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 greater objectives in computer graphics is to be able to generate fotorealistic images and do it in real time. Unfortunately the actual 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 fotorealistic results but lacking performance, which makes them impossible to compute at interactive frame rates with the computational power available in the present.

Over the last years, some new techniques 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. These techniques use a volumetric pre-filtered representation of the scene and a rendering algorithm based on cone tracing to compute an approximation to global illumination in real time.

What is pretended through this study is an analysis on the practicability of such algorithms in real-time applications and apply the new capabilities of the OpenGL API to simplify/optimize the implementation of these algorithms.

RESUMO

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

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

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 interactiva. Estas técnicas usam uma pré-filtragem da cena usando uma representação volumétrica da cena e um algoritmo baseado em *cone tracing* para calcular uma aproximação da iluminação global em tempo real.

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/optimizar a sua implementação.

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).

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).

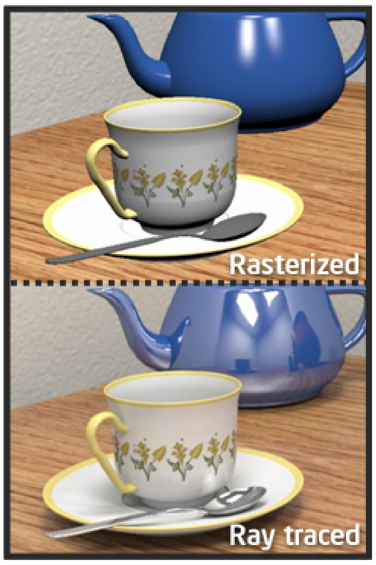

Figure 1: Rasterization vs Ray tracing. *Source:* http://www.cs.utah.edu/ jstratto/state_of_ray_tracing/

To reach fotorealism, several algorithms have been proposed, such as recursive ray tracing (Whitted, 1980), bi-directional pathtracing (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).

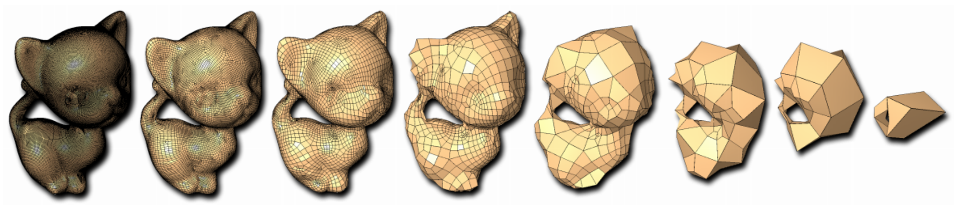

Figure 2: Geometry simplification. Information about the geometry is lost with an increasing level of filtering. *Source:* Daniels et al. (2008)

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).

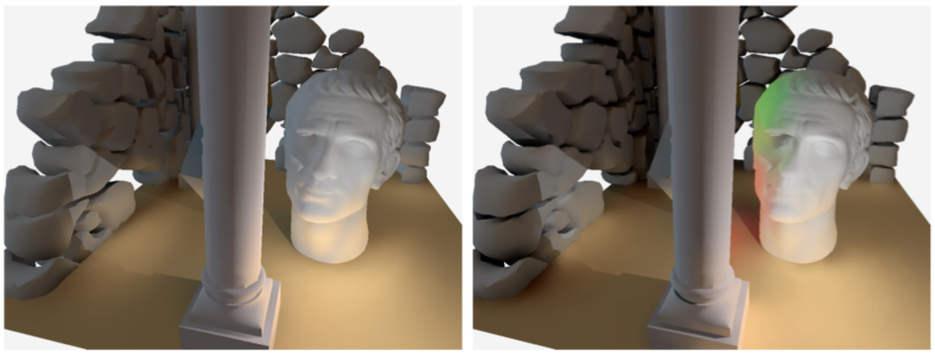
Figure 3: Indirect illumination on a scene with a hidden object behind the column. In the left image, only objects in camera space are taken into account and thus the hidden objects are disregarded since they are not visible by the current camera. *Source:* Thiedemann et al. (2011)

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 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.

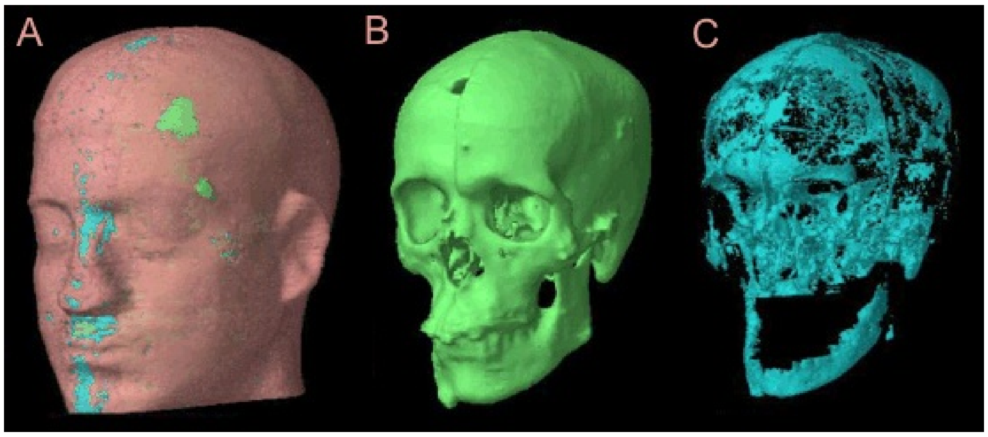

Figure 4: Voxels used to view medical data. *Source:* URL

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 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 several improvements for future work.

2

RELATED WORK

2.1 SHADOW MAPPING

Shadow Mapping is a method published in 1978 (Williams, 1978) that allows to add shadows to a 3D scene. Finding if a fragment is in shadow is the same as finding if the fragment is visible from the light.

The basic shadow mapping algorithm consists in two separate passes. First, the scene is rendered from the light point of view and a texture storing the depth of the objects in the scene is created. This texture represents which pixels are lit and how far those pixels are from the light. Then, it is possible to test if a fragment is visible or not from the light by finding its coordinate as seen from the light and comparing it with the depth texture previously created (shadow map).

One of the main problems of this algorithm is that it greatly depends on the resolution of the shadow map. Some common optimizations are to add a small bias when comparing the depth of the fragment with the shadow map, or using Percentage Close Filtering (PCF - Reeves et al. (1987)). However, many other optimizations and algorithms are available in order to add shadows to a scene. Some of the more popular techniques are Cascaded Shadow Maps (Engel, 2006), Variance Shadow Maps (Donnelly and Lauritzen, 2006), Exponential Shadow Maps (Annen et al., 2008), or Light Space Perspective Shadow Maps (Wimmer et al., 2004).

2.2 DEFERRED RENDERING

In forward rendering, the vertices of the objects present in the scene are transformed and lighting computations are performed for every fragment generated after rasterization. This approach brings a problem with highly complex scenes. Since objects can be covered by several objects, shading might be computed multiple times for nothing because only the closest fragment to the camera counts. Another problem is that forward rendering does not scale well when multiple light sources are added to the scene. The fragment shader computes lighting for every light, even if the light is far away from the camera and its area of effect does not reach the corresponding fragment.

Deferred shading is an idea that was first referenced in the paper ”The triangle processor and normal vector shader: a VLSI system for high performance graphics” (Deering et al., 1988), but the actual technique using G-Buffers was only introduced later in a paper called ”Comprehensible rendering of 3-D shape” (Saito and Takahashi, 1990).

Deferred shading is a technique that allows to separate geometry computations from lighting calculations. It can be divided in two steps: A first pass, called the Geometry Pass, in which no shading is performed and a second pass, called the Lighting Pass, that actually performs the lighting computations.

In the Geometry Pass, the geometric transformations are applied to the objects in the vertex shader, but instead of sending the attributes to the fragment shader for lighting computations, they are written into what is known as the Geometry Buffer (G-Buffer). The G-Buffer is simply a group of several 2D textures that will store the vertex attributes, usually positions or depth, normals and materials (one texture per vertex attribute). The attributes are written all at once by using a feature available in OpenGL called Multiple Render Targets (MRT). Thanks to the depth test, at the end of this pass the textures in the G-Buffer only contain the processed vertex attributes for the fragments closer to the camera.

The Lighting Pass consists in the rendering of a full-screen quad and sample all the pixels in the G-Buffer, performing lighting computations in the same way that they were done during forward rendering. Since the G-Buffer only contains the fragments closer to the camera, the lighting calculations are effectively only done once for each pixel on the screen. Another way is to render a sphere (for point lights) or a cone (for spot lights) and only perform the lighting calculations on the area relevant to the light source, discarding every fragment that is not inside its area of influence.

The main advantage of deferred rendering is the ability to handle multiple light sources without a significant performance hit by allowing the lighting computations to be calculated only for the pixels that the light actually affects. The greater disadvantage of this algorithm is its lack of support to transparent materials, as well as the need to store more data in the G-Buffer to allow the use of multiple materials.

2.3 REFLECTIVE SHADOW MAPS

Similarly to the deferred shading technique, Reflective Shadow Maps (RSM - Dachsbacher and Stamminger (2005)) attach multiple render targets (MRT) to the shadow mapping output, extending the classical shadow mapping algorithm in order to view each pixel of the shadow map as a virtual point light that emits light to the scene.

The scene is rendered from the point of view of the light and world-space position, normal and flux are stored in multiple 2D textures attached using the MRT capability offered by OpenGL. When the scene is rendered from the camera point of view, the pixels from the RSM can be considered as a source of indirect illumination. By combining the attributes previously stored in the 2D textures, it is possible to generate the irradiance of each pixel in the shadow map. The sum of the irradiance of all pixels in the shadow map represents the indirect lighting contribution in the scene.

Since a shadow map can contain a great number of pixels, only a randomly chosen subset will be used to sample indirect lighting for each pixel on the screen, in order to keep rendering feasible in real-time.

2.4 RAY TRACING

In nature, a light source emits light rays that travel trough 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, the rays start from the view plane and are launched into the scene.

The first ray tracing algorithm was called ray casting (Appel, 1968). The main idea of the ray casting algorithm is to shoot rays from the view plane and terminate the traversal of the ray when it intersects some object on the scene. It allows the computation of the shading of the objects, but it does not mimic nature correctly since rays do not reflect and refract around the scene.

To address this issue, Recursive Ray Tracing was proposed (Whitted, 1980). This algorithm extends the ray casting approach by shooting secondary rays after the primary ray has encountered an object. By shooting a ray in the reflection direction it is possible to generate mirror-like materials and a refraction ray will create the effect of transparent materials. The algorithm is recursive, which means it is possible to continue the traversal of the rays after hitting multiple objects, rendering multiple reflections.

Apart from the rendering time, these 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 trough the center of the pixels. However this increases even more the amount of computation needed for the algorithm.

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 trough the screen 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 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.

Since then, multiple algorithms have been proposed to speed the rendering process or generate a higher quality rendering such as Bi-Directional Path Tracing (Lafortune and Willems, 1993), Photon-Mapping (Jensen, 1996) or Metropolis Light Transport (Veach and Guibas, 1997).

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).

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. The algorithm starts by casting rays from the view plane into the volume, sampling it at equally spaced intervals. The data is interpolated at each sampling point since the volume is usually not aligned with the camera (usually using trilinear interpolation). The interpolated scalar values are then mapped to optical properties by using a transfer function, forming an RGBA color value. The color values are composited along the ray using front-to-back or back-to-front alpha blending until the ray exits the volume.

2.5 VOXELIZATION

Voxelization or 3D scan conversion is the process of mapping a 3D object made of 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.

On surface voxelization, only the voxels that are touched by the triangles are set, 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).

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 (0 or 1), in which case it is usually referenced as a binary voxelization.

3

REAL-TIME VOXEL-BASED GLOBAL ILLUMINATION ALGORITHMS

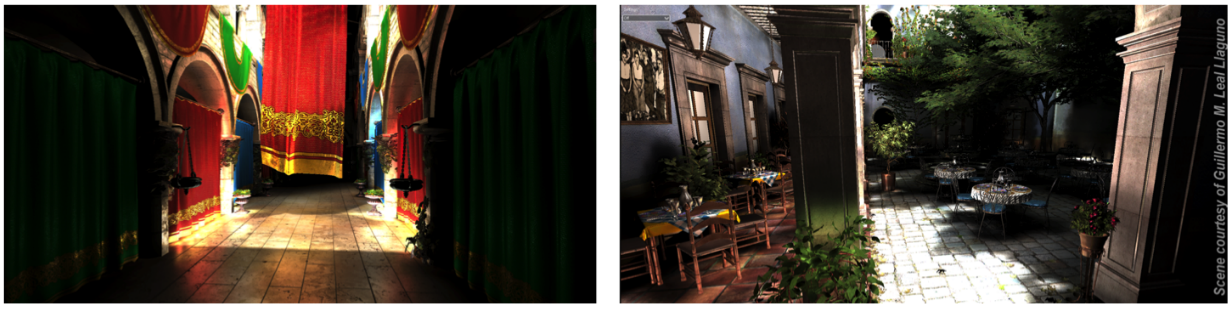

Figure 5: Voxel-based Global Illumination. *Source:* Crassin et al. (2011)

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 intersect with triangles) is too expensive for real time applications.

Voxels have many benefits when compared with triangles, such as their ability to easily handle transparency, reflections and refraction by using volume ray casting (section 2.4), thanks to their volumetric representation. They are also cheaper to intersect than triangles, which makes them a good choice for ray tracing.

