH, texCoords).r;
ambient.b = texture(blueSH, texCoords).r;
ambient *= 0.8862269262f; // PI/(2*sqrt(PI))

colorOcclusion.rgb *= ambient;
```

Listing 4.34: Converting illuminated voxel into VPL and storing in the corresponding texture

The grid center is snapped to the camera position, so glossy reflections only cover a small area around the camera. To ensure that no popping artifacts are introduced when the camera is moving, the light is faded out with the distance to the grid center, storing the result into a mipmapped 3D texture (the reflection grid) with the same resolution in the finer level as the voxel grid generated in the voxelization pass (and an RGBA16F data format).

The light in the finer mipmap level of the reflection grid needs to be mipmapped through the coarser levels. For this purpose, a compute shader is used in the same fashion than in Section

4.2.1. The mipmapping process is performed in multiple passes, one per mipmap level, where each thread computes a simple average of the eight corresponding voxels in the higher mipmap level, storing the result using an image store operation.

4.3.5 *Global Illumination Rendering*

To compute the final image with approximate global illumination, a full-screen quad is rendered using deferred rendering and world-space positions, normals and material are retrieved from the geometry buffer.

Global illumination can be subdivided in two components: direct and indirect illumination. Direct illumination has already been computed in a previous pass using a shadow map and the result is sampled from a texture. Indirect lighting can in its turn be subdivided in two other components: diffuse and specular (in this case, glossy) indirect lighting.

Due to the usage and propagation of VPLs through the whole scene, the diffuse component does not require cone tracing. To compute diffuse indirect illumination, we start by computing spherical harmonics coefficients for the normal retrieved from the geometry buffer. Like before, zonal harmonics are used to generate these coefficients (Listing 4.31).

Then, the 3D texture coordinates for the corresponding position are computed and the three 3D textures containing the propagated virtual point lights are sampled and weighted by the normal coefficients to generate indirect diffuse illumination (Listing 4.35).

```
// Sample a LPV to get the diffuse contribution from the VPLs
vec3 GetDiffuseIllum(in vec3 offset, in vec4 surfaceNormalLobe, in sampler3D
    redSHCoeffsMap, in sampler3D greenSHCoeffsMap, in sampler3D
   blueSHCoeffsMap)
{
  // get texCoords into 3D textures
  vec3 texCoords = vec3(16.5f, 16.5f, 16.0f) + offset;
  texCoords.xyz /= 32.0f;

  // sample red/ green/ blue SH-coeffs trilinearly from the 3D textures
  vec4 redSHCoeffs = texture(redSHCoeffsMap, texCoords);
  vec4 greenSHCoeffs = texture(greenSHCoeffsMap, texCoords);
  vec4 blueSHCoeffs = texture(blueSHCoeffsMap, texCoords);

  // Do diffuse SH-lighting by simply calculating the
  // dot-product between the SH-coeffs from the virtual
  // point lights and the surface SH-coeffs.
```

```
  vec3 diffuse;
  diffuse.r = dot(redSHCoeffs, surfaceNormalLobe);
  diffuse.g = dot(greenSHCoeffs, surfaceNormalLobe);
  diffuse.b = dot(blueSHCoeffs, surfaceNormalLobe);

  return diffuse;
}
```

Listing 4.35: Sampling the VPLs in LPV in order to obtain diffuse indirect lighting

Since two grids are being used, both have to be sampled and the distance from the center of the grid (unsnapped) is used to interpolate between their resulting indirect diffuse lighting contribution to provide a smooth transition between them (Listing 4.36, Figure 75).

```
// The distance for lerping between fine and coarse resolution grid has to
   be calculated with
// the unsnapped grid-center, in order to avoid artifacts in the lerp area.
// gridCenter is passed as an uniform (camera Position)
// fragmentPosition is the world position retrieved from the g-buffer
// inverseFineGridCellSize is also passed as an uniform
vec3 lerpOffset = (fragmentPosition - gridCenter)*inverseFineGridCellSize;
float lerpDist = length(lerpOffset);

// lerp between results from both grids
// diffuseGIPower is a user-defined value to control the diffuse light
   intensity
float factor = clamp((lerpDist - 12.0f) * 0.25f, 0.0f, 1.0f);
diffuseIllum = mix(fineDiffuseIllum, coarseDiffuseIllum, factor);
diffuseIllum = max(diffuseIllum, vec3(0));
diffuseIllum /= PI;
diffuseIllum = pow(diffuseIllum, vec3(diffuseGIPower));
```

Listing 4.36: Interpolation between the two diffuse lighting grids

For glossy reflections, similarly to section 4.2, voxel cone tracing is used to accumulate lighting contributions from the voxels along the cone axis.

First, the reflected direction from the eye to the world-position of the fragment is computed and a ray is launched in that direction, accumulating color and occlusion from the lit reflection grid until total occlusion is reached (Figure 76).

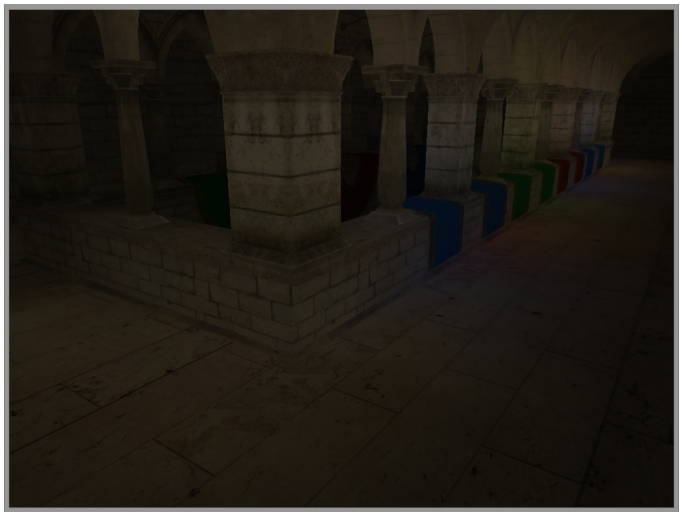


Figure 75: Result obtained from applying the VPLs for indirect diffuse lighting.

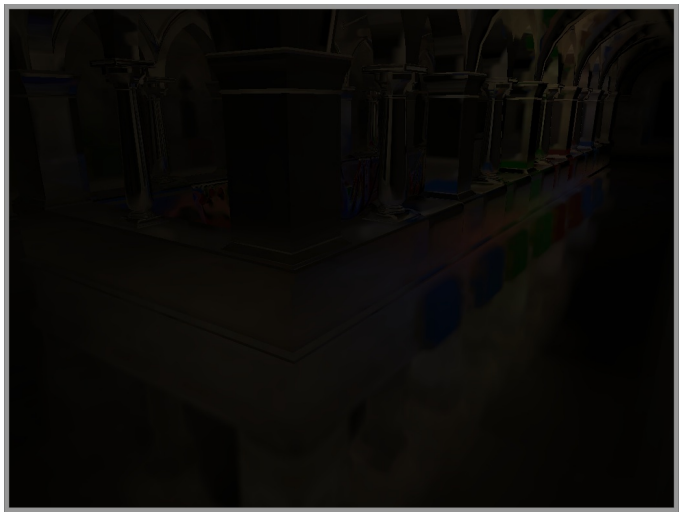


Figure 76: Result obtained from using voxel cone tracing to compute the glossy reflections from the reflection grid.

The difference from the previously detailed voxel cone tracing pass (Listing 4.13) is that the sampled values are faded out according to the maximum propagation distance and the distance to the grid center in order to ensure a smooth fade-out of the reflection as the camera moves through the scene (Listing 4.37). This is because the reflection grid does not cover the whole scene.

```
// In the VCT pass, fade out sampled values with maximum propagation
// distance to be synchronous with propagated virtual point lights
float fade = clamp(traceDistance / maxTraceDist, 0.0f, 1.0f);
vec4 colorOcclusion = textureLod(reflectionGrid, texCoords, level) * fade;
```

```
(...)

// Fade out accumulated reflection with the distance to the reflection grid
    center.
// In combination with writing initially the color values the same
// way into the reflection-grid, this ensures a smooth fade-out of the
    reflections as the viewer camera moves.
reflection *= 1.0f - clamp(length(position)/resolutionGridExtent, 0.0f, 1.0f
    );
```

Listing 4.37: Fading the reflection in order to provide a smooth fade-out as the distance increases

Finally, direct, diffuse indirect and glossy reflections are added together to generate the approximation to global illumination (Figure 77).

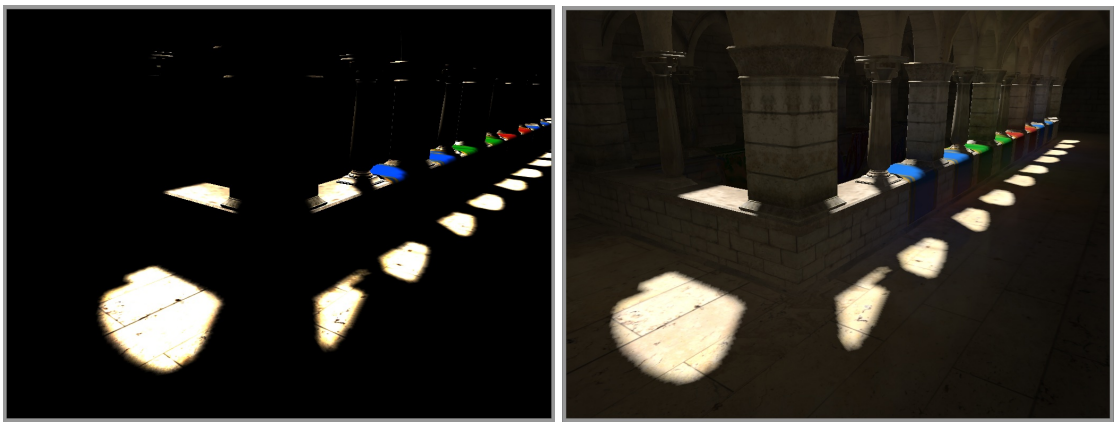


Figure 77: Result obtained from combining the indirect diffuse and specular lighting with the direct light result

Since the center of the voxel grids is kept synchronous with the viewer camera, the global illumination is kept to a limited frustum around it. However, in this manner the algorithm does not depend on the size of the scene, being perfectly capable of handling large scenes without losing its interactivity.

Another potential advantage is that the reflection grid is separate, making it very easy to disable it and spare memory and processing time if the indirect diffuse contribution is sufficient for the current scene.

4.4 REAL-TIME NEAR-FIELD GLOBAL ILLUMINATION BASED ON A VOXEL MODEL

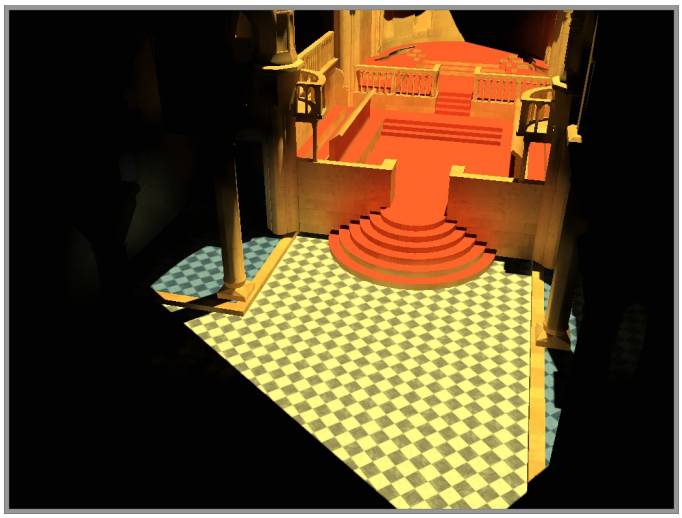


Figure 78: Result obtained by the near-field global illumination algorithm

This algorithm allows the computation of an approximation to indirect diffuse illumination in real time by using a binary voxel representation of the scene together with reflective shadow maps (Figure 78).

First of all, bitmasks are computed on the CPU and stored into 1D and 2D textures in order to send them to the shaders when needed.

The scene is then voxelized into a 2D texture by using a mapping of the vertices of the objects to a texture atlas. Since the scene is going to be rasterized by disabling the depth test and face culling, the texture atlas must guarantee that each triangle will have its own space in the resulting 2D texture. In this work the atlas provided by the authors were used.

After the voxelization process, a display list is created in order to discard invalid texel values from the texture atlas created during the voxelization pass.

The display list is used with point rendering to inject the voxel values into a voxel grid represented by a 2D texture and the voxel grid is mipmapped in order to produce an hierarchical representation of the scene.

The main difference with the other algorithms described in this thesis is that no direct light injection pass is performed. Since the voxel grid only encodes a binary representation of the scene, direct lighting will have to be sampled during the indirect lighting computation pass, using a reflective shadow map.

The algorithm can thus be subdivided into several passes:

1. Bitmask Creation
2. Voxelization
    a) Binary Atlas Creation
    b) Pixel Display List Creation
    c) Voxel Grid Creation
3. Mipmapping
4. Indirect Lighting Computation

4.4.1 *Bitmask Creation*

The voxel grid created by the voxelization process is encoded using a 2D texture where each texel represents a stack of voxels. The voxels are encoded one per bit in order to generate the binary representation of the scene.

To encode or decode the voxels from the voxel grid, several bitmasks are created in the CPU, in a preprocessing step.

First, a bitmask for the OR test is created, which will be useful for inserting the voxels in the voxel grid. This is done by creating a 1D texture with 128 bits, resulting in 128 distinct depth values (32 bits per value and 4 channels RGBA) and a data format RGBA32UI.

The texture encodes first 32 RGBA values in which the R component has only one bit set, starting from the most significant bit and ending in the less significant bit, while the other channels are null (Listing 4.38). Then, the same is done for the other 3 color channels.

```
int bits = 32; // bits per texture channel
GLuint bitposition = (GLuint)(pow(2.0, (double)bits - 1));//0x80000000U;
GLuint R, G, B, A;

std::vector<GLuint> bitmaskORData;
for (int i = 0; i < bits; i++)  // first 31 texels: 1-bit in R
{
  R = bitposition;
  G = 0;
  B = 0;
  A = 0;

  bitmaskORData.push_back(R);
```

```
  bitmaskORData.push_back(G);
  bitmaskORData.push_back(B);
  bitmaskORData.push_back(A);

  bitposition = bitposition >> 1;
}
```

Listing 4.38: Bitmask OR creation for the R channel

Another 1D texture is then created with size 128 and a data format RGBA32UI in order to encode the bitmask for XOR test. This is done in a similar fashion as the for the OR bitmask, but instead of setting one bit per color channel, all bits but one are set (Listing 4.39).

```
int bits = 32; // bits per texture channel
GLuint all_1 = (GLuint)(pow(2.0, (double)bits) - 1);//0x7FFFFFFFU (2^31 -1)
GLuint shifted_ones = all_1;
GLuint R, G, B, A;

std::vector<GLuint> bitmaskXORData;
for (int i = 0; i < bits; i++)  // first 31 texels: 1-bit in R
{
  R = shifted_ones;
  G = all_1;
  B = all_1;
  A = all_1;

  bitmaskXORData.push_back(R);
  bitmaskXORData.push_back(G);
  bitmaskXORData.push_back(B);
  bitmaskXORData.push_back(A);

  shifted_ones = shifted_ones >> 1;
}
```

Listing 4.39: Bitmask XOR creation for the R channel

The bitmask XOR is then used in order to compute a 2D texture that will be used to find out if a voxel has been intersected by a ray. This 2D texture is a square with size 128 and with a data format RGBA32UI. By attaching this texture to the current framebuffer and rendering a fullscreen quad, a shader can be used to compute the ray bitmasks using the bitwise exclusive or operator (Listing 4.40).

```
layout(binding = 0) uniform usampler1D bitmaskXOR;
layout(location = 0) out uvec4 bitRay;

void main()
{
   bitRay = texelFetch(bitmaskXOR, int(gl_FragCoord.x), 0) ^ texelFetch(
      bitmaskXOR, int(gl_FragCoord.y)+1, 0);
}
```

Listing 4.40: Fragment shader for the computation of the ray bitmasks

4.4.2 *Binary Atlas Creation*

In order to be able to generate the binary atlas texture, each model loaded by the application must have a mapping of the surfaces of the object to a texture atlas. If the model already loads its textures from a texture atlas, the texture coordinates of the model could be used to perform the creation of the binary atlas. However, to avoid problems such as holes or overdraw during the voxelization and creation of the voxel grid, some care must be taken with the size of the resulting binary atlas (Thiedemann et al., 2012). This means that the size of the binary atlas must be set according to the object being transformed and the resolution of the voxelization pass. A size too large will cause overdraw since multiple voxels might end up in the same cell, while a size too small might cause holes in the resulting voxel grid due to missing voxels.

To store the binary atlas, a 2D texture with half size floating point RGBA values is created (RGBA16F). First, the binary atlas is attached to the framebuffer and cleared with some value in order to be able to identify invalid texels in the next pass. Then a fullscreen quad is rendered with a viewport matching the size of the atlas. As for the other voxelization approaches described previously, the depth test and face culling are disabled in order to avoid missing information in the resulting voxelization.

The vertex shader computes the world-space position, sending it to the fragment shader, and transforms the atlas texture coordinate into Normalized Device Coordinates (NDC). The fragment shader simply outputs the received world-space position into its corresponding position in the atlas texture (Figure 79).

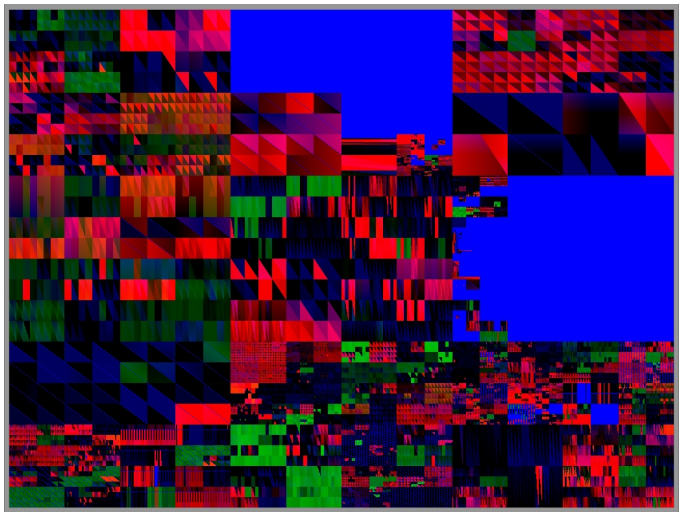


Figure 79: Result obtained by rendering the scene into a texture atlas

In order to be able to insert or remove objects to the scene without having to recompute the whole atlas, the scene is rendered once for each object with different atlas textures attached to the framebuffer, so that each object will have its corresponding binary texture atlas.

4.4.3 *Pixel Display List Creation*

Now that the atlas texture contains the world-space positions of the surfaces of the object, it would be possible to generate one vertice for each texel in the texture atlas and issue a draw call that would insert these vertices into the voxel grid. Another way would be to render a fullscreen quad with a viewport size matching the atlas texture dimensions.

However, a lot of texels contain invalid values that we wish to discard to reduce the amount of vertices issued in the draw call.

So, in the same manner as Thiedemann et al. (2012), the atlas texture is read back to the CPU and traversed, discarding all invalid texels using the previously defined threshold value with which the atlas texture was cleared (Listing 4.41).

For each valid texel, a point is generated using the texture coordinates (which vary between 0 and the atlas resolution) and inserted into a pixel display list. One advantage of this process is that the point size can be increased in order to attempt to close holes if the resolution of the texture atlas is too small.