Voxels can also be stored in an octree structure, which can be used to accelerate ray tracing and store geometry on a compressed format at the same time.

But they also have their disadvantages, the greater of which being the memory consumption. Voxel data sets tend to be considerably greater than polygon data (Foley et al., 1990). Also, using a data structure such as an octree to store the voxel data makes it difficult to handle dynamic objects, since the octree needs to be updated whenever an object changes places or form.

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 global illumination in real time have been introduced. These algorithms are very similar in their structure, as will be demonstrated in the following chapter. 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 (Figure 5).

There are several algorithms and data structures to perform each of these steps, each of them with advantages and disadvantages, but this dissertation will be focused on the recent algorithms for computing global illumination in real time.

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, we need a way to pre-filter the appearance of the objects on the scene. Pre-filtering not just the textures but the geometry as well will provide a scalable solution to compute 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 an overall density distribution and an overall reflectance function to approximate the interaction of light within this volume (Figure 6). This observation, that permits a pre-filtering of the scene geometry into a volumetric representation, was made by Perlin (Perlin and Hoffert, 1989) and Kajiya and Kay (Kajiya and Kay, 1989).

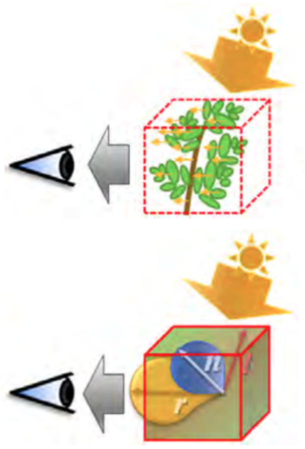
Figure 6: Voxel Lighting. *Source:* Crassin (2011)

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. 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 us to obtain a multiresolution representation of the voxel grid based on mipmapping, making it possible to automatically control the level of detail by sampling to different mipmap levels of the voxel grid.

The general idea of this technique is to pre-filter the scene using a voxel representation (3.1.1) and store the values in a sparse octree structure in order to get a hierarchical representation of the scene (3.1.2). 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 (3.1.3), 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 7, 3.1.4).

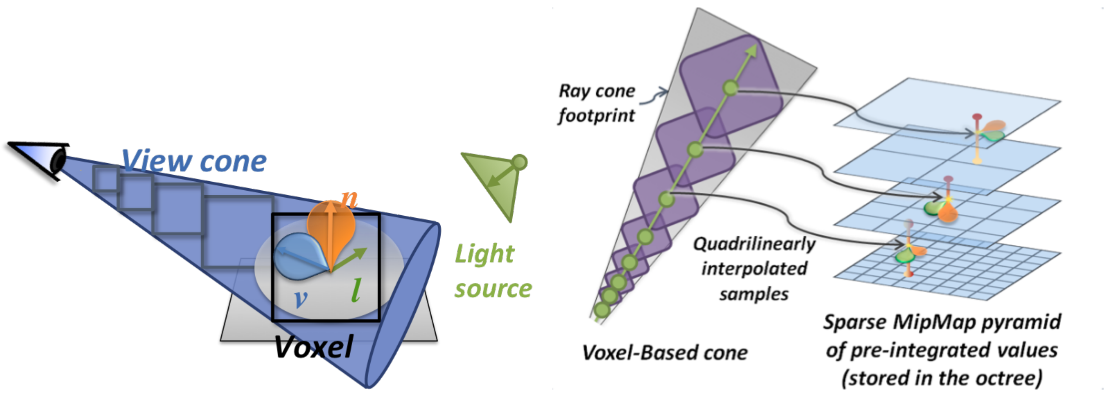
View cone
n
v
l
Voxel
Light source
Ray cone footprint
Quadrilinearly interpolated samples
Voxel-Based cone
Sparse MipMap pyramid of pre-integrated values (stored in the octree)

Figure 7: 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 eventually turns this operation too computationally expensive, ruining the performance of the algorithm.

The update of dynamic elements in this kind of data structures is a problem that still needs to be solved and new approaches to update these structures in a faster way need to emerge, or new data structures that are more rapidly updated while keeping the advantages offered by octrees.

3.1.1 *Voxelization*

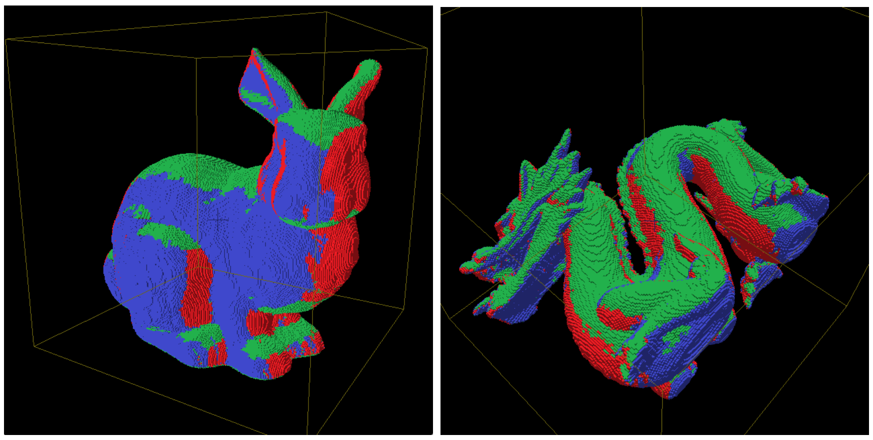
Figure 8: Voxelization. Red: projection along x-axis. Green: projection along y-axis. Blue: projection along z-axis

Voxelization approaches can be separated into two different types: surface voxelization and solid voxelization (section 2.5). For the scope of our problem, surface voxelization is preferred since light is reflected at the surface of the materials.

Since the main objective is to compute indirect illumination in real-time, achieving very fast voxelization of a triangle-based representation is critical. Static geometry can be voxelized as a pre processing pass, but dynamic objects need to be voxelized in every frame.

This surface voxelization algorithm uses the GPU hardware rasterizer and the new image load-/store interface exposed by OpenGL 4.2 to achieve a real-time voxelization of the triangles of the mesh, performed during a single rendering pass (Figure 8).

The key element for this voxelization process 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 merely the one (chosen from the three main axes of the scene) that maximizes the surface of the projected triangle.

Based on this observation, the voxelization process can be divided in several steps (Figure 9).

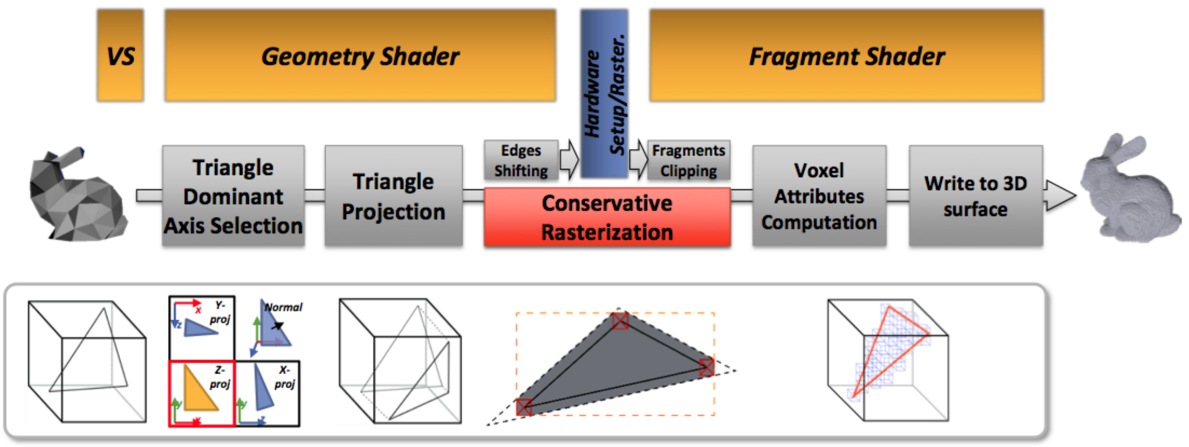

Figure 9: Voxelization Pipeline. *Source:* Crassin and Green (2012)

First, the dominant axis of the triangle normal is 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 will be used for this purpose, where 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 $\mathbf{n}$ the triangle normal and $\mathbf{v}_{\{x,y,z\}}$ the three main axis of the scene.

Once the dominant axis of the triangle normal has been selected, the triangle is projected along this axis. This projection is simply a classical orthographic projection, setting its limits so that the projection covers the entire scene to be voxelized, and this is still done in the geometry shader by swizzling the vertices of the triangle to match this projection. A very important detail is the need to disable the depth test in order to prevent early culling.

After passing by the geometry shader, the triangle is fed into the standard setup and rasterization pipeline to perform 2D scan conversion (rasterization). If the triangle is fed right after projecting it along the dominant axis, a problem still subsists. During rasterization, each triangle generates multiple 2D fragments corresponding to the pixels intersected by the triangle. The problem is that only the coverage of the pixel center is tested during the rasterization process. This can cause some artifacts on the voxelization process (Figure 10).

Thus, to ensure a proper voxelization, we need to guarantee that every pixel touched by a triangle will generate a 2D fragment. One way to do this is to use multisampling, generating a fragment for any of the multisampling locations covered by a triangle. However, this method still does not guarantee a proper voxelization. A more accurate way to ensure a conservative voxelization is to use a technique known as conservative rasterization, and it corresponds to the third step of the voxelization algorithm.

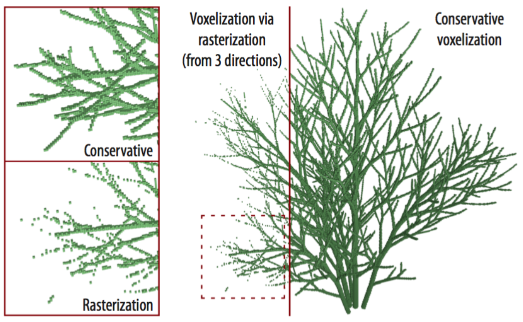
Conservative
Rasterization
Voxelization via rasterization (from 3 directions)
Conservative voxelization

Figure 10: Conservative Voxelization. *Source:* Schwarz and Seidel (2010)

This process is based on the work of Hasselgren et al. (2005). The general idea is to calculate a bounding box for the triangle and slightly shift the edges of the triangle outwards, expanding it. The bounding box can then be used later on the fragment shader to discard the excess fragments generated in the rasterization pass by enlarging the triangle (Figure 11).

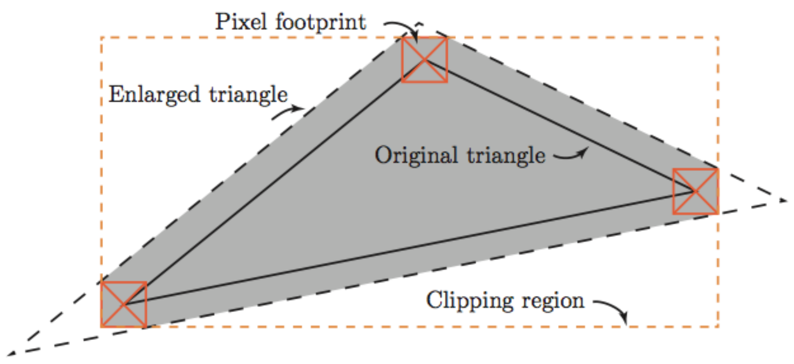

Figure 11: Triangle Expansion in Conservative Rasterization. *Source:* Crassin and Green (2012)

After rasterization, voxel attributes are computed within the fragment shader. These attributes are any useful attribute we would want to store per voxel. Since the main objective is to compute global illumination, shading parameters such as albedo and normals need to be stored. Also, the 3D position inside the voxel grid must be determined in order to store these attributes in the correct voxel.

This generates voxel fragments. A voxel fragment is the 3D generalization of the 2D fragment and corresponds to a voxel intersected by a triangle.

Once the voxel fragments are generated, they can be written into a buffer using image load/-store operations, generating a voxel fragment list. This voxel fragment list is a linear vector of entries stored inside a preallocated buffer object. It contains several arrays of values, one containing the 3D coordinate of each voxel fragment, and all the others containing the attributes we want to store for each voxel. To manage this list, a counter of the number of fragments of the list is maintained as a single value stored inside another buffer object and updated with an atomic counter.

Since we want to generate fragments corresponding to the maximum resolution of the octree, 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 access is used to write the voxel data.

3.1.2 *Sparse Voxel Octree*

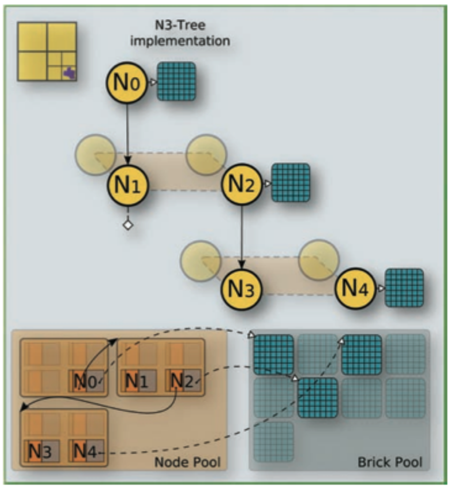

Figure 12: Sparse Voxel Octree Structure. *Source:* Crassin et al. (2010)

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.

The data structure chosen 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 LOD control mechanism.