```
GLfloat* positionBuffer = new GLfloat[atlasResolution * atlasResolution *
    3];
```

```
// read the binary atlas back to the CPU
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, atlas);
glGetTexImage(GL_TEXTURE_2D, 0, GL_RGB, GL_FLOAT, positionBuffer);

// Create a display list
pixelDisplayList = glGenLists(1);
glNewList(pixelDisplayList, GL_COMPILE);

glPointSize(1.0);   // = 1.0: exactly one voxel per vertex
                   // > 1.0: more than 1 voxel per inserted point (might
                      close holes with too small atlas)

glBegin(GL_POINTS);
for (int y = 0; y < atlasResolution; y++)
{
  for (int x = 0; x < atlasResolution; x++)
  {
    // read the world positions
    float z = positionBuffer[(3 * y * atlasResolution + 3 * x) + 2];
    if (z < threshold) // valid
    {
      glVertex2i(x, y);
      counter++;
    }
  }
}
glEnd();
glEndList();

delete[] positionBuffer;
```

Listing 4.41: Creation of the pixel display list from the binary atlas. This display list will be used to store the voxels in the binary voxel grid.

4.4.4 *Voxel Grid Creation*

The next step is to generate the binary representation of the scene stored in the form of a binary voxel grid. The binary voxel grid is a mipmapped 2D texture storing RGBA unsigned integers. Its size is defined in accordance with the voxelization resolution and each texel represents a stack of 128 voxels along the voxelization depth.

The creation of the binary voxel grid starts by rendering the pixel display list with the voxel grid texture bound to the framebuffer. Since a binary voxelization is being used, the RGBA channels of the voxel grid texture encode the voxels using a bitmask. To be able to use the bitmask OR texture created previously to store the voxels in the voxel grid, a logical OR operation has to be defined for the framebuffer (Listing 4.42).

```
glEnable(GL_COLOR_LOGIC_OP);
glLogicOp(GL_OR);
```

Listing 4.42: Enabling a logical OR operation on the framebuffer

The bitmask OR texture is bound and an orthographic projection is defined in order to be able to transform the world-space positions of the vertices in the pixel display list into voxel grid coordinates. In this way, the extents of the orthographic projection control the region that is voxelized.

The vertex shader fetches the world-space position from the atlas texture using the texture coordinate passed as a vertex from the pixel display list. The world-space coordinate is transformed (using the view and orthographic projection matrices) in order to obtain the voxel grid coordinates. The Z component of the computed voxel grid coordinates is then mapped to [0, 1] and passed to the fragment shader. Since each texel of the voxel grid actually represents a stack of voxels along a certain depth, this Z coordinate is actually the coordinate of the bitmask that needs to be retrieved from the bitmask OR texture in order to set the correct bit in the voxel grid using the logical OR operation on the framebuffer (Listing 4.43).

```
// Vertex Shader
in vec4 position; // positions in range [0..atlasWidth-1]x[0..atlasHeight-1]
out float mappedZ;
layout(binding = 0) uniform sampler2D textureAtlas; // contains positions in
    world space

void main ()
{
```

```
  // Fetch world space position from atlas.
  vec3 pos3D = texelFetch(textureAtlas, ivec2(position.xy), 0).xyz;

  // Transform into voxel grid coordinates
  gl_Position = projectionMatrix * viewMatrix * vec4(pos3D, 1.0);

  // Map z-coordinate to [0,1]
  mappedZ = gl_Position.z * 0.5 + 0.5;
}


// Fragment Shader
layout(binding = 1) uniform usampler1D bitmask;
in float mappedZ;
layout (location = 0) out uvec4 result;

void main()
{
   result = texture(bitmask, mappedZ); // Set bit in voxel grid
}
```

Listing 4.43: Shaders for the creation of the voxel grid from the pixel display list

4.4.5 *Mipmapping*

The voxel grid needs to be mipmapped in order to generate a binary hierarchical representation of the scene. To perform the mipmapping, a full-screen quad is rendered for each of the lower levels of the mipmapped texture and with the respective mipmap level bound to the framebuffer.

The fragment shader then samples the 4 neighbor texels in the upper level of the mipmap hierarchy. Since a texel represents a stack of voxels along the depth axis, the depth is kept for each mipmap level, so only the x and y axis are joined. To achieve this, the 4 texels are simply joined using a logical OR operation (Listing 4.44).

```
layout (location = 0) out uvec4 result;
layout(binding = 0) uniform usampler2D voxelTexture;
uniform int level; // read from this mipmap level
uniform float inverseTexSize; // 1.0 / mipmapLevelResolution

void main()
```

```
{
   // texture coordinates of 4 neighbor texels in source voxel texture
  vec2 offset1 = vec2(inverseTexSize, 0.0);   // right
  vec2 offset2 = vec2(0.0, inverseTexSize);   // top
  vec2 offset3 = vec2(inverseTexSize, inverseTexSize); // top right
  vec2 coord; // this pixel
  coord.x = (((gl_FragCoord.x-0.5)*2.0)+0.5)*inverseTexSize;
  coord.y = (((gl_FragCoord.y-0.5)*2.0)+0.5)*inverseTexSize;

  // Lookup 4 neighbor texels ( ~ voxel stacks)
  uvec4 val1 = textureLod(voxelTexture, coord, level);
  uvec4 val2 = textureLod(voxelTexture, coord+offset1, level);
  uvec4 val3 = textureLod(voxelTexture, coord+offset2, level);
  uvec4 val4 = textureLod(voxelTexture, coord+offset3, level);

  result = val1 | val2 | val3 | val4;
}
```

Listing 4.44: Mipmapping of the binary voxel grid

4.4.6 *Indirect Lighting Computation*

To compute the diffuse indirect lighting, two buffers are needed. One is used to bounce the rays around the scene and the other actually stores the diffuse indirect illumination after computation, so that it can be added to direct lighting afterwards. Both are defined as 2D textures storing RGB values at half floating point precision and with a size matching the window extents.

Multiple passes are performed in order to launch several rays per pixel. For each of these rays, the voxel grid is used to compute a ray/voxel intersection, writing the hit positions into a buffer called the hit buffer. The hit buffer is then read in another pass and diffuse indirect illumination is computed with the help of reflective shadow maps. Hardware blending is used to average the samples obtained by the different rays.

Let us start with the creation of the hit buffer. A full-screen quad is rendered using deferred rendering and the world-position is fetched from the geometry buffer. This world-space position is the starting point of the reflected ray.

Launching rays through the scene is performed using a ray tracing algorithm. By using a cosine weighted distribution, a direction for the ray is computed. For a better sampling of the

diffuse indirect illumination, a Hammersley sequence (Colbert and Krivánek, 2008) is created in the CPU and used together with an auxiliary texture to rotate the rays in a random direction (Listing 4.45). This texture is created previously on the CPU and it is defined as a small 2D texture storing random data (with values between 0 and 1) in RGB floating point format.