The sparse voxel octree is a very compact pointer-based structure (Figure 12). 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). 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 13). 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.

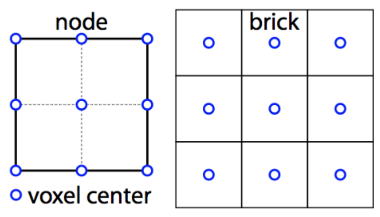

Figure 13: Voxel Brick. *Source:* Crassin et al. (2011)

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 14). After its creation, voxel fragments (3.1.1) are written in the leaves and mipmapped into the interior nodes of the tree (3.1.3).

The subdivision of the octree is done in three steps (Figure 15). First, the nodes that need to be subdivided are flagged using one thread per entry on the voxel fragment list. Each thread

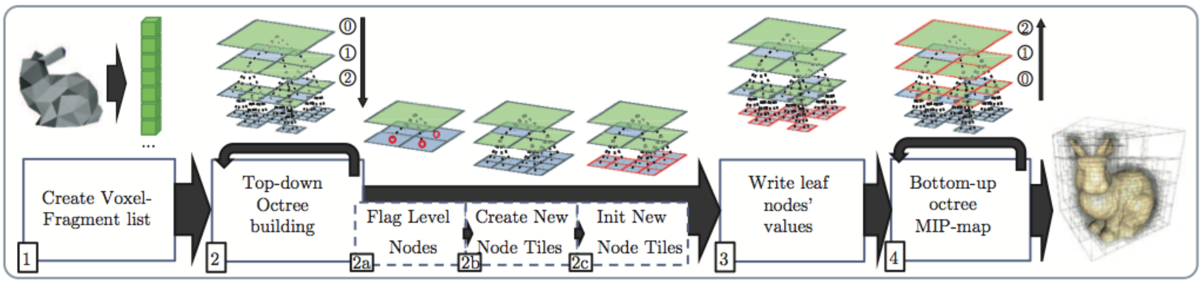

Figure 14: Steps for the creation of the sparse voxel octree structure. *Source:* Crassin and Green (2012)

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. Since allocations can occur at the same time, they are controlled using a shared atomic counter.

The last step is to initialize the new nodes to null child pointers. This is performed in a separate pass to allow using one thread per each node of the new octree level.

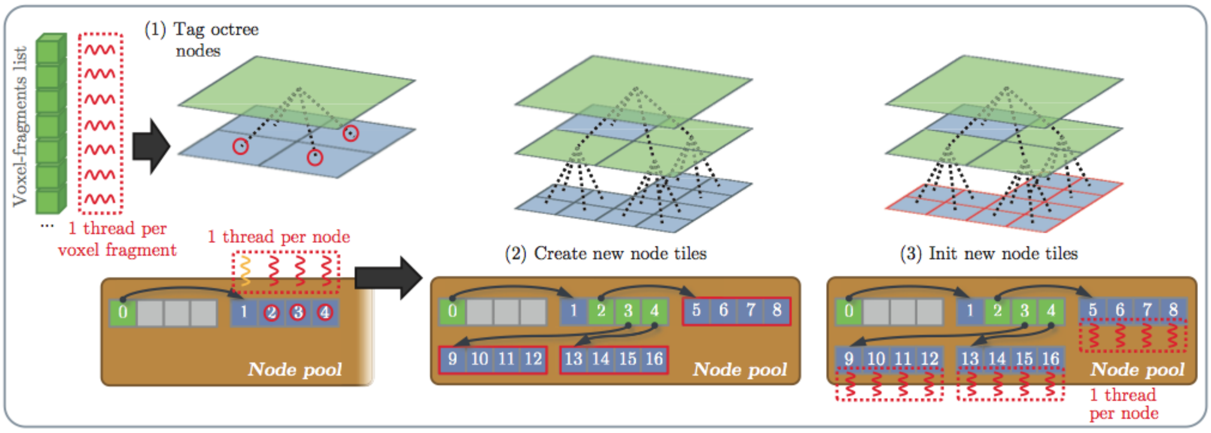

Figure 15: Node Subdivision and Creation. *Source:* Crassin and Green (2012)

Once the octree is built, the leaves of the tree need to be filled with the voxel fragments. This is achieved using one thread per entry on the voxel fragment list and since multiple voxel fragments may try to write their attributes in the same destination, atomic operations are needed. All values falling in the same destination voxel will be averaged. To do so, all values are added using an atomic add operation, updating at the same time a counter, so that the summed value can then 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 are mipmapped into the interior nodes of the tree (3.1.3).

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 is used in order to differentiate each type of object and prevent overwriting of static nodes and bricks.

3.1.3 *Mipmapping*

In order to generate an hierarchic 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. This results on a 3^3 Gaussian weighting kernel, which is an optimal reconstruction filter in this case (Crassin et al., 2011) (Figure 16).

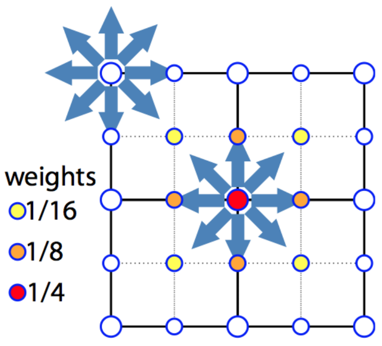

Figure 16: Mipmapping Weighting Kernel. *Source:* Crassin et al. (2011)

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 17). 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.

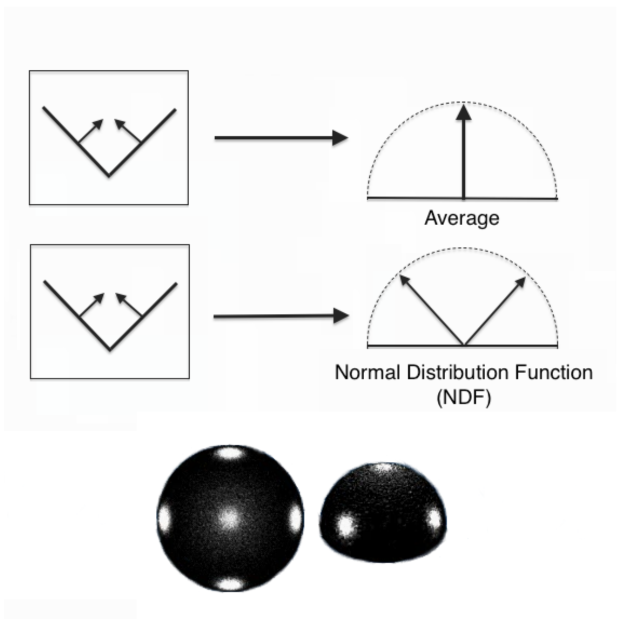

Figure 17: Normal Distribution Function (NDF).

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 18).

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

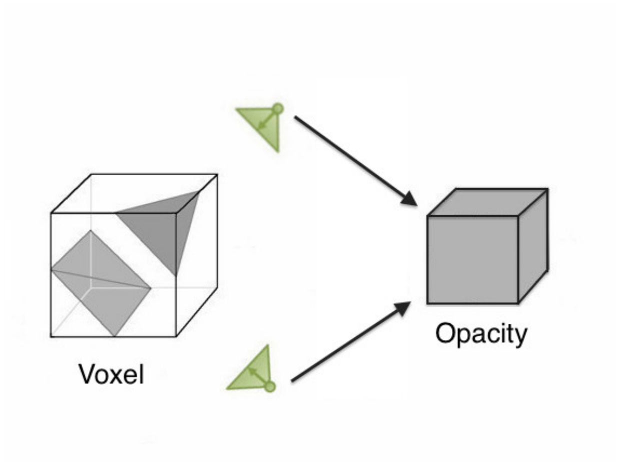

Figure 18: Opacity is stored as a single value inside a voxel, causing a lack of view dependency.

order to properly account for its visibility.

3.1.4 *Voxel Cone Tracing*

Before computing global illumination, information about the lighting needs to be added to the sparse voxel octree. The scene is rasterized from all light sources in order to determine 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.

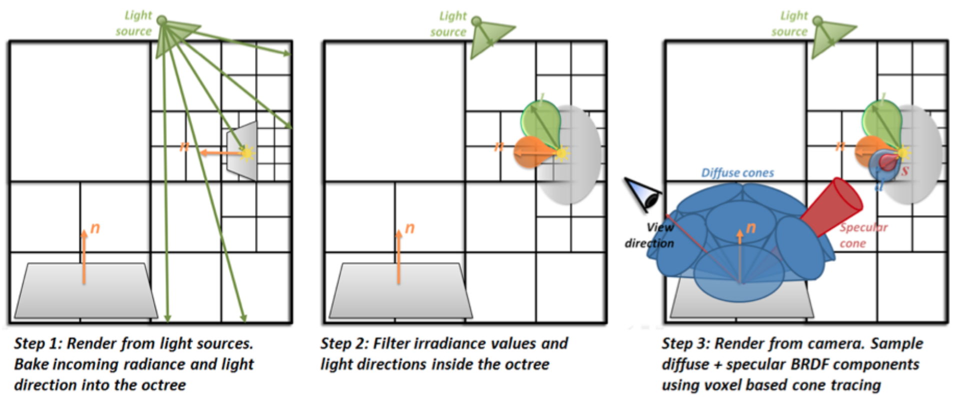
Light source
n
Step 1: Render from light sources. Bake incoming radiance and light direction into the octree
Light source
n
Step 2: Filter irradiance values and light directions inside the octree
Light source
Diffuse cones
View direction
n
Specular cone
s
d
Step 3: Render from camera. Sample diffuse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

Figure 19: 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 perform a final gathering Jensen (1996)) and collect illumination on the octree in order to estimate the diffuse contribution for the indirect illumination.

A single cone is launched in the reflected direction to capture the specular contribution of the indirect illumination. Finally, global illumination is obtained by combining direct and indirect illumination (Figure 19).

This voxel cone tracing pass (Figure 20) is slightly different than true cone tracing (section 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 according to the classical emission-absorption optical model (Max, 1995; Hadwiger et al., 2006).

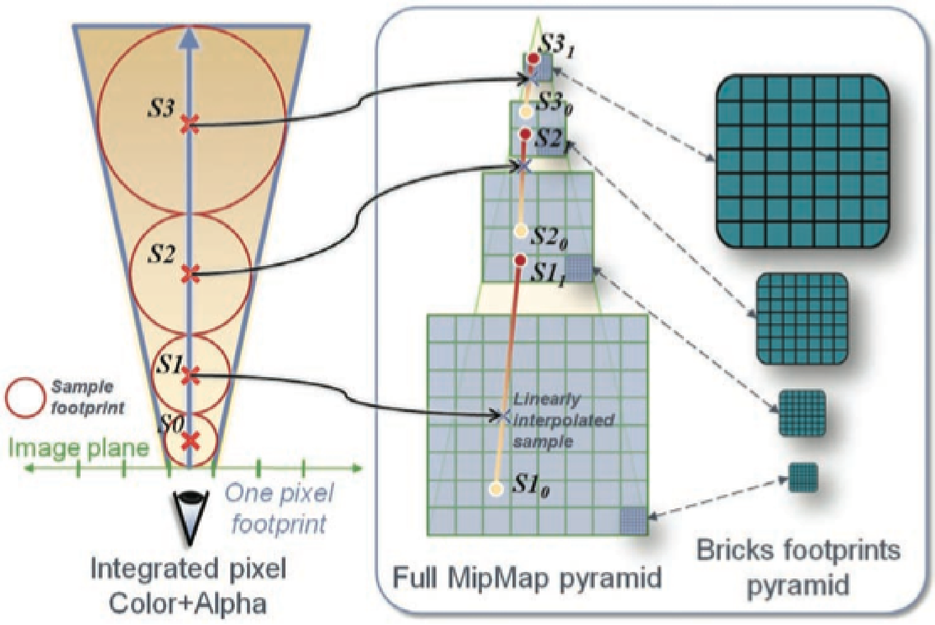

Figure 20: Voxel Cone Tracing. *Source:* Crassin et al. (2010)

The classical emission-absorption optical model is based on geometrical optics. It assumes that light propagates along a straight line when there is no interaction with matter. When light interacts with matter, it can be absorbed, scattered or emitted by the material. This model neglects scattering, representing only local light emission and absorption.

Light emission (amount of energy emitted by the material) and absorption (amount of energy that is absorbed by the material) affect the amount of light energy along a ray. This light energy is described by its radiance $I(x, \omega)$, defining the radiation field in any point x, given the light direction ω.

$$I(x, \omega) = \frac{\mathrm{d}Q}{\mathrm{d}A \cos\theta \mathrm{d}\Omega \mathrm{d}t}$$

The emission-absorption optical model leads to the volume rendering integral:

$$I(D) = I_0 \mathrm{e}^{-\int_{s0}^{D} \kappa(t)\mathrm{d}t} + \int_{s0}^{D} q(s) \mathrm{e}^{-\int_{s}^{D} \kappa(t)\mathrm{d}t} \mathrm{d}s$$

with κ the absorption coefficient, q the emission and integration from the entry point into the volume $s = s0$ to the exit point toward the camera $s = D$.

The volume rendering integral can be evaluated incrementally, with either a front-to-back or a back-to-front compositing scheme. The preferred compositing scheme in this case is the front-to-back, since it allows to stop the evaluation when the accumulated transparency reaches zero.

The front-to-back compositing scheme can be expressed as:

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

with α the opacity defined as $\alpha = 1 - T$ and T the transparency.