```
vec3 P_world = texture(positionBuffer, UV).xyz; // this pixel's world space
    coord.

vec2 rand = texelFetch(randTex, ivec2(mod(ivec2(gl_FragCoord.st), ivec2(
    randTexSize))), 0).rg;

// currentRay is the index of the current ray to be traced
float u1 = fract(samplingSequence[currentRay].x + useRandRay * rand.x);
float u2 = fract(samplingSequence[currentRay].y + useRandRay * rand.y);

// cosine weighted sampling (pdf = cos / PI)
float r = sqrt(u1*spread);
float phi = 2 * PI * u2;

float x = cos(phi) * r;
float y = sin(phi) * r;
float z = sqrt(max(0.0, 1.0 - x*x - y*y));

// construct ray
vec3 ray = normalize(x*Tangent + y*BiTangent + z*Normal);
```

Listing 4.45: Generation of the reflected ray using an Hammersley sequence for importance sampling

The start and end point of the ray are then defined. To avoid self-shadowing, the starting point of the ray is advanced by an offset at least the size of the voxel diagonal (Listing 4.46).

```
// Start and end point of the ray segment in world coordinates
// voxelOffsetNormalScale is a user defined offset scaling factor
// Offsets are scene-dependent -> used to prevent self-shadowing
origin   = P_world + voxelOffsetNormalScale * voxelDiagonal * N +
    startOffset * ray;
vec3 end = P_world + voxelOffsetNormalScale * voxelDiagonal * N + radius *
    ray;
```

Listing 4.46: Definition of the start and end point of the ray

The intersection test is performed in a loop, advancing the ray in small steps and testing against the mipmap hierarchy if an intersection is encountered. In order to limit the computation time, a number of maximum iterations is defined by the user. Also, the sampling does not initiate at the finest mipmap level (the root node), but is advanced by at least one level since the intersection with the root node was already computed (Listing 4.47).

```
bool IntersectHierarchy(in int level, in vec3 posTNear, inout float tFar,
    out uvec4 intersectionBitmask)
{
  // Calculate pixel coordinates ([0,width]x[0,height]) of the current
     position along the ray
  float res = float(1 << (maxMipMapLevel - level));
  ivec2 pixelCoord = ivec2(posTNear.xy * res);

  // Voxel width and height in the unit cube
  vec2 voxelWH = vec2(1.0) / res;

  // Compute voxel stack (AABB) in the unit cube belonging to this pixel
     position
  // (Slabs for AABB/Ray Intersection)
  vec2 box_min = pixelCoord * voxelWH; // (left, bottom)

  // Compute intersection with the bounding box
  // It is always assumed that an intersecion occurs
  // origin and dir are global variables
  vec3 tmin = (vec3(box_min, 0.0) - origin) / dir;
  vec3 tmax = (vec3(box_min + voxelWH, 1.0) - origin) / dir;

  vec3 real_max = max(tmin,tmax);

  // the minimal maximum is tFar
  tFar = min(1.0, min( min(real_max.x, real_max.y), real_max.z));

  // Now test if some of the bits intersect
  float zFar = tFar*dir.z + origin.z ;

  // Fetch bitmask from hierarchy and compute intersection via bitwise AND
  intersectionBitmask = (texture(bitmaskXORRays & texelFetch(voxelTexture,
     vec2(min(posTNear.z, zFar), max(posTNear.z, zFar)));
  return (intersectionBitmask != uvec4(0));
}
```

```
(...)

// Set current position along the ray to the ray's origin
vec3 posTNear = origin;

bool intersectionFound = false;
uvec4 intersectionBitmask = uvec4(0);

// Choose the mipmap level for the initial iteration
int level = min(3, maxMipMapLevel-1);

for(int i = 0; (i < steps) && (tNear <= tFar) && (!intersectionFound); i++)
{
    float newTFar = 1.0f;

    if(IntersectHierarchy(level, posTNear, newTFar, intersectionBitmask))
    {
        // If we are at mipmap level 0 and an intersection occured, we have
           found an intersection of the ray with the voxels
        intersectionFound = (level == 0);

        // Otherwise we have to move down one level and start testing from
           there
        level --;
  }
  else
  {
    // If no intersection occurs, we have to advance the position on the ray
         to test the next element of the hierachy.
    // Furthermore, add a small offset computed beforehand to handle
       floating point inaccuracy.
    tNear = newTFar + offset;
    posTNear = origin + tNear * dir;

    // Move one level up
    level++;
  }
}
```

Listing 4.47: Ray tracing through the binary voxel grid and intersection test

Since voxels are encoded in groups, they can be represented with an Axis Aligned Bounding Box (AABB). So, to perform the intersection test, a ray/AABB test is employed. If the ray intersects the bounding box, the bitmask is tested against the ray bitmask computed during initialization using a bitwise operation AND to check whether there are bits that intersect the ray. The intersection point is then used to compute and write the voxel position of the intersection point (in world-coordinates) into the hit buffer (Listing 4.48).

```
if(intersectionFound)
{
    // Compute the position of the highest or lowest set bit in the
       resulting bitmask
    // Compute highest bit if ray is not reversed to the voxelization
       direction, lowest if reversed.
    bool reversed = dot(ray, voxelizingDirection) < 0.0;
    int bitPosition = 0;
    int x = 0;
    if(!reversed)
    {
        // get the position of the highest bit set
        int v;
        for(v = 0; x == 0 && v < 4; v++) // r g b a
        {
            x = int(intersectionBitmask[v]);
            if(x != 0)
            {
                int pos32 = int(log2(float(x)));
                bitPosition = (3-v)*32 + pos32;
            }
        }
    }
    else
    {
        // get the position of the lowest bit set
        int v;
        for(v = 3; x == 0 && v >= 0; v--) // r g b a
        {
            x = int(intersectionBitmask[v]);
            if(x != 0)
            {
                int pos32 = int(log2(float(x & ~(x-1)))+0.1);
                bitPosition = (3-v)*32 + pos32;
```

```
            }
        }
    }

  // 0.0078125 = 1/128 (length of one voxel in z-direction in unit-
     coordinates)
  // 0.00390625 = 0.5/128 (half voxel)
  posTNear.z = float(127 - bitPosition) * 0.0078125  + 0.00390625;

  hitPos = (inverseViewProjToUnitMatrixVoxelCam * vec4(posTNear, 1.0)).xyz;
}
```

Listing 4.48: Finding the real hit position of the intersection

Now that the hit buffer is filled with the hit positions, all the information necessary to compute diffuse indirect illumination is available.

The algorithm starts by fetching the intersection point from the hit buffer and transform it into light space using the view and projection matrices from the shadow map pass.

Then, this coordinate is projected into the reflective shadow maps to retrieve the corresponding position, normal and direct lighting color (Figure 80).

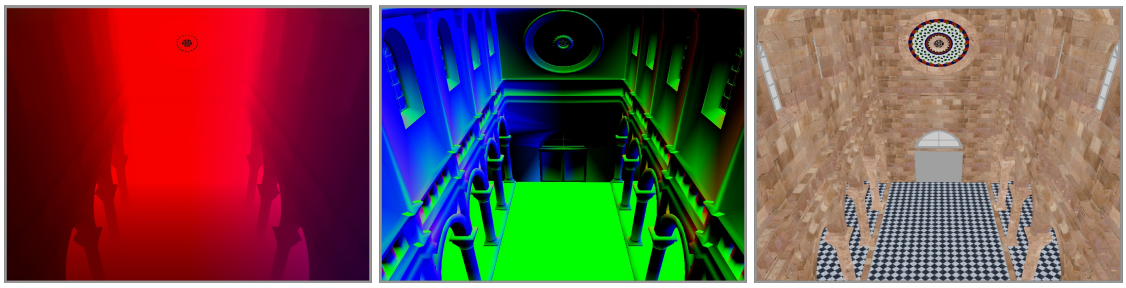


Figure 80: Reflective shadow map storing world positions, normals and material color.

The direct radiance is only valid if the distance from the hit position to the world-space position retrieved from the reflective shadow map is smaller than a certain threshold, or else the hit point lies in the shadow of the source. Also, since only front faces are lit and reflect indirect light, the normal is used to check whether the hit point lies in the front face of the surface (Listing 4.49).