This voxel cone tracing technique can also be used to approximate 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 will then be 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 21).

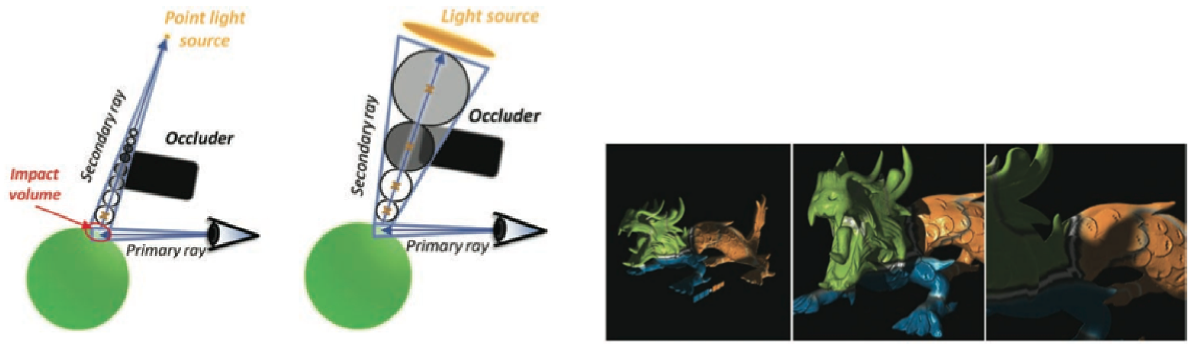

Figure 21: Estimating Soft Shadows trough Voxel Cone Tracing. *Source:* Crassin (2011)

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 the corresponding MIP-map level is chosen to estimate the result (Figure 22). Since the sparse voxel octree does not need to be traversed so deeply, the rendering becomes faster for an increased introduction of blur.

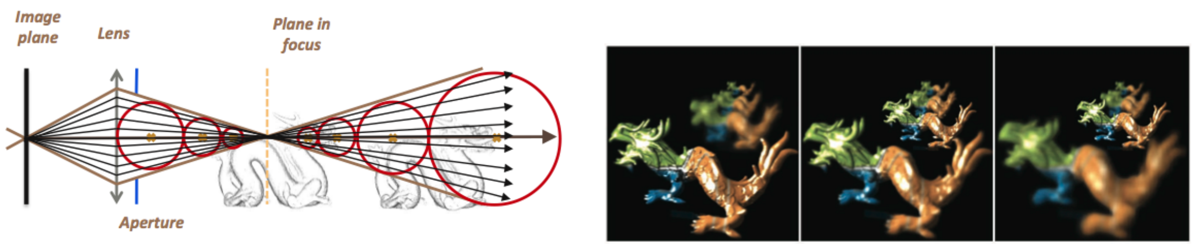

Figure 22: Estimating Depth of Field Effects trough Voxel Cone Tracing. *Source:* Crassin (2011)

Direct Illumination Injection

The scene is rendered from the light's view (using rasterization) and outputs a world position, generating a Reflective Shadow Map (section 2.3). 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 difficulties of this process is that voxels are repeated for neighboring bricks in order to allow using fast hardware filtering. The approach selected to solve this problem is to perform 6 passes, two for each axis (Figure 23).

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.

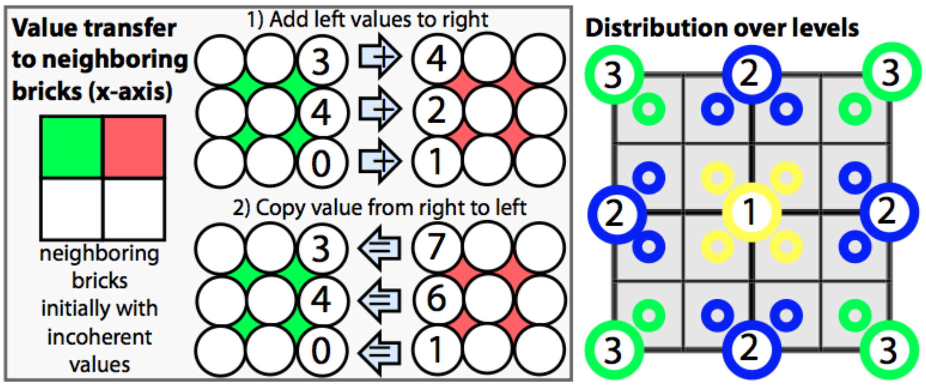
Value transfer to neighboring bricks (x-axis)
neighboring bricks initially with incoherent values
1) Add left values to right
3
4
0
4
2
1
2) Copy value from right to left
3
4
0
7
6
1
Distribution over levels
3
2
3
2
1
2
3
2
3

Figure 23: 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 mip-mapped 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 23).

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 splatting the photons. 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.

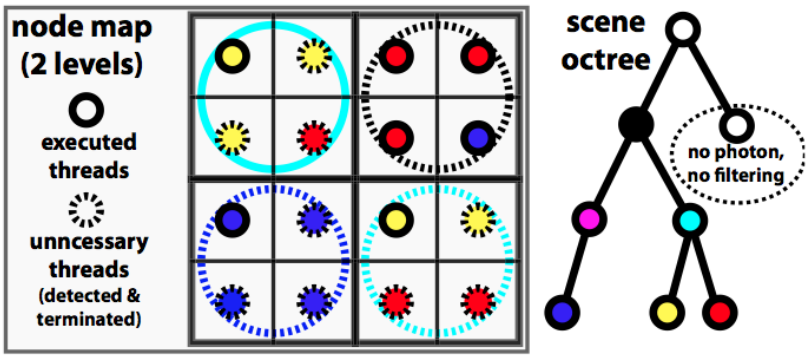

Figure 24: Node Map. *Source:* Crassin et al. (2011)

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 24). This map is a Mip-map 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 derives from 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 25). It is built during the mip-mapping process, when building or updating the sparse voxel octree with the lighting information. 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.

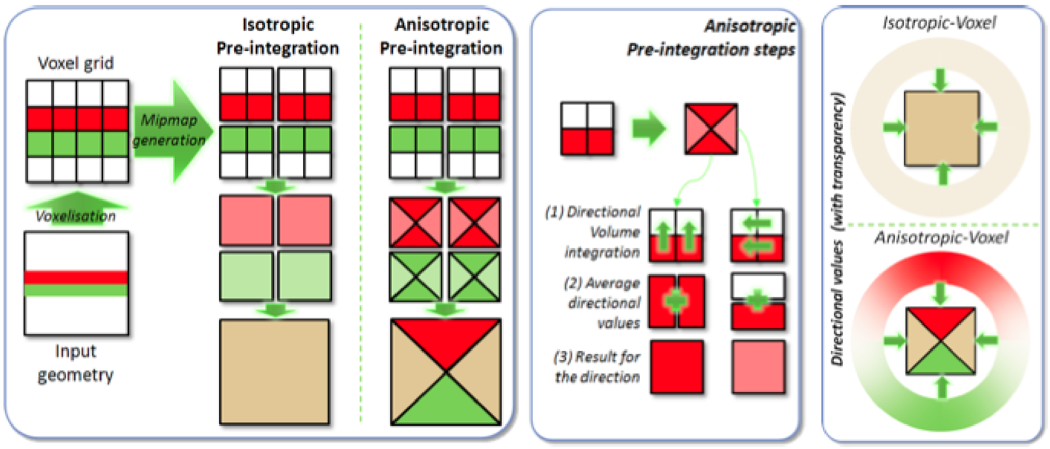
Voxel grid
Mipmap generation
Voxelisation
Input geometry
Isotropic Pre-integration
Anisotropic Pre-integration
Anisotropic Pre-integration steps
(1) Directional Volume integration
(2) Average directional values
(3) Result for the direction
Isotropic-Voxel
Anisotropic-Voxel
Directional values (with transparency)

Figure 25: Anisotropic Voxel Representation. *Source:* Crassin et al. (2011)

Since 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, memory consumption is only increased by 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.

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 26).

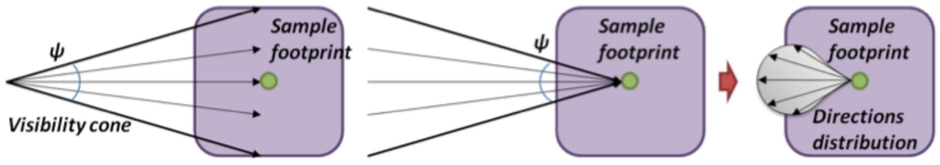

Figure 26: Directions distribution. *Source:* Crassin et al. (2011)

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 global illumination in real-time 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 inside the near-field at interactive frame-rates (Thiedemann et al., 2011).

3.2.1 *Voxelization*

The voxelization method used creates a binary voxelization of the scene (Eisemann and Décoret, 2006). First of all, the models must be mapped to a texture atlas. In this way, by rendering the model into its corresponding texture atlas, a discretization of the surface of the object is created and then used to generate a voxel grid (Figure 27). It borrows some ideas from depth-peeling voxelization, but instead of peeling an object and saving its layers to textures before voxelization, it renders the complete object to a single atlas texture image on a single rendering pass.

The bits of the RGBA channels of the texture atlas are used to encode the world-positions, producing a binary voxel grid. However, it is also possible to encode any type of data (e.g. radiance, normals) by using a 3D texture, creating a multivalued voxel grid.

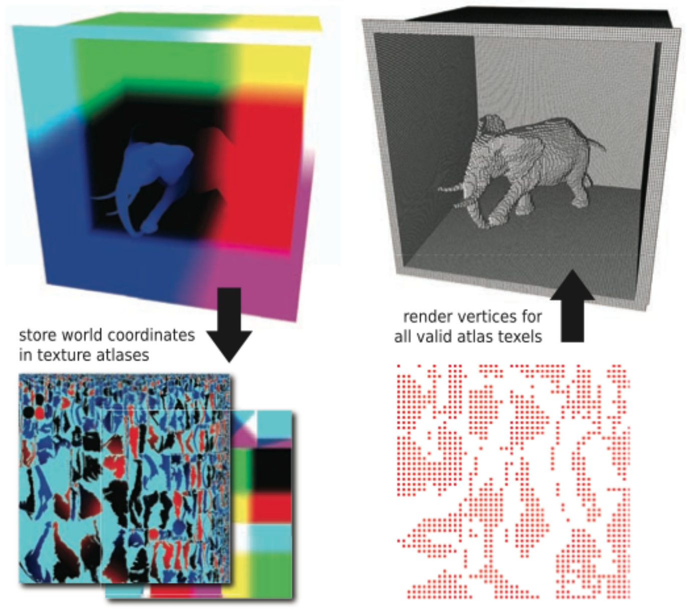
store world coordinates in texture atlases
render vertices for all valid atlas texels

Figure 27: Binary Voxelization. *Source:* Thiedemann et al. (2012)

This algorithm presents several advantages:

- Independent of the depth complexity of the scene.
- Does not exhibit problems with polygons parallel to the voxelization direction.
- Applicable to moderately deforming models. Strong deformations can corrupt the mapping from the object to the texture atlas. If deformations are known in advance, it is possible to use different atlas mappings for each stage of the deformation.
- Good performance, being suited to real-time applications.

3.2.2 *Binary Voxelization*

The algorithm can be divided in two steps. First, all objects are rendered, storing their world-space positions to one or multiple atlas textures. However, having one texture atlas for each object allows for a flexible scene composition, since objects can be added or removed without having to recreate the whole atlas.

Before inserting the voxels into the grid, it is necessary to set the camera on the scene. Its frustum will define the coordinate system of the voxel grid.

Then, for every valid texel in the texture atlas, a vertex is generated and inserted into a voxel grid using point rendering. In order to identify valid texels, the texture atlas is cleared with an invalid value (outside of the range of values) and this value is used as a threshold.

Altough the selection process could be done in the GPU (e.g. using a geometry shader to emit only valid texels), it is done as a preprocess on the CPU. After an initial rendering into the texture atlas, the values are read back to the the CPU and a display list is created, holding only the vertices for the valid texels.

The display list is then rendered using point rendering, transforming the world-space position from the texture atlas into the coordinate system of the voxel grid, according to the voxelization camera. The depth of the point is then used in combination with a bitmask to determine the position of the bit that represents the voxel in the voxel grid, and finally setting the correct bit on the voxel grid. This is possible by relying on a unidimensional texture previously created on the CPU that maps a depth value to a bitmask representing a full voxel at that certain depth interval.

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.

The atlas resolution should be chosen carefully. Using a low resolution for the texture atlas can create holes in the voxel grid. However, if the resolution is too high, the same voxel will be filled repeatedly, hurting the performance of the algorithm since the performance is directly related to the number of vertices generated and rendered using the display list.

3.2.3 *Data Structure/Mip-Mapping*

This approach relies on a binary voxelization, stored in a 2D texture. Each texel represents a stack of voxels along the negative z-axis of the voxelization camera, since each bit encodes the presence of geometry at a certain depth along the voxelization direction.

This 2D texture is used to create a mip-map hierarchy by joining the texels along the x and y axis. The depth resolution along the z axis is kept at each mip-map level in order to allow the rendering algorithm to decide more precisely if the traversal of this hierarchical structure can be stopped earlier. Each of the mip-map levels are generated manually and stored in the different mip-map levels of a 2D texture by joining four adjacent texels of the previous mip-map level (Figure 28).