```
void main()
{
   // initialize resulting luminance with black
   result = vec3(0);
```

```
    // get intersection point with voxel scene
    vec3 hitPos = texture(hitBuffer, UV).xyz;

    // transform intersection point into light space
    vec4 projCoord = shadowMapProjection * shadowMapView * vec4(hitPos, 1.0);
    projCoord.xyz /= projCoord.w;
    projCoord.xyz = projCoord.xyz * 0.5 + 0.5;

    // fetch RSM values
    vec4 position  = textureProj(positionMap, projCoord);
    vec4 color = textureProj(colorMap, projCoord); // luminance
    vec3 normal = textureProj(normalMap, projCoord).rgb;

    // compute ray from hit position to g-buffer position
    vec3 ray = normalize(texture(GPosition, UV).xyz - hitPos);
    float lightZ = position.w;
    float pixelSide = pixelSideZNear * lightZ; // pixelSideZNear = pixelSide
        / zNear

    // only front faces are lit and senders of indirect light
    bool normalCondition = normalCheck ? dot(normal, ray) >= 0.0 : true;
    float cosAlpha = normalCheck ? max(0.001, dot(normal, lightDirection)) :
        1.0;

    // check if the hit point is valid
    if(hitPos.z < 100.0
        && (projCoord.w > 0.0)
        && distance(hitPos, position.xyz) <  min(4.0 * voxelDiagonal, max(
            voxelDiagonal, pixelSide/cosAlpha*distanceThresholdScale))
        && normalCondition)
    {
        // output luminance
        // sampleContrib = user-defined contrast / number of samples
        result = color.rgb * sampleContrib;
    }
}
```

Listing 4.49: Computation of diffuse indirect illumination using the hit buffer and RSMs

Finally, direct and indirect illumination are combined by rendering a full-screen quad and summing the contributions of each pixel. To avoid noise in the resulting image, the indirect

lighting result is blurred using a geometry aware blur before adding its contribution to the global illumination result.

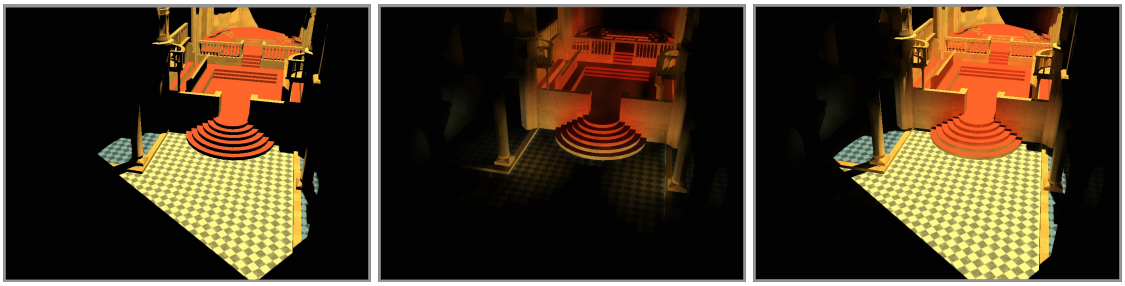


Figure 81: Direct (left) and indirect (middle) are added together to produce global illumination (right).

4.5 VOLUMETRIC LIGHT EFFECTS

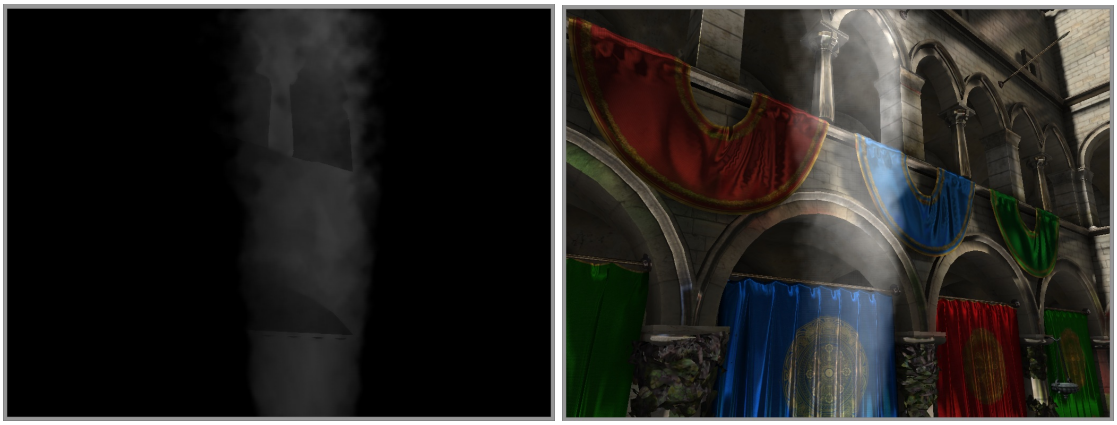


Figure 82: Volumetric smoke effect produced from a voxel grid.

Since using a regular voxel grid wastes a lot of memory with empty voxels to compute the approximation to global illumination, this empty space can be used for the computation of other effects.

Voxel data has been used for multiple purposes in computer graphics. One of such examples is the computation of volumetric effects. Since in reality aerosols like fog, dust or mist never occupy the same volume in space than solid geometry, this observation can be used to use the empty space of the voxel grid to store the data necessary for the calculation of volumetric effects (Figure 82).

The main idea is simple. The volumetric effects are simulated through particle effects, and particles are used to store information on the voxel grid. This grid is traversed afterwards using ray marching and multiple samples are taken in order to compute the atmospheric effect.

First, a particle system is used in order to simulate particles for the desired effect. 10000 particles were used to simulate a smoke effect, in which each particle moves in an ascending direction as time passes (Figure 82).

Each such particle uses an **imageAtomicAdd** operation to add its contribution to the corresponding voxel in the voxel grid. At the end of this pass, each voxel in the voxel grid contains information about how many particles are contained inside its corresponding volume in space (Listing 4.50).

```
layout(binding = 0, r32ui) coherent volatile uniform uimage3D voxelGrid;
uniform int voxelResolution;
in vec4 worldPos;

void main()
{
  vec3 texPos = worldPos.xyz * 0.5 + 0.5;
  imageAtomicAdd(voxelGrid, ivec3(texPos * voxelResolution), 1);
}
```

Listing 4.50: Each particle updates a counter in the voxel atomically.

The volumetric lighting effect is computed at half the resolution of the screen in order to be fast (performance-wise). For this purpose, a 2D texture with an RGBA16F data format is created using half the window dimensions. This texture is bound to the framebuffer and a fullscreen quad is rendered using deferred rendering (considering a directional light that cover the whole scene).

First, the world position is retrieved from the geometry buffer. Together with the camera position, this world position is used to compute a direction which will be used to launch a ray for the ray marching process (Listing 4.51).

The ray marching step value is computed according to the number of steps (which is a user-defined value). The starting position is advanced using an offset computed from a 4x4 dither pattern in order to allow using a relatively low number of steps in the ray marching process and avoid banding artifacts (Figure 83).

```
// 4x4 Dither pattern based on the Bayer Matrix
float ditherPattern[4][4] =
{
    { 0.0f, 0.5f, 0.125f, 0.625f},
```

```
    { 0.75f, 0.22f, 0.875f, 0.375f},
    { 0.1875f, 0.6875f, 0.0625f, 0.5625},
    { 0.9375f, 0.4375f, 0.8125f, 0.3125}
};

vec3 worldPos = texture(positions, UV).xyz;
vec3 rayVector = worldPos - cameraPos;

float rayLength = length(rayVector);
vec3 rayDirection = rayVector / rayLength;

float stepLength = rayLength / NUM_STEPS;    // NUM_STEPS is user-defined
vec3 stepVal = rayDirection * stepLength;

// Dithered Ray Marching
vec2 xy = gl_FragCoord.xy;
int x = int(mod(xy.x, 4));
int y = int(mod(xy.y, 4));

vec3 currentPosition = cameraPos + stepVal * ditherPattern[x][y];
```

Listing 4.51: Computation of the ray starting point and direction for the ray marching process. The starting point is advanced by an offset computed using a dither pattern.

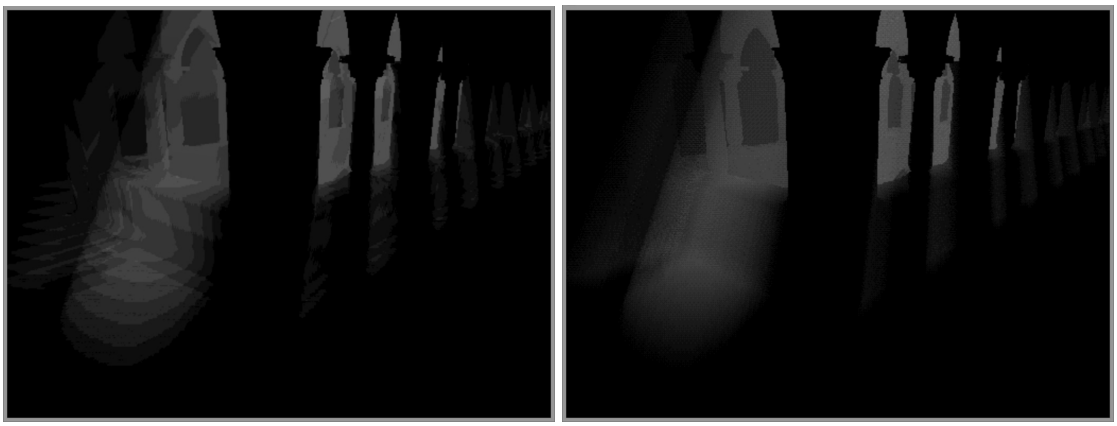


Figure 83: A low number of samples produces banding in the result. Adding an offset based on a dither pattern adds noise, improving the result.

The current sampling position is projected against a shadow map to find out if it receives (and thus scatters) light. If light is visible from the current sampling position, the voxel grid is sampled to retrieve the scattering amount (Listing 4.52).

```
for (int i = 0; i < NUM_STEPS; i++)
{
  vec4 shadowCoord = shadowMapProjection * shadowMapView * vec4(
     currentPosition, 1.0f);
  shadowCoord.xyz /= shadowCoord.w;
  shadowCoord.xyz = shadowCoord.xyz * 0.5 + 0.5;

  float shadowMapValue = texture(shadowMap, vec3(shadowCoord.xy, shadowCoord
     .z));

  if (shadowMapValue > 0)
  {
    vec3 texPos = currentPosition * 0.5 + 0.5;
    float val = texture(voxelGrid, texPos).x;
    float gScatter = 1.0 - clamp(255*val, 0.0, 1.0);

    accumFog += ComputeScattering(dot(rayVector, lightPosition), gScatter).
       xxx * lightColor.rgb;
  }

  currentPosition += stepVal;
}

accumFog /= NB_STEPS;

volumetric = vec4(accumFog, 1.0);
```

Listing 4.52: Ray marching through the voxel to retrieve the scattered light in the camera direction

The scattering amount is used with the Henvey-Greenstein phase-function to compute the amount of light scattered in the direction of the camera (Listing 4.53).

```
float ComputeScattering(float lightDotView, float gScatter)
{
  float result = 1.0f - gScatter;
  result *= result;
  result /= (4.0f * PI * pow(1.0f + (gScatter * gScatter) - (2.0f * gScatter
     ) * lightDotView, 1.5f));