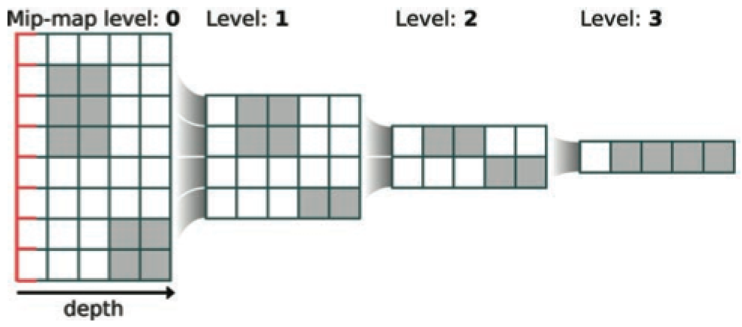

Figure 28: Mip-mapping. *Source:* Thiedemann et al. (2012)

3.2.4 *Rendering*

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.

Since the binary voxelization is 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 (Figure 29).

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.

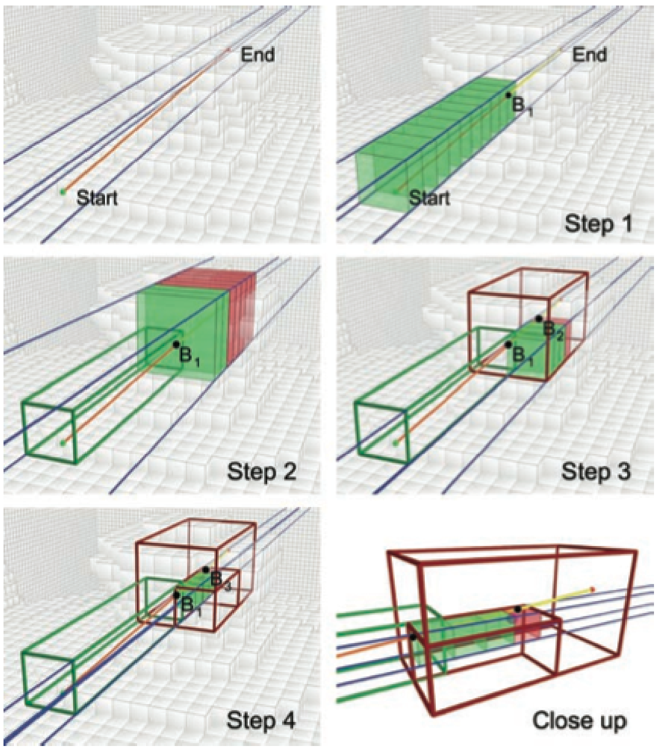

Figure 29: Hierarchy traversal. Blue lines: bounding box of the voxels in the actual texel. Green and red lines: bitmask of the active texel (empty - green and non empty - red). The green and red cuboids: history of the traversal for the texel (no hit - green and possible hit - red). *Source:* Thiedemann et al. (2012)

A bitmask is stored at each texel, representing a stack of voxels along the direction of depth 3.2.1. 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 30). 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 (defined by the user).

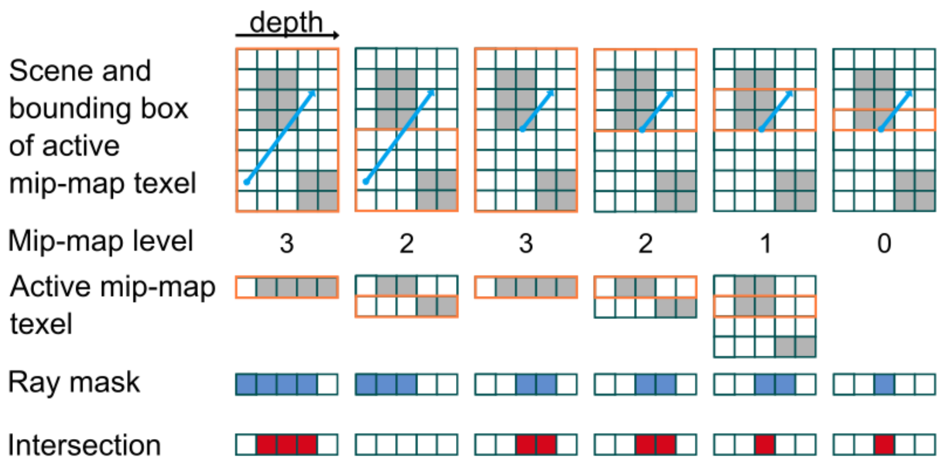

Figure 30: Hierarchy traversal in 2 dimensions. The blue arrow represents the current extent of the ray and in orange the bounding box of the current mip map levels is displayed. *Source:* Thiedemann et al. (2011)

The next step of the algorithm is to compute near-field illumination. For this purpose a reflective shadow map is generated (Dachsbacher and Stamminger, 2005) that contains direct light, position and normal for each pixel visible from the light position. Different techniques are employed for different types of lights: shadow maps are used for spotlights and cube maps for point lights.

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 31). **N** rays are cast using a cosine-weighted distribution, starting from the receiver position **x** with a maximum distance **r**.

Intersection tests are performed for each ray to determine the first intersection point. If a 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 hitpoint 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. This leads to $\epsilon = max(v, \frac{s}{\cos\alpha} \cdot \frac{z}{z_{near}})$.

Since the sample directions are generated using a cosine distribution function 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.

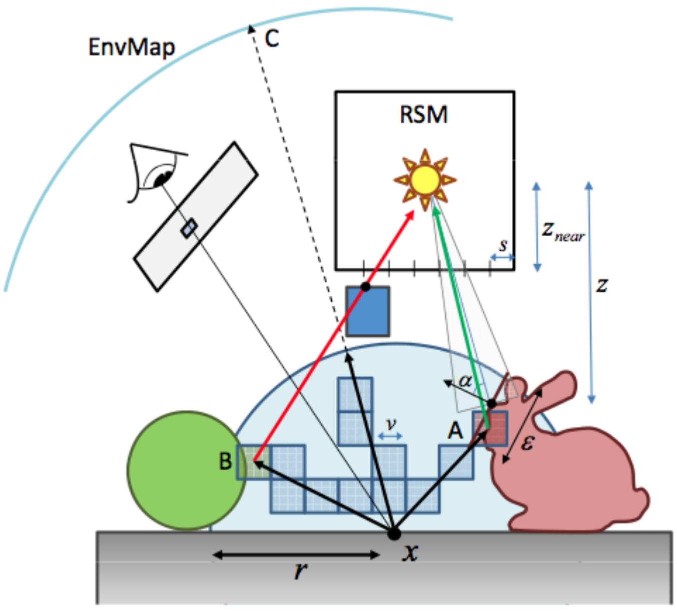

Figure 31: Near-field Indirect Illumination. *Source:* Thiedemann et al. (2011)

Indirect light has to be computed on a lower resolution in order to be feasible in real-time. Standard techniques like interleaved sampling and a geometry-aware blur filter are employed to be able to compute indirect light on a subset of all pixels.

In contrast to other image-based approaches, this method does not depend on the camera position, thus detecting senders and blockers that are invisible to the camera correctly.

However, due to the voxel discretization and the image-space blur, it is not possible to compute glossy materials properly.

It is possible to modify the rendering algorithm in order to extend its capabilities to better approximate global illumination, including compute glossy reflections. It is possible to create a path tracer based on the voxelized scene representation and to evaluate the visibility of virtual point lights ((Keller, 1997)) using the presented intersection test. Altough this technique presents a better approximation to global illumination, it does not compute in real-time, and is thus out of this context.

3.3 RASTERIZED VOXEL-BASED DYNAMIC GLOBAL ILLUMINATION

This method uses recently introduced hardware features to compute an approximation to global illumination in real-time (Doghramachi, 2013).

First, a voxel grid representation for the scene is created using the hardware rasterizer. The voxelization algorithm is similar to the one previously explained in Section 3.1.1.

The scene is rendered and written into a 3D texture buffer (voxel grid) using atomic functions, creating a 3D grid representation of the scene. This grid contains the diffuse albedo and normal information of the geometry on the scene and is recreated each frame, thus it is fully dynamic and does not rely on precalculations.

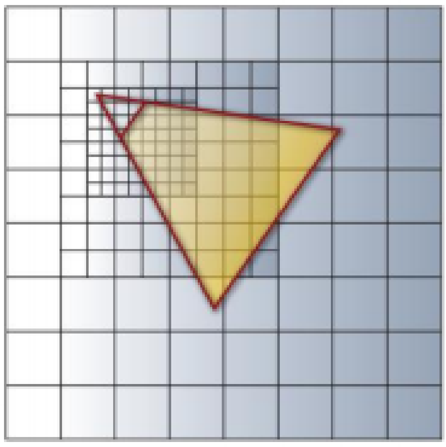
Figure 32: Nested Voxel Grids

The voxel grid 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 will have the same number of cells, but the size of the cells is increased (Figure 32). This allows to increase the detail of the 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.

After the grid has been created, the voxels are illuminated by each light source. The direct illumination is then 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.

In order to compute the indirect illumination, the generated virtual point lights will be propagated within the grid. This technique does not require the creation of a reflective shadow map nor the injection of virtual point lights into a grid afterwards, as opposed to the light propagation volume technique (Kaplanyan and Dachsbacher, 2010). The proposed technique can be subdivided into five distinct steps (Figure 33).

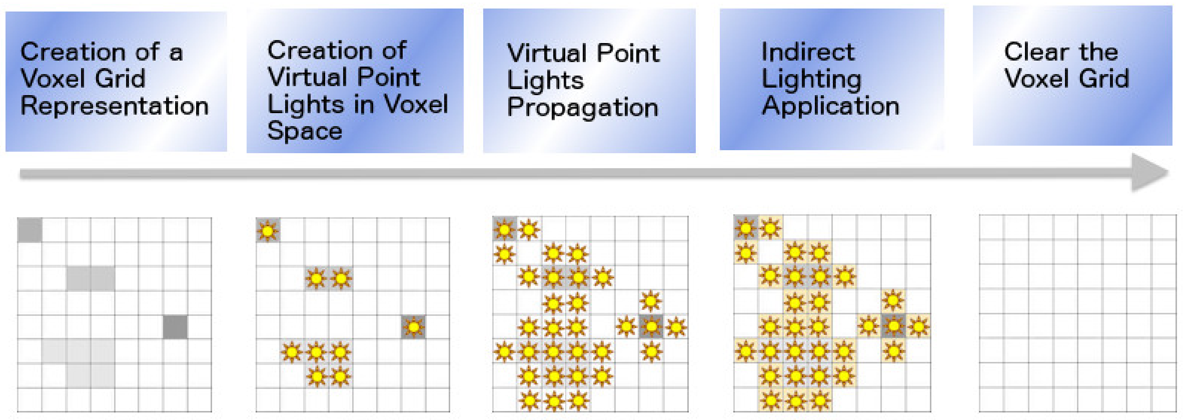
Creation of a Voxel Grid Representation
Creation of Virtual Point Lights in Voxel Space
Virtual Point Lights Propagation
Indirect Lighting Application
Clear the Voxel Grid

Figure 33: Pipeline of the algorithm

3.3.1 *Creation of the Voxel Grid Representation*

The voxel grid moves synchronously with the viewer camera and is snapped permanently to the grid cell boundaries to avoid flickering due to its discrete representation of the scene (Figure 34). To correctly map the scene to the voxel grid, an orthographic projection is used and thus, three view-matrices are used for the three different directions of projection (x,y,z). A set of properties for the cubic voxel grid also need to be defined: its extent, position and view-projection matrices.

The geometry inside the grid boundaries is rendered with disabled color writing and without depth testing in order to generate a fragment for every voxel containing scene geometry.

The view-matrix is chosen according to the major axis of the normal, in order to maximize the number of fragments generated for the primitive. The triangle is expanded using conservative rasterization in order to guarantee that every part of the triangle touching a voxel will generate a fragment. The resulting fragments are written into a 3D read-write structured buffer in the fragment shader with the help of atomic operations.

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, the color contrast value is calculated and used to write the fragments into the grid, thus giving preference to high contrast colors (high difference in their color channels).

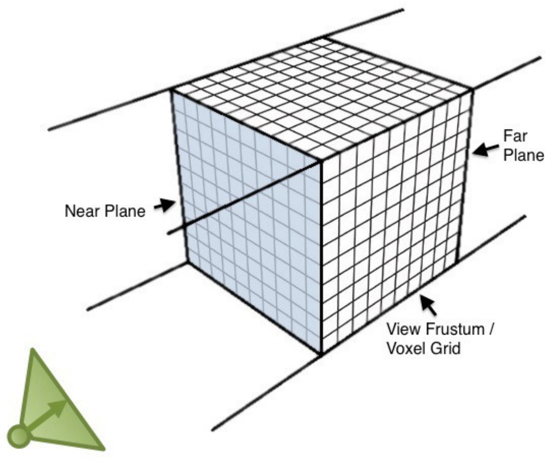

Figure 34: Orthographic Projection with a Voxel Grid in the View Frustum

The closest face of a tetrahedron to which the current normal is closest is also determined in order to account that normals can be opposite in the same voxel. This allows to write the normal into the normal mask channel corresponding 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 will not have any negative impact on the result (Doghramachi, 2013).

3.3.2 *Creation of Virtual Point Lights in Voxel Space*

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 are illuminated according to the type of the light source (Figure 35).

The voxels are converted into a second-order spherical harmonic representation of virtual point lights, combining the results of all light sources by using additive hardware blending. The second-order spherical harmonics coefficients for the three color channels are then written into three 2D texture arrays, one for each spherical harmonics channel.

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).

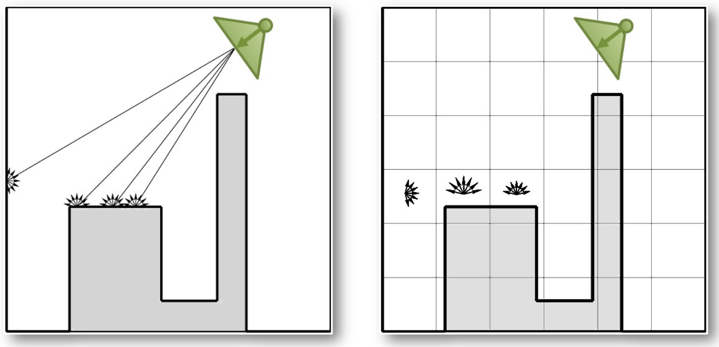

Figure 35: Lit surfaces are treated as secondary light sources and clustered into a voxel grid

3.3.3 *Virtual Point Lights Propagation*

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

Each virtual point light cell propagates its light to its surrounding six neighbor cells. During the propagation, the previously created voxel grid (3.3.1) is used to compute the occlusion of the light transport to the neighbor cells in order to avoid light leaking. 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 36). In the first iteration no occlusion is used in order to let the light distribute initially.

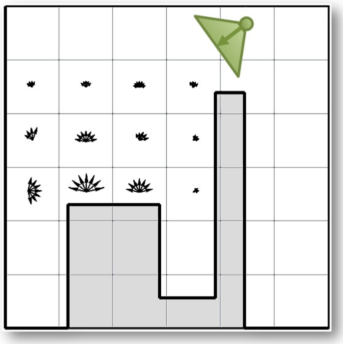
Figure 36: Virtual Point Light are propagated in the Voxel Grid

3.3.4 *Indirect Lighting Application*

The previously propagated virtual point lights are then applied to the scene to simulate indirect illumination. In order to do this, a depth buffer and a normal buffer are needed. The depth buffer contains information that allows to reconstruct the world-space position of the visible pixels and the normal buffer contains the perturbed normal information of the pixels.

A full-screen quad is rendered and the world-space position and the normal of each pixel is reconstructed. With the world-space position, the previously generated grid is sampled using linear hardware filtering and the third dimension is manually filtered to achieve smooth results. Lighting is then applied to the pixels using the sampled spherical harmonics coefficients and the surface normal. This method allows the computation of diffuse indirect illumination. However, a coarse approximation of the specular lighting is possible by extracting a dominant light source from spherical harmonics coefficents (pike Sloan, 2008).

The final step is to clear the buffer used for the voxel grid.

4 IMPLEMENTATION

4.1 TECHNOLOGICAL CHOICES

For the implementation of the chosen algorithms, several technological choices had to be made.

The most important choices are between the programming language and the graphics programming interface to use.

For real-time computer graphics, there are mainly two APIs that we can choose from: OpenGL and DirectX.

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 trough the internet. Altough it is multi-platform, 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.

DirectX is a collection of APIs that can handle a large amount of functions related not just to graphics, but multimedia in general. It provides libraries that can handle for example 3D graphics, sound and input. However, it is closely bound to the Microsoft Windows platform.

Both are very capable and very well maintained APIs, but since OpenGL presents the advantage of being cross-platform and its lack of some functionalities can be easily surpassed by using some extra libraries, this has been the technology chosen to deal with the 3D graphics of the applications.

The programming language chosen was C++. Since it will deal with the core of our applications, it has to be closely related to the other libraries used. Most of the libraries related to computer graphics are written in C++, so it is an obvious choice since it allows to use the libraries without having to rely on additional wrappers. It also offers great performance, which is essential for this kind of applications.

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 very 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). It still depends on other libraries, but provides a wrapper to perform all the operations in a very simple manner.

4.2 INTERACTIVE INDIRECT ILLUMINATION USING VOXEL CONE TRACING

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 descended until the level desired.

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 the voxel data is stored.

4.2.1 *Voxel Cone Tracing with a Full Voxel Grid*

In order to compute an approximation to global illumination using Voxel Cone Tracing, a hierarchic voxel representation of the scene has to be created. The algorithm is divided in several passes:

1. Voxelization
2. Light Injection
3. Mipmapping

4. Voxel Cone Tracing

Data Structures

In order to compute an approximation to global illumination using cone tracing, a voxel based representation of the scene must be created. The first step is to voxelize the scene in order to determine the necessary information to fill the voxel grid. After voxelization, voxel fragments are outputted to a buffer called a voxel fragment list (Listing 4.1).

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

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

Listing 4.1: Voxel Fragment List

The voxel fragments will be used later in the light injection pass to fill a 3D texture with the lighting information in the scene. This 3D texture is created with a size matching the voxelization resolution and with a RGBA8 texture format.

Voxelization

Each time the scene is voxelized, the voxel grid must first be cleared in order to avoid inconsistencies. This is done using an empty framebuffer object.

During initialization, a framebuffer object is created having no texture attachments bound to it. Then, when the grid has to be cleared, the framebuffer is bound, the 3D texture storing the voxel grid is attached to one of the color attachments of the framebuffer object and the texture is cleared using the *glClear* command.

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 define the volume that has to be voxelized.

To do that, an orthographic projection is defined, in such a way that its frustum covers the area 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 that will be stored in a voxel fragment list (Listing 4.1).

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

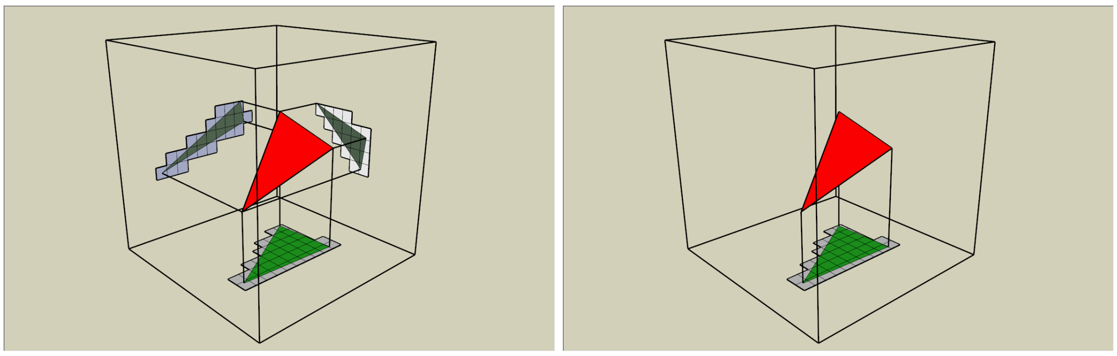

Figure 37: Projection of a triangle trough the three main axis of the scene. The Y axis is chosen for the voxelization since it is the one that will generate maximum number of fragments during rasterization.
Source: https://developer.nvidia.com/content/basics-gpu-voxelization

In order to produce the maximum number of fragments per triangle, each triangle must be projected along its dominant axis (Figure 37). The first step is to determine the normal of the triangle and finding which of the x,y,z components has the greater value. 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 4.2).

```
vec4 screenPos[3];
if (dominantAxis == eyeSpaceNormal.z)
  {
    screenPos[0] = projection * vec4(vPosition[0].xyz, 1.0);
    screenPos[1] = projection * vec4(vPosition[1].xyz, 1.0);
    screenPos[2] = projection * vec4(vPosition[2].xyz, 1.0);
  }
  else if (dominantAxis == eyeSpaceNormal.y)
  {
    screenPos[0] = projection * vec4(vPosition[0].xzy, 1.0);
    screenPos[1] = projection * vec4(vPosition[1].xzy, 1.0);
    screenPos[2] = projection * vec4(vPosition[2].xzy, 1.0);
  }
  else if (dominantAxis == eyeSpaceNormal.x)
  {
    screenPos[0] = projection * vec4(vPosition[0].zyx, 1.0);
    screenPos[1] = projection * vec4(vPosition[1].zyx, 1.0);
    screenPos[2] = projection * vec4(vPosition[2].zyx, 1.0);
  }
```

Listing 4.2: Computation of screen coordinates with vertex swizzling

Since the 2D fragments generated after rasterization only take into account pixels that intersect triangles trough 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.

This process, known as conservative rasterization (Figure 10) implies the computation of a screen space bounding box of the triangle before shifting its vertices outwards. This bounding box will serve to discard extra fragments generated during rasterization in the fragment shader.

Finally, the fragment shader is in charge of storing the voxel fragments into the voxel fragment list. First, 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.

Light Injection

In order to fill the voxel grid with the voxel fragments, the number of fragments written in the previous pass need to be determined. This information is 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, the attribute-less capability of the core profile in OpenGL is used, allowing to issue a draw call with no vertex buffer attached in order to launch a certain number of threads. A draw call with rasterization disabled and only one vertex is issued in order to launch a single thread that is in charge of altering the values of an indirect draw call structure (Listing 4.3).

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

Listing 4.3: Indirect Draw Structure

Now that the buffer containing the draw call parameters has the correct values, an indirect draw call is issued once again with attribute-less rendering, where the number of vertices are read from the indirect draw buffer, launching in this way one thread per entry on the voxel fragment list.

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 section 2.1 and the light parameters to determine the shading of each fragment, according to the Phong reflection model.

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.

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 4.4).

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. This creates a problem: 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 null, modify the alpha value so that the voxel is considered fully opaque (alpha = 1).

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.

```
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 4.4: RGBA8 Image Atomic Average Function

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 hierarchic 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. Since the mipmapping is performed level by level, no atomic operations are needed in order to ensure that the results stay coherent.

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 hierarchic 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 is performed using a deferred rendering approach (section 2.2). A full-screen quad is rendered and the fragment shader simply samples the geometry buffer in order to retrieve the positions, normals, colors and direct illumination.

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. 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.

Tracing cones trough the voxel grid is very similar to using volume ray casting (section 2.4). 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.

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.

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 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 then 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 increases between each sample. The proper mipmap level is chosen according to the size of that volume, since each voxel corresponds to a different volume in space in different mipmap levels. Since the full grid is stored in a 3D texture, it is possible to sample the voxels using hardware quadrilinear filtering.

This algorithm already allows to compute a very good approximation to indirect illumination. However, some artifacts are noticeable. 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 , and common ways to solve the problem are to reduce the stepping size, increasing in this way the number of samples taken for each cone, 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. This approach had a small impact on performance, and it was sufficient to remove visible banding on the scene.

Another problem is that since voxels define a volume in space, 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, five more voxel grids are needed, one for each direction, in order to be able to filter the irradiance anisotropically during the mipmap pass. 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. Then in the cone tracing pass, instead of sampling only one grid, three samples are taken from the three directional 3D textures weighted by the cone direction. Although the visual quality of the result is increased, this approach has some drawbacks: the memory consumption is greatly increased and the cone tracing pass has to do more texture calls, 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. However, multiple reflections are still ignored.

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 before light injection. 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 (subsection 4.2.1). However, some of these passes are different, due to the change of the data structure storing the voxel grid:

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

Voxelization

The voxelization pass is exactly the same used for the full voxel grid approach (subsection 4.2.1). At the end of this pass, a fragment list containing voxel fragments is available (Listing 4.1), with the information needed for the light injection pass and for the creation of the sparse voxel octree.

Sparse Voxel Octree Creation

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

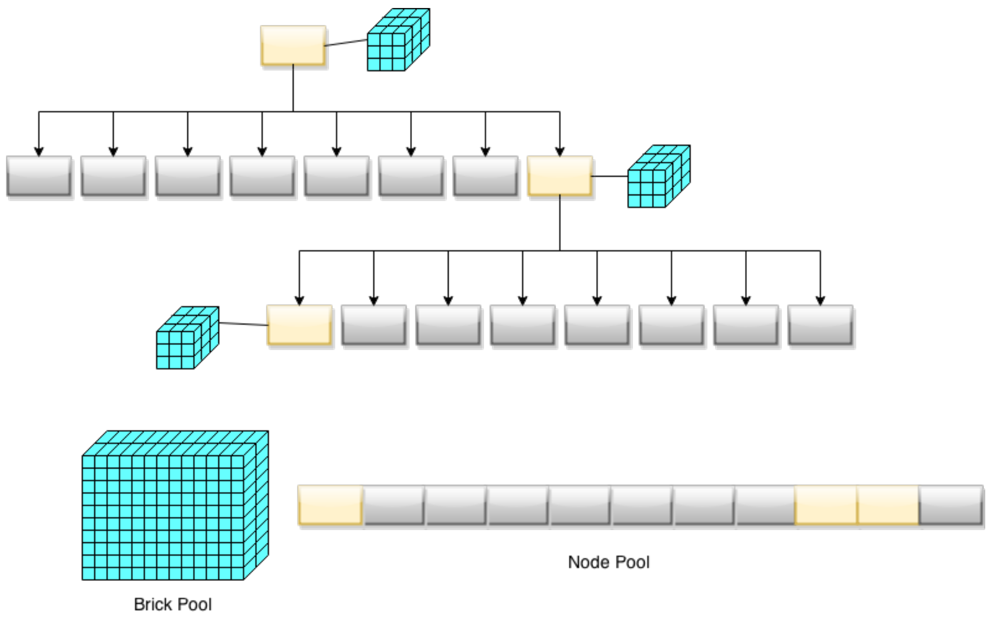

Figure 38: Octree and Octree Pools.

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 4.5). 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), 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 (Figure 3.1.4).

The brick pool is a 3D texture 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 13). Each brick stores voxel data corresponding to each node tile of the sparse voxel octree.