  return result;
}
```

Listing 4.53: Mie scaterring approximated with Henyey-Greenstein phase function

Finally, the position is advanced by the ray marching step and the process is repeated until all the samples have been taken.

Since dithering has been used to advance the starting point of the rays and since the result has been generated with only half the resolution of the current window, a bilateral upsampling is employed to smooth the result, conserving at the same time the edges of the geometry in the scene (Figure 84).

Figure 84: By applying a bilateral upsampling, the dithered result is smoothed (left) and applied to the scene (right).

The performance measurements can be found on Table 2. By using half the resolution of the screen, the dithered ray marching process is much faster.

	Half Resolution	Full Resolution
Particle Update	0.21	0.21
Dithered Ray Marching	0.49	1.80
Bilateral Upsampling	1.09	1.14
Measured FPS(with GI)	41	39

Table 2: Performance measurements (in ms) for the volumetric lighting at full resolution (1024×768) and half the screen resolution.

The process described until now is somewhat detached from the computation to the approximation to global illumination. In fact, it could be done completely separately (although still using the same voxel grid) and added to the final result. However, a problem would appear: the cone tracing step does not take into account the smoke when computing the glossy reflections.

One way to solve this would be to launch an additional ray during the cone tracing step, to compute the volumetric effect. However, the optimizations made to increase performance by rendering at half the resolution of the window would be useless, thus increasing the computational cost of this effect.

4.6 ANALYSIS

All the algorithms have been implemented using OpenGL and the Very Simple * Libs (Lighthouse3d), which provide a profiler for CPU and GPU. All times have been measured in an NVIDIA Geforce GTX 750 Ti and driver version 353.30.

For the algorithm described in section 4.4, since a texture atlas was needed, the scene was tested using the Sibenik Catedral model and its respective atlas, provided with the reference implementation. All the other algorithms were tested using the Sponza model provided at Crytek's website.

In order to compare the visual result from the different algorithms, a reference image was obtained from Sugihara et al.. The paper states that the reference image was obtained using the Embree Path Tracer (Intel) and using 1024 samples per pixel.

The performance times (in ms) for the grid and sparse voxel octree implementations (4.2) can be seen in Table 3.

The sparse voxel octree, besides its difficult implementation, is much slower than using a voxel grid. The performance is still better if an anisotropic voxel grid is used, which increases the number of grids that need to be maintained by six times. Even if the scene is considered static and the SVO is created as a preprocess, the voxel cone tracing pass is several orders of magnitude slower.

Also, the voxel grid uses roughly 67MB of memory for a grid with a size 256^3. The objetive of the sparse voxel octree is to reduce the memory used by a regular grid. For the sponza model, the sparse voxel octree actually uses around 80MB more than a single regular grid. This is due to the need to encode colors, normals and irradiance in bricks with size 3^3, while the regular grid only stores directly the irradiance (since it does not assume a static scene and revoxelizes every frame).

So, the sparse voxel octree is not a viable solution for encoding the hierarchical voxel representation of the scene.

Comparing isotropic and anisotropic representations in terms of performance, it is obvious that the need to maintain multiple grids has its impact on performance. As would be expected,

the computation times from the light injection and occlusion correction are increased by about six times. The mipmapping process is even slower, probably due to the use of multiple texture fetches instead of using hardware trilinear sampling to retrieve the voxel values (since they cannot be simply averaged). The cone tracing pass is also slower, due to the need to perform three texture samples instead of a single one to gather lighting in the scene.

	Full Grid (Isotropic)	Full Grid (Anisotropic)	Sparse Voxel Octree
Voxelization	2.05	2.07	4.85
SVO Creation	-	-	64.72
Light Injection	1.76	10.46	3.51
Correct Occlusion	1.88	11.33	-
Mipmap	0.71	7.93	3.86
Voxel Cone Tracing	4.20	10.91	51.74
Measured FPS	45	19	8

Table 3: Performance measurements (in ms) for the voxel cone tracing algorithm, using multiple data structures: a single full grid, six grids for anisotropic filtering and a sparse voxel octree.

In terms of visual quality, the anisotropic representation produces a darker output, since light does not propagate in every direction. Both anisotropic and isotropic grids fail to capture some details such as the color bleeding in the pillars.

Table 4 shows the performance timings (in ms) for the light propagation volumes technique described in section 4.3.

	Light Propagation Volume
Voxelization	12.28
Direct Light Injection	0.11
Direct Light Propagation	1.79
Reflection Grid Creation	4.61
Reflection Grid Mipmap	2.26
Global Illumination	7.86
Measured FPS	30

Table 4: Performance measurements (in ms) for the Light Propagation Volume implementation.

The voxelization process is a lot slower than the one measured in Figure 85 for the single voxel grid. Although 3 grids are used in this case, two of them have a very low resolution.

The direct light injection and propagation are very fast, since the grids have a very low resolution. The creation and mipmapping of the reflection grid takes a bit more time, but that was expected due to the need to decode the VPLs and the higher resolution of the grid.

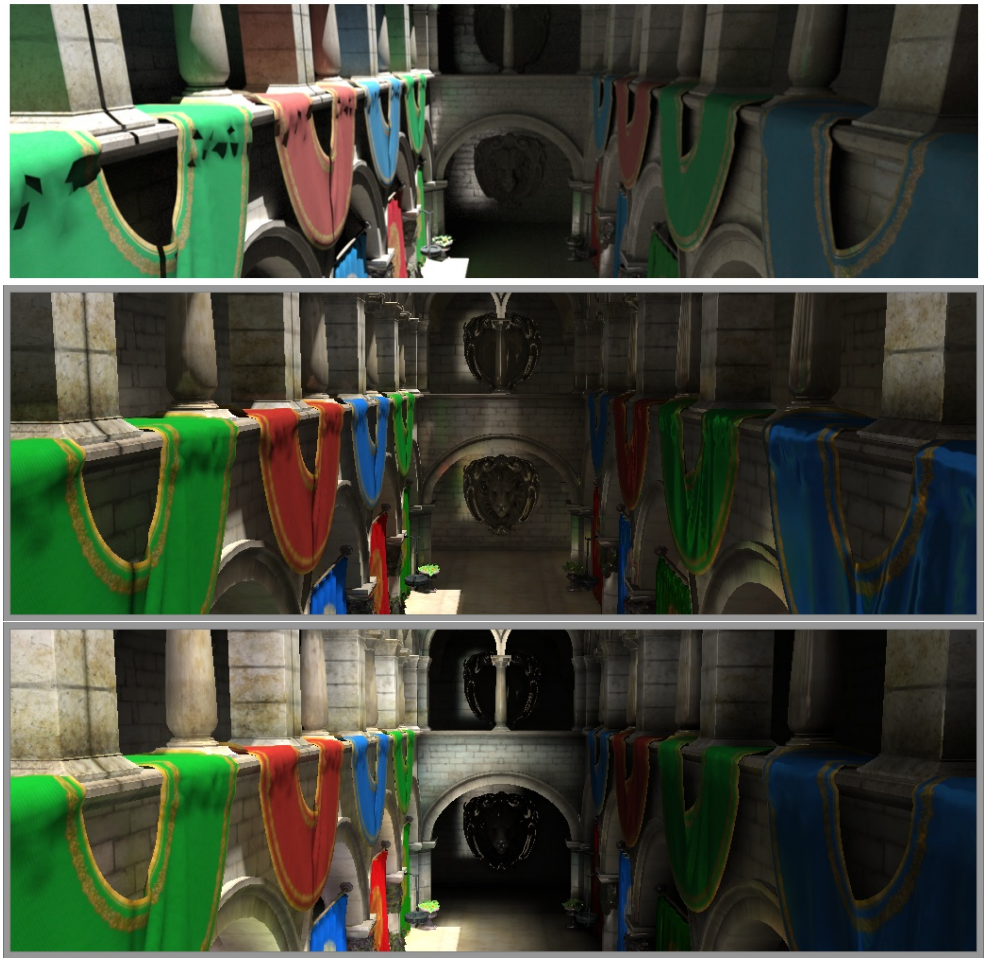


Figure 85: Comparative results between the reference image (upper), isotropic grid (middle) and anisotropic grid (down).

The computation of the approximation to global illumination is actually slower in this algorithm than it is using a regular voxel grid (which launched 5 cones instead of a single one in this case). This could be due to having to decode VPLs and to perform two texture fetches since the results from the two diffuse grids have to be linearly interpolated.

In terms of visual quality, the light propagation volumes results in a more bland lighting and does not capture as much color bleeding or specular reflections as the other algorithms (Figure 86).

In Table 5, the performance measurements from the algorithm described in section 4.4 are presented.

The creation of the atlas is very fast. However, having to read its contents to the CPU in order to create the display list is very slow. By considering the scene as static and performing

Figure 86: Comparative results between the reference image (upper) and the light propagation volumes technique (down).

	Near-field GI
Atlas Creation	0.53
Pixel Display List	73.14
Voxel Grid Creation	0.12
Mipmap	4.61
Indirect Lighting	40.46
Measured FPS	7

Table 5: Performance measurements (in ms) for the near-field indirect illumination algorithm.

voxelization (atlas creation, pixel display list generation and creation of the voxel grid) as a preprocessing step, around 36 frames per second can be obtained. However, the computation of the approximation to indirect lighting is slower than the other voxel cone tracing approaches. This algorithm was not evaluated in terms of visual quality of the result, since there was no way to compare it to the other approaches.

Although no implementation was made for the algorithm described in section 3.5, some images were obtained from the reference implementation provided by NVIDIA. These results can be observed in Figure 87.

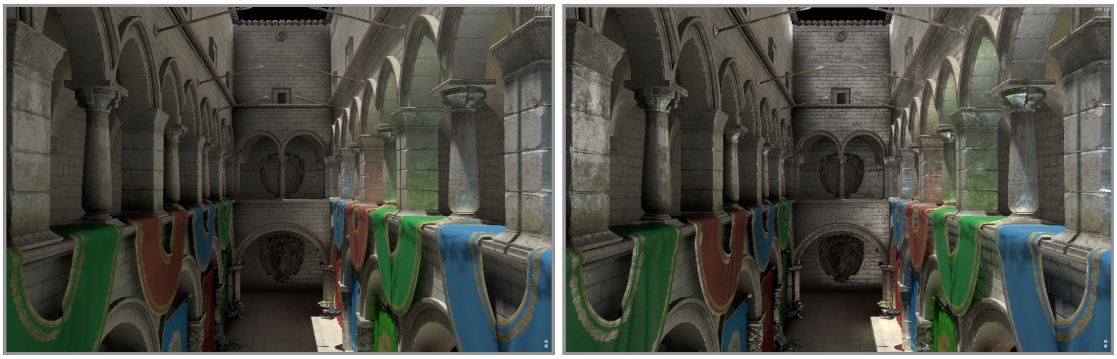


Figure 87: Results from the VXGI implementation provided by NVIDIA. The image on the right has the specular contribution increased by a scale factor.

Although this implementation also uses voxel cone tracing, it seems to better capture some details on the scene, such as the color bleeding in the pillars. It also offers better performance, since around 72 fps were obtained.

5

CONCLUSIONS

Computing global illumination is a very complex process. In reality, light emitters produce an immense number of photons that can bounce in the scene in a near infinite number of paths. In order to be able to do this in real time, approximations need to be computed instead.

The most common rendering algorithms can be subdivided in two classes: ray-based approaches and rasterization approaches.

Ray-tracing uses integral approximations to compute the light paths and generate very plausible results. However, the process is computationally very expensive, being unable to produce results in real-time.

Rasterization ignores indirect light bounces, producing only a direct lighting result. Since only one path exists from the light emitter to the observer, this process is very fast, but the result severely lacks realism. Several effects can be added to improve the visual quality of the rendered images such as SSAO, shadows and fog among others. Still, the result is far from the ray-tracing solutions.

Voxels have been used for multiple purposes before, such as collision detection, fluid simulation or the computation of volumetric effects.

If the indirect light contributions from the surfaces of the objects in the scene are approximated using a volume in space, a voxel-based representation of the scene and a hybrid rendering process can be used to compute an approximation of a single indirect light bounce in real-time.

The most common approach is to compute a voxel-based representation using a process known as voxelization. This representation can then be used to accumulate the indirect light contributions by launching rays through the scene.

Some approaches go even further: instead of a ray, a cone is launched through the scene to gather the indirect light contributions from a hierarchical voxel representation of the scene. This hierarchical voxel structure is sampled at different resolution levels according to the span of the cone.

Other approaches use a binary voxel representation of the scene to encode the presence of geometry and use reflective shadow maps to sample the parameters needed to accumulate the indirect light contributions through the scene.

It has been shown that these algorithms share a similar structure, requiring the scene to be pre-filtered using some kind of voxelization algorithm. The result is a voxel grid that can be encoded in multiple ways, such as a 3D texture, a 2D texture or a sparse voxel octree.

All of these data structures have advantages and disadvantages. Full grids stored in 3D textures waste a lot of memory with empty voxels, but are accessed faster. Sparse voxel octrees are costly to generate and maintain updated, and do not always generate a lower memory footprint than a regular voxel grid. 2D textures are smaller than 3D textures but they are only suitable to store a binary representation of the scene, due to the need to use the texels to store a stack of voxels along the voxelization depth, which in turn decreases the visual quality of result obtained.

More recently, 3D clipmaps have been proposed to encode such types of data structures. The 3D clipmaps seem to match perfectly the voxel cone tracing needs, since they provide higher spatial resolution of voxels near the observer and coarser representations of the scene further away from the camera. Besides the low memory footprint, their fast access and update trough toroidal addressing is also a very good feature for the storage of the voxel data.