```
struct OctNode
{
  uint nodePtr;
  uint brickPtr;

  uint neighborX;
  uint neighborY;
  uint neighborZ;
};

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

Listing 4.5: Sparse Voxel Octree Structure

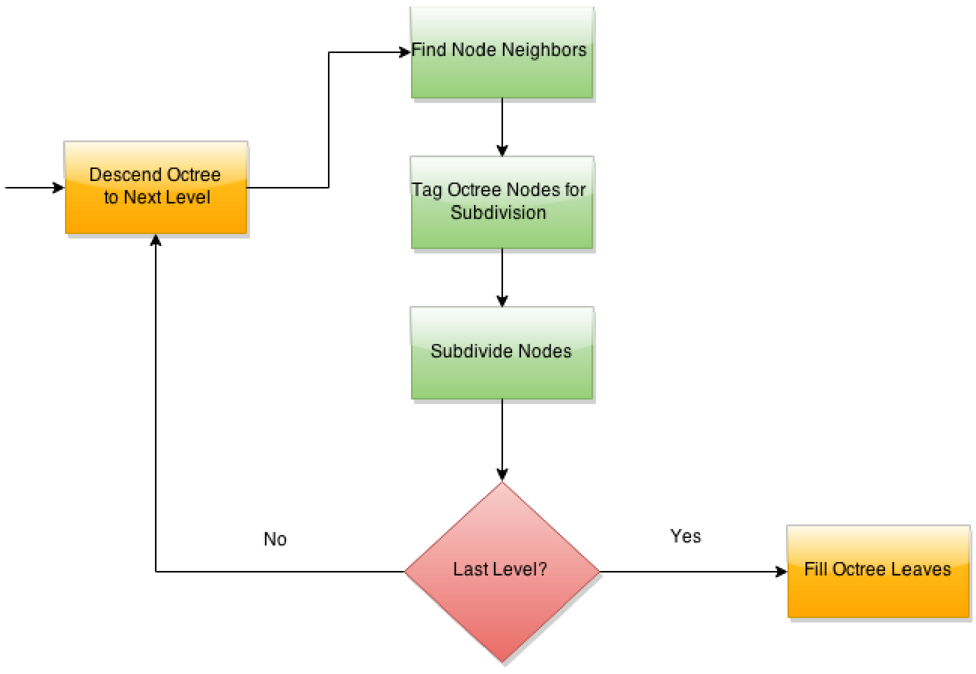

Figure 39: Octree Subdivision.

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 39. The subdivision of the octree is performed in three passes:

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

Before starting the octree subdivision, the number of threads for the indirect draw call must be set. Similarly to 4.2.1, a draw call is issued in order to read the atomic fragment counter value and modify the indirect draw structure parameters in order to be able to launch 1 thread for each entry in the voxel fragment list.

The first pass 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, each thread reads the world position from the voxel fragment list and traverses the octree using the kd-restart algorithm. The traversal starts from the root node, computing the volume dimensions for the 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 and the neighbor node address is written into the corresponding field in the octree node.

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. 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. The 30$^{\text{th}}$ bit of the node pointer of the octree node is then set in order to mark the node for subdivision.

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 4.6).

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

Listing 4.6: Indirect draw structure storing 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.

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.

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 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. 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 4.4). 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. 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 then spread trough the whole voxel brick.

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. Using attribute-less rendering, one thread per leaf node is issued. Each of the threads 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.

Mipmapping

Since bricks store information about neighboring voxels 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.

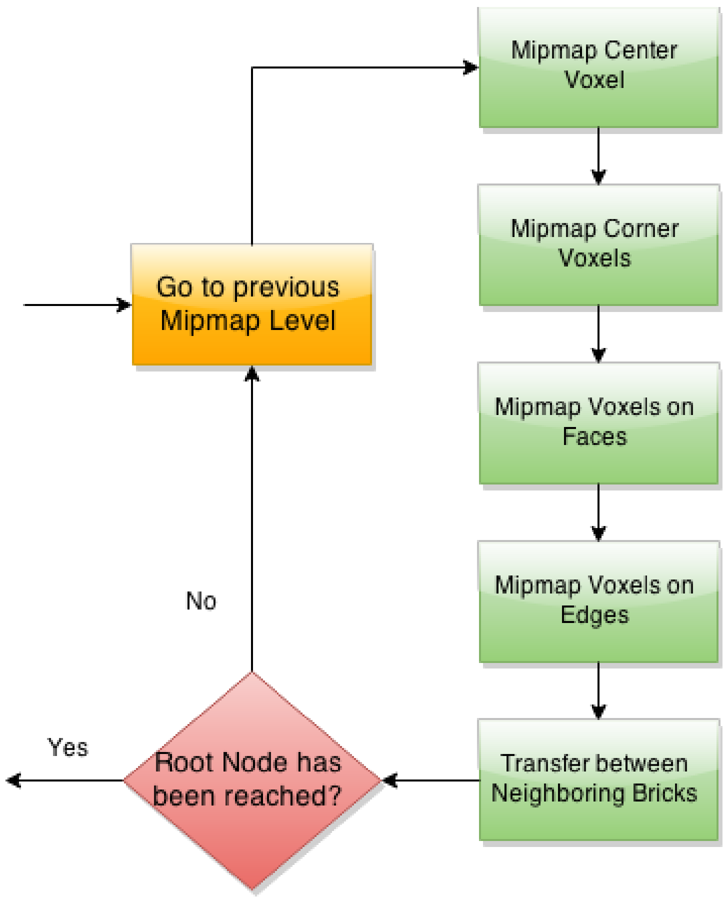
Mipmap Center Voxel
Mipmap Corner Voxels
Go to previous Mipmap Level
Mipmap Voxels on Faces
No
Mipmap Voxels on Edges
Yes
Root Node has been reached?
Transfer between Neighboring Bricks

Figure 40: Octree Mipmapping.

The indirect draw call parameters are altered using the number of nodes stored previously, launching indirect draw calls for the nodes on the current level. 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 voxel. For the rest of the voxels, some information has to come from neighboring bricks in order to complete the result. So, the approach taken is to compute only a partial averaged value using the children bricks, and then complete the result using the same neighbor transfer scheme used to complete the result on the leaves of the sparse voxel octree. 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 and 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 (subsection 3.1.3).

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 2.3 containing information about the world position from the fragments seen from the light's perspective, and a full-screen quad with a viewport corresponding to the reflective shadow map is rendered. The world position is sampled in the fragment shader and converted to voxel grid coordinates ([0, voxel grid resolution]) and the sparse voxel octree is traversed until the leaf node is encountered, retrieving the address corresponding to one of the four corners of the corresponding brick in the brick pool. 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 Phong reflection model , storing the result in the irradiance brick pool. 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 subsection 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. Now, the sparse voxel octree has to be traversed up to the desired mipmap 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 bypass this limitation, hardware trilinear filtering is used and the filtering between mipmap levels is performed manually.

4.3 RASTERIZED VOXEL-BASED DYNAMIC GLOBAL ILLUMINATION

The algorithm has suffered some changes since the original work (Doghramachi, 2013) was published. It initially only allowed to compute an approximation of the diffuse indirect illumination. However, it has been later extended to be able to approximate glossy reflections. This comes to the cost of an extra higher resolution grid that will be sampled using Voxel Cone Tracing to approximate glossy 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.

Also, the buffers clearing pass has been moved in this implementation from the last to the initial step. This was simply a choice that does not really change the behavior of the algorithm, it just allows to avoid clearing the buffer after its creation during the initialization.

Thus, the algorithm can now be subdivided into 7 distinct steps:

1. Clear Buffers
2. Voxelization
3. Direct Light Injection
4. Direct Light Propagation
5. Reflection Grid Creation
6. Reflection Grid Mipmapping
7. Global Illumination Rendering

4.3.1 *Data Structures*

For this implementation, two grids are used for diffuse indirect illumination. They both have the same size (32^3), but cover a different frustum. The grids can be static and thus cover always the same volume on the scene, or they can 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 slightly 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. The frustum is defined as an orthographic projection of the scene.

The grids are created as a Shader Storage Buffer, which is 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.

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

Listing 4.7: struct definition of a voxel in the voxel grid

The reflection grid is a higher resolution grid (256^3) that covers the same frustum as the smallest grid of the indirect diffuse illumination. It is defined as two mipmapped 3D texture that only store a single unsigned integer encoding color and occlusion. One for the voxelization pass, and one to store illumination after the light propagation of the direct illumination into the lower resolution grids is performed.

Since lighting will be encoded as virtual point lights using a spherical harmonics representation, three buffers (one for each color channel) are needed. Instead of 2D texture arrays as in the original implementation, 3D textures are used to encode the virtual point lights. This allows to use hardware quadrilinear filtering instead of sampling trilinearly each slice of the array and performing linear interpolation manually between the different slices.

4.3.2 *Buffer Clearing*

The algorithm starts by clearing the buffers storing the grids. For this, a compute shader with $4 \times 4 \times 4$ work groups and with a local size of $8 \times 8 \times 8$ is created. This allows us to launch actually $32 \times 32 \times 32$ threads and query the 3D voxel each thread needs to clear with gl_GlobalInvocationID.

Each thread then clears its corresponding voxel with a null value using image store operations.

4.3.3 *Voxelization*

The voxelization pass is actually very similar to the one previously explained in section 4.2. It is performed once for each grid (fine, coarse and reflection grids). A viewport size of 64×64 is used for the low resolution grids and a viewport size of 512×512 is used for the higher resolution reflection grid.

As before, depth testing and the color mask/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/reflection grids and the coarse grid. These parameters are set by the user and will affect the voxelized area of the scene, and thus the area that will have approximate indirect illumination.

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

The geometry shader finds the dominant axis for the normal and computes its projection along that axis. Instead of using three different matrices for the projection of the triangle along the dominant axis, the same method used before is employed and a swizzle matrix is computed and used to swizzle the vertices of the triangle towards the dominant axis of its normal. The edges of the triangle are then expanded outwards in order for the rasterizer to generate fragments for every pixel touched by a triangle. This time however, no bounding box is computed to discard the excess fragments later in the fragment Shader.

The fragment shader retrieves the RGB material color and encodes it in linear space into the last 24 bits of an unsigned integer. The higher 8 bits are then used to encode the contrast value (difference between the color channels) computed from the retrieved color in linear space, as well as 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.
For the fine and coarse grids it is still needed to compute and encode normal information for writing in the buffers. Since fragments in the same voxel can have opposite normals, special care has to be given when writing normal values. The dot product between the normal and the face of a tetrahedron is determined and written into the highest 5 bits of the normal mask. Each channel of the normal is then encoded in 9 bits (1 for the sign and 8 for the value) and written in the remaining bits of the normal mask. Finally, according to the dot product, the normal mask is written into the corresponding channel of the vector in the buffer using an atomicMax operation. This way, the closest normal to the tetrahedron will automatically dominate.

4.3.4 *Direct Light Injection*

For the light injection, a 32 $\times$ 32 quad is rendered using instanced rendering. 32 instances are rendered in each pass, and this is performed for the fine and coarse grid only.

The vertex shader simply outputs the vertex position and the gl_InstanceID to the geometry shader. The geometry shader then emits the vertices and the corresponding instance ID of the triangle to the fragment shader. Finally, the fragment shader uses the input variable gl_FragCoord and the instance ID passed by the geometry shader to retrieve the corresponding voxel from the attached buffer. The color and normal are then decoded and the world-space position is computed using the voxel grid coordinate. With this information, a shadow map is used to compute diffuse direct illumination depending on the type of the light.

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.

4.3.5 *Direct Light Propagation*

Now that the spherical harmonics grids (which represent our virtual point lights) are lit with the direct illumination, they have to be propagated trough the grid in order to add their contribution to neighbor voxels. To do this, a compute shader with 4 $\times$ 4 $\times$ 4 work groups and with a local size of 8 $\times$ 8 $\times$ 8 is created. Similarly to the buffer clearing pass, this allows to obtain the index into the voxel grid of each thread by querying the gl_GlobalInvocationID input variable. We initialize the voxel values with the spherical harmonics coefficients of the current grid position and then the contributions from the six neighboring voxels is computed. We start by determining the direction from the neighbor voxel to the current cell and its corresponding solid angle. From here, it is possible to obtain the spherical harmonics coefficients for that direction and compute the flux from the neighbor cell to the face of the current cell by weighting them by the solid angle. However, we also need to account for the occlusion in order to perform the light propagation accurately. The grid buffer that contains the color/occlusion mask and the normal masks is accessed to obtain the information for the neighboring voxel. The normals are decoded and the closest to the specified direction is calculated. By using the same zonal harmonics function utilized in the previous pass, the occlusion coefficients are computed. Finally, summing all the contributions from the neighboring voxels correcly 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. This process is repeated multiple times to allow the lighting to propagate trough the grid. In the first pass, no occlusion weight is used to allow an initial propagation of the virtual point lights.