Another approach to deal with the wasted memory from regular 3D voxel grids is to use the empty space of the 3D texture storing the voxel grid to encode necessary data to compute other visual effects. An example of an atmospheric effect was presented, which allowed to compute animated smoke in real time by using the voxel grid.

Some research can be made in the following of this work in order to improve the results obtained by the approximation to global illumination or to make a better use of the wasted space of a regular voxel grid.

In particular, the generation of multiple bounces of light could increase the voxel cone tracing visual quality at the expense of performance. The empty space of the voxel grid could be used also for fluid simulation (since in reality, atmospheric effects such as smoke, solid objects and fluids do not occupy the same volume in space). It should also be possible to use the voxelization result to compute collisions, since the voxel grid encodes the presence of geometry in space.

BIBLIOGRAPHY

Allard, Jérémie; Faure, François; Courtecuisse, Hadrien; Falipou, Florent; Duriez, Christian, and Kry, Paul G. Volume contact constraints at arbitrary resolution. *ACM Trans. Graph.*, 29 (4):82:1–82:10, July 2010. ISSN 0730-0301. doi: 10.1145/1778765.1778819. URL http://doi.acm.org/10.1145/1778765.1778819.

Amanatides, John. Ray tracing with cones. *SIGGRAPH Comput. Graph.*, 18(3):129–135, January 1984. ISSN 0097-8930. doi: 10.1145/964965.808589. URL http://doi.acm.org/10.1145/964965.808589.

Asirvatham, Arul and Hoppe, Hugues. Terrain rendering using gpu-based geometry clipmaps. In Pharr, Matt, editor, *GPU Gems 2*, pages 27–45. Addison-Wesley, 2005.

Colbert, Mark and Kriv´anek, Jaroslav. Gpu-based importance sampling. In Nguyen, Hubert, editor, *GPU Gems 3*, pages 459–475. Addison-Wesley, 2008.

Crane, Keenan; Llamas, Ignacio, and Tariq, Sarah. *Real Time Simulation and Rendering of 3D Fluids*, chapter 30. Addison-Wesley, 2007.

Crassin, Cyril. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, UNIVERSITE DE GRENOBLE, July 2011. URL http://maverick.inria.fr/Publications/2011/Cra11. English and web-optimized version.

Crassin, Cyril and Green, Simon. CRC Press, Patrick Cozzi and Christophe Riccio, 2012. URL http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf,ChapterPDF.

Crassin, Cyril; Neyret, Fabrice; Lefebvre, Sylvain, and Eisemann, Elmar. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. URL http://maverick.inria.fr/Publications/2009/CNLE09. to appear.

Crassin, Cyril; Neyret, Fabrice; Sainz, Miguel, and Eisemann, Elmar. *Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In book: GPU Pro*, chapter X.3, pages 643–676. A K Peters, 2010. URL http://maverick.inria.fr/Publications/2010/CNSE10.

Crassin, Cyril; Neyret, Fabrice; Sainz, Miguel; Green, Simon, and Eisemann, Elmar. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum (Proc. of Pacific Graphics 2011)*, 2011. URL http://research.nvidia.com/publication/interactive-indirect-illumination-using-voxel-cone-tracing, NVIDIApublicationwebpage.

Dachsbacher, Carsten and Stamminger, Marc. Reflective shadow maps. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, I3D '05, pages 203–231, New York, NY, USA, 2005. ACM. ISBN 1-59593-013-2. doi: 10.1145/1053427.1053460. URL http://doi.acm.org/10.1145/1053427.1053460.

Daniels, Joel; Silva, Cláudio T.; Shepherd, Jason, and Cohen, Elaine. Quadrilateral mesh simplification. *ACM Trans. Graph.*, 27(5):148:1–148:9, December 2008. ISSN 0730-0301. doi: 10.1145/1409060.1409101. URL http://doi.acm.org/10.1145/1409060.1409101.

Doghramachi, Hawar. Rasterized voxel-based dynamic global illumination. In Engel, Wolfgang, editor, *GPU Pro 4*, pages 155–171. CRC Press, 2013.

Dong, Zhao; Chen, Wei; Bao, Hujun; Zhang, Hongxin, and Peng, Qunsheng. Real-time voxelization for complex polygonal models. In *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, PG '04, pages 43–50, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2234-3. URL http://dl.acm.org/citation.cfm?id=1025128.1026026.

Donnelly, William and Lauritzen, Andrew. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 161–165, New York, NY, USA, 2006. ACM. ISBN 1-59593-295-X. doi: 10.1145/1111411.1111440. URL http://doi.acm.org/10.1145/1111411.1111440.

Eisemann, Elmar and Décoret, Xavier. Single-pass gpu solid voxelization for real-time applications. In *Proceedings of Graphics Interface 2008*, GI '08, pages 73–80, Toronto, Ont.,

Canada, Canada, 2008. Canadian Information Processing Society. ISBN 978-1-56881-423-0. URL http://dl.acm.org/citation.cfm?id=1375714.1375728.

Everitt, Cass. Interactive order-independent transparency, 2001.

Fang, Shiaofen; Fang, Shiaofen; Chen, Hongsheng, and Chen, Hongsheng. Hardware accelerated voxelization. *Computers and Graphics*, 24:200–0, 2000.

Forest, Vincent; Barthe, Loïc, and Paulin, Mathias. Real-time hierarchical binary-scene voxelization. *J. Graphics, GPU, & Game Tools*, 14(3):21–34, 2009.

Fournier, Alain. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, Vancouver, BC, Canada, 11 May 1992.

Han, Charles; Sun, Bo; Ramamoorthi, Ravi, and Grinspun, Eitan. Frequency domain normal map filtering. *ACM Trans. Graph.*, 26(3), July 2007. ISSN 0730-0301. doi: 10.1145/1276377.1276412. URL http://doi.acm.org/10.1145/1276377.1276412.

Hasselgren, Jon; Akenine-Mö ller, Tomas, and Ohlsson, Lennart. *Conservative Rasterization*, pages 677–690. GPU Gems 2. Addison-Wesley Professional, 2005.

Henyey, L. C. and Greenstein, J. L. Diffuse radiation in the galaxy. *Astrophysical Journal*, 93: 70–83, 1941.

Intel, . Embree path tracer. URL https://embree.github.io/.

Jarosz, Wojciech; Jensen, Henrik Wann, and Donner, Craig. Advanced global illumination using photon mapping. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, pages 2:1–2:112, New York, NY, USA, 2008. ACM. doi: 10.1145/1401132.1401136. URL http://doi.acm.org/10.1145/1401132.1401136.

Jensen, Henrik Wann. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag. ISBN 3-211-82883-4. URL http://dl.acm.org/citation.cfm?id=275458.275461.

Kajiya, J. T. and Kay, T. L. Rendering fur with three dimensional textures. *SIGGRAPH Comput. Graph.*, 23(3):271–280, July 1989. ISSN 0097-8930. doi: 10.1145/74334.74361. URL http://doi.acm.org/10.1145/74334.74361.

Kaplanyan, Anton and Dachsbacher, Carsten. Cascaded light propagation volumes for real-time indirect illumination. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-939-8. doi: 10.1145/1730804.1730821. URL http://doi.acm.org/10.1145/1730804.1730821.

Kaufman, Arie and Shimony, Eyal. 3d scan-conversion algorithms for voxel-based graphics. In *Proceedings of the 1986 Workshop on Interactive 3D Graphics*, I3D '86, pages 45–75, New York, NY, USA, 1987. ACM. ISBN 0-89791-228-4. doi: 10.1145/319120.319126. URL http://doi.acm.org/10.1145/319120.319126.

Lafortune, Eric P. and Willems, Yves D. Bi-directional path tracing. In *PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS '93*, pages 145–153, 1993.

Laine, Samuli and Karras, Tero. Efficient sparse voxel octrees. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '10, pages 55–63, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-939-8. doi: 10.1145/1730804.1730814. URL http://doi.acm.org/10.1145/1730804.1730814.

Levoy, Marc. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, July 1990. ISSN 0730-0301. doi: 10.1145/78964.78965. URL http://doi.acm.org/10.1145/78964.78965.

Li, Wei; Fan, Zhe; Wei, Xiaoming, and Kaufman, Arie. Flow simulation with complex boundaries. In Pharr, Matt, editor, *GPU Gems 2*, pages 747–764. Addison-Wesley, 2005.

Lighthouse3d, . Very simple * libs. URL http://www.lighthouse3d.com/very-simple-libs/.

Mie, Gustav. Beitrage zur optik truber medien, speziell kolloidaler metallosungen. *Annalen der Physik*, 330(3):377–445, 1908.

Neyret, Fabrice. Modeling, animating, and rendering complex scenes using volumetric textures. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):55–70, January 1998. ISSN 1077-2626. doi: 10.1109/2945.675652. URL http://dx.doi.org/10.1109/2945.675652.

NVIDIA, . Vxgi. URL https://developer.nvidia.com/vxgi.

NVIDIA, . Nv_conservative_raster, 2014. URL https://developer.nvidia.com/sites/default/files/akamai/opengl/specs/GL_NV_conservative_raster.txt.

NVIDIA, . The basics of gpu voxelization, 2015. URL https://developer.nvidia.com/content/basics-gpu-voxelization.

Perlin, K. and Hoffert, E. M. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3):253–262, July 1989. ISSN 0097-8930. doi: 10.1145/74334.74359. URL http://doi.acm.org/10.1145/74334.74359.

Pham, Tuan Q. and Vliet, Lucas J. Separable bilateral filtering for fast video preprocessing. In *In IEEE Internat. Conf. on Multimedia and Expo, CD1–4*, pages 1–4. IEEE, 2005.

Schwarz, Michael and Seidel, Hans-Peter. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph.*, 29(6):179:1–179:10, December 2010. ISSN 0730-0301. doi: 10.1145/1882261.1866201. URL http://doi.acm.org/10.1145/1882261.1866201.

Shopf, Jeremy. Mixed resolution rendering. Presentation, Game Developers Conference 2009, 2009. URL http://developer.amd.com/wordpress/media/2012/10/ShopfMixedResolutionRendering.pdf.

Sugihara, Masamichi; Rauwendaal, Randall, and Salvi, Marco. Layered reflective shadow maps for voxel-based indirect illumination. pages 117–125.

Tanner, Christopher C.; Migdal, Christopher J., and Jones, Michael T. The clipmap: A virtual mipmap. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 151–158, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi: 10.1145/280814.280855. URL http://doi.acm.org/10.1145/280814.280855.

Thiedemann, Sinje; Henrich, Niklas; Grosch, Thorsten, and Müller, Stefan. Voxel-based global illumination. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 103–110, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0565-5. doi: 10.1145/1944745.1944763. URL http://doi.acm.org/10.1145/1944745.1944763.

Thiedemann, Sinje; Henrich, Niklas; Grosch, Thorsten, and Müller, Stefan. Real-time near-field global illumination based on a voxel model. In Engel, Wolfgang, editor, *GPU Pro 3*, pages 209–229. A K Peters, 2012.

Toksvig, Michael. Mipmapping normal maps. *J. Graphics Tools*, 10(3):65–71, 2005. URL http://dblp.uni-trier.de/db/journals/jgtools/jgtools10.html#Toksvig05.

UnrealEngine, . Light propagation volumes. URL https://docs.unrealengine.com/latest/INT/Engine/Rendering/LightingAndShadows/LightPropagationVolumes/index.html.

Veach, Eric and Guibas, Leonidas J. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. doi: 10.1145/258734.258775. URL http://dx.doi.org/10.1145/258734.258775.

Vos, Nathan. Volumetric light effects in killzone: Shadow fall. In Engel, Wolfgang, editor, *GPU Pro 5*, pages 127–147. CRC Press, 2014.

Whitted, Turner. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980. ISSN 0001-0782. doi: 10.1145/358876.358882. URL http://doi.acm.org/10.1145/358876.358882.

Williams, Lance. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, August 1978. ISSN 0097-8930. doi: 10.1145/965139.807402. URL http://doi.acm.org/10.1145/965139.807402.

Zhang, Long; Chen, Wei; Ebert, David S., and Peng, Qunsheng. Conservative voxelization. *Vis. Comput.*, 23(9):783–792, August 2007. ISSN 0178-2789. doi: 10.1007/s00371-007-0149-0. URL http://dx.doi.org/10.1007/s00371-007-0149-0.