4.3.6 *Reflection Grid Creation*

The creation of the reflection grid is done using a compute shader with $32 \times 32 \times 32$ work groups and with a local size of $8 \times 8 \times 8$, since the 3D texture for this reflection grid has a 256^3 resolution. First, the color/occlusion is retrieved from the grid previously generated during the voxelization pass. Then, the spherical harmonics coefficients of the finest voxel grid are sampled (since the reflection grid has the same frustum). In this way, an ambient term is extracted from the corresponding voxel. If the grid center is snapped to the camera position, glossy reflections only cover a small area around the camera. To ensure that no popping artifacts are introduced when the camera is moving, the ambient term is faded out with the distance to the grid center. This ambient term is then stored in the lit reflections 3D texture and the previously generated reflection grid is cleared. In this way, no extra pass is needed to clear the grid.

4.3.7 *Reflection Grid Mipmapping*

In the original algorithm, the author used the DirectX API to automatically generate 4 mipmap levels for the 3D texture. In OpenGL however, altough the specification states that the mipmap generation function (glGenerateMipmap) accepts 3D textures as the target to generate the mipmaps, in practice the function call would not operate properly and no mipmaps were created in the hardware used for testing. To bypass this issue, a manual mipmapping was implemented using the compute shader capabilities. First of all, the memory for the mipmaps has to be allocated during the initialization. Since only 4 mipmaps are needed, only 4 levels need to be allocated. However, OpenGL once again would not behave properly if only 4 texture levels were allocated/defined. For the texture object to function properly, all mipmap levels needed to be allocated. To perform the mipmapping, the compute shader is invoked 3 times, one for each lower mipmap level, and with a decreasing number of thread groups for each level (16, 8 and 4) in order to launch the correct number of threads for each level. The local size continues with the same size as with the other compute shader invocations in previous passes. Each thread will then sample the eight voxels on the upper level corresponding to the voxel grid coordinate (once again determined with gl_GlobalInvocationID), average them and store the result using an image store operation.

4.3.8 *Global Illumination Rendering*

To compute global illumination, a full-screen quad is rendered using deferred rendering, and world-space positions, normals and material are retrieved.

Global illumination can be subdivided in two components: direct and indirect illumination. Direct illumination has already been computed in a previous pass using deferred shading. Indirect lighting can in its turn be subdivided in two other components: diffuse and specular (in this case, glossy) indirect lighting.

To compute diffuse indirect illumination, we start by computing spherical harmonics coefficients for the normal in the current world-space position. Once again, zonal harmonics are used to generate these coefficients. 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. Since two grids are being used, both have to be sampled and the distance from the center of the grid is used to interpolate between their results to obtain a smooth transition between them.

For glossy reflections, a very different process is used. Similarly to section 4.2, voxel cone tracing is used to accumulate lighting contributions from the voxels along the cone axis. We start by computing the reflected direction from the eye to the world-position of the fragment and launch a ray in that direction, accumulating color and occlusion from the lit reflection grid until total occlusion is reached. Since different mipmap levels are sampled depending on the traced distance from the starting point and the cone aperture, the ray actually resembles a cone. The difference now 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 trough the scene.

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

Since the center of the voxel grids can be kept synchronous with the viewer camera and keeps the global illumination to a limited frustum around it, this algorithm does not depend on the size of the scene, being perfectly capable of handling large scenes without losing its interactivity. Another great advantage is that the reflection grid is created in separate passes, 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

Differently from the other algorithms implemented, this one also performs some computation on the CPU. First of all, it computes some bitmasks and stores them into 1D and 3D textures in order to send them to the shaders when needed. Also, after the voxelization process, a display list is created in order to discard invalid texel values from the texture atlas. However, its greatest disadvantage is the need for the models surfaces used during the voxelization process to be mapped to a texture atlas. This is done simply by generating another definition of the object (for example using a Wavefront .obj format) in which the texture coordinates are used to map the vertices of the object to a position in the texture atlas. Then, when rendering the object during voxelization, the atlas texture coordinates are simply read from the texture coordinates passed to the shader.

The algorithm can be subdivided into several passes:

1. Voxelization
    a) Binary Atlas Creation
    b) Pixel Display List Creation
    c) Voxel Grid Creation
2. MIP-mapping
3. Indirect Lighting Computation

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 differently. Direct lighting will be sampled during the indirect lighting computation pass, using a Reflective Shadow Map generated previously.

4.4.1 *Data Structures*

Several bitmasks need to be created. These are simply 1D textures storing information about the bitmasks in an unsigned integer RGBA format of size 128. .

A 2D texture is used to encode the bitmasks for the rays launched during the indirect ilumination pass. This texture is created by attaching the texture to the framebuffer and drawing a full-screen quad with a viewport of size 128, using one of the bitmasks previously created. Then, in the fragment shader, the x and y coordinates of the fragment (retrieved using the gl_FragCoord variable) are used to fetch the bitmask for each coordinate and generate the final result using the bitwise exclusive or operator (XOR).

For storing the binary atlas, a 2D texture with half size floating point RGBA values is created. Its size is a user-defined value, since it greatly depends on the object, in order to avoid holes or overdraw during the voxelization and creation of the voxel grid (section 3.2).

The voxel grid will be used to store the binary representation in our scene after voxelization. It is a MIP-mapped 2D texture storing RGBA unsigned integers. Its size is defined in accordance with the voxelization resolution and each texel represents a stack of oxels along the voxelization depth.

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.

For a better sampling of the diffuse indirect illumination, an auxiliar texture is used to rotate the rays randomly. It is defined as a small 2D texture storing random data in RGB floating point format.

4.4.2 *Binary Atlas Creation*

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 texture atlas. The model loaded uses a previously generated description of the object that contains the mapping of the surfaces of the object to a texture atlas in the texture coordinates definition. The vertex shader simply 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. In order to be able to identify invalid texels in the next pass, the atlas texture is cleared with some value that will serve as a threshold to discard invalid texels.

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. However, a lot of texels contain invalid values that we wish to discard to reduce the amount of vertices issued in the draw call. For this, 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. 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. The point size can be increased in order to attempt to close holes if the resolution of the texture atlas is too small.

4.4.4 *Voxel Grid Creation*

The next step is to generate the voxel grid. For this purpose, the pixel display list is rendered with the voxel grid texture bound to the framebuffer. An orthographic projection is defined in order to be able to transform the world-space positions into voxel grid coordinates. This way, the extents of the orthographic projection control the region that is voxelized. Since we are using a binary voxelization, the channels of the voxel grid texture encode the voxels using a bitmask. To be able to do this, a logical OR operation has to be defined for the framebuffer (Eisemann and Décoret, 2006). The vertex shader fetches the world-space position from the atlas texture using the texture coordinate (passed as a vertex). The vertex coordinate is transformed (using the orthographic view and projection matrices) and the Z coordinate of the vertex is passed to the fragment shader, mapped to [0, 1] . Since each texel of the voxel grid actually represents a stack of voxels along a certain depth, this Z coordinate is actually the coordinate that needs to be retrieved from the bitmask texture and outputted by the fragment shader in order to set the correct bit in the voxel grid.

4.4.5 *Mipmapping*

The voxel grid needs to be mipmapped in order to generate a binary hierarchic 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.

4.4.6 *Indirect Lighting Computation*

To compute diffuse indirect lighting, several rays will be launched for each 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.

Let us start with the creation of the hit buffer. A full-screen quad is rendered and the world-position is fetched from the g-buffer. This world-space position is the starting point of the reflected ray. By using a cosine weighted distribution, a direction for the reflected ray is computed. Then, to avoid self-shadowing, the starting point of the ray is advanced by an offset at least the size of the voxel diagonal. The start and end point of the ray are then defined. 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 1 level since the intersection with the root node was already computed. 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's 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 unit-coordinates) into the hit buffer.

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. 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 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.

Finally, direct and indirect illumination are combined by rendering a full-screen quad and summing the contributions of each pixel. However, since indirect lighting is computed for a much lower resolution that the window size, the indirect lighting result is blurred using a geometry aware blur before adding its contribution to the global illumination result.

5 CONCLUSIONS

Several algorithms for computing an approximation to global illumination in real-time applications were presented in this thesis. It has been shown that these algorithms share a similar structure, requiring the scene to be pre-filtered using some kind of voxelization algorithm.

In section 4.2, the voxelization pass creates a fragment list storing information about the world positions, colors and normals of the scene by rasterizing the triangles of the scene in the direction of the dominant axis of their normal.

This fragment list is then used to create a pre-filtered hierarchic representation of the lighting in the scene in order to be able to launch cones to gather indirect illumination. One cone is launched in the direction of the reflection in order to gather the specular contribution to indirect illumination, while five cones are launched in the the hemisphere around the normal vector in order to gather the diffuse indirect illumination.

The cone tracing pass steps along the cone axis, sampling the data structure storing voxel data at different mipmap levels based on the cone diameter, and alpha-blending is used to accumulate the samples (thus treating voxels as a participating media) until opacity saturates.

The voxel cone tracing approach was tested against a full voxel grid, which allowed for an easier and faster access to the data structure, speeding most of the steps of the algorithm, at the cost of an higher memory usage. To reduce memory usage, a sparse voxel octree was created entirely on the GPU, allowing to reduce the memory footprint of the voxel grid, at the cost of having to traverse the octree each time a voxel has to be sampled or updated.

The algorithm presented in section 4.3 shares some similarities with the voxel cone tracing approaches. The main differences are in the way diffuse indirect illumination is computed. It uses two cascaded grids for the diffuse indirect illumination and one higher resolution grid to compute glossy reflections.

The first thing to do is to create a voxelized representation of the scene by rasterizing the triangles in the direction of the dominant axis of their normal. Then, the resulting fragments are used to encode color based on their contrast, and normals using the closest face to a tetrahedron, storing them in a buffer.

Next, the fragments in the buffer are used together with a shadow map to compute diffuse albedo, storing the result into the voxel grids using a second order spherical harmonics representation. This generates voxel grids containing virtual point lights that are then propagated along the three axes of the grid.

The grids storing the propagated virtual point lights are sampled in order to generate diffuse indirect illumination, while the higher resolution grid is mipmapped in order to use voxel cone tracing to sample glossy reflections.

The algorithm described in section 4.4 maps the vertices of the objects in the scene to a 2D texture atlas. This process, called binary voxelization, encodes the world positions of all the geometry in the scene into the texture atlas, generating a binary representation of the scene.

The texture atlas is read back to the CPU and all valid texels are used to generate a pixel display list. This pixel display list is the rendered in order to insert all the texels into a voxel grid, encoded as a 2D texture. Since a 2D texture is being used, each texel in the voxel grid represents a stack of voxels in the voxelization depth and a texture storing bitmaps (previously generated on the CPU) is used in order to encode the voxels correctly.

The voxel grid is then mipmapped by joining the texels in the x and y directions, keeping the depth with the same precision between mipmap levels since it is already limited by the choice to use each texel as a stack of voxels.

Diffuse indirect illumination is then computed by launching rays trough the scene, intersecting them with the voxel grid. When an intersection is found, a reflective shadow map is used to sample the direct lighting contribution of the sampling position.

All these approaches use a voxel grid representation of the scene in order to compute an approximation to global illumination. This voxel grid 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 in the other side are costly to generate and maintain updated, but reduce the memory footprint of the voxel grid by collapsing empty voxels. 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 the obtained result.

In order to counter these problems, new approaches need to arise in order to decrease the memory cost of the voxel grids, allowing at the same time to rapidly update and access them. One way to decrease the wasted memory is to use the empty space of the 3D texture storing the voxel grid to encode necessary data to compute other visual effects. For example, voxel grids have been used for atmospheric effects (Vos, 2014) and fluid simulation (Roble et al., 2005). More recently, Nvidia has presented a new technique in order to approximate global illumination based on voxel cone tracing, called VXGI (https://developer.nvidia.com/vxgi) that uses 3D clipmaps to encode voxel data.

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.

Annen, Thomas; Mertens, Tom; Seidel, Hans-Peter; Flerackers, Eddy, and Kautz, Jan. Exponential shadow maps. In *Proceedings of Graphics Interface 2008*, GI '08, pages 155–161, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society. ISBN 978-1-56881-423-0. URL http://dl.acm.org/citation.cfm?id=1375714.1375741.

Appel, Arthur. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM. doi: 10.1145/1468075.1468082. URL http://doi.acm.org/10.1145/1468075.1468082.

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.

Deering, Michael; Winner, Stephanie; Schediwy, Bic; Duffy, Chris, and Hunt, Neil. The triangle processor and normal vector shader: A vlsi system for high performance graphics. *SIGGRAPH Comput. Graph.*, 22(4):21–30, June 1988. ISSN 0097-8930. doi: 10.1145/378456.378468. URL http://doi.acm.org/10.1145/378456.378468.

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. Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78. ACM SIGGRAPH, 2006. URL http://maverick.inria.fr/Publications/2006/ED06.

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.

Engel, Woflgang F. page 197–206. Charles River Media, Boston, Massachusetts, 2006.

Foley, James D.; van Dam, Andries; Feiner, Steven K., and Hughes, John F. *Computer Graphics: Principles and Practice (2Nd Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. ISBN 0-201-12110-7.

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.

Hadwiger, Markus; Kniss, Joe M.; Rezk-salama, Christof; Weiskopf, Daniel, and Engel, Klaus. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. ISBN 1568812663.

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.

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.

Keller, Alexander. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. ISBN 0-89791-896-7. doi: 10.1145/258734.258769. URL http://dx.doi.org/10.1145/258734.258769.

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.

Max, Nelson. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995. ISSN 1077-2626. doi: 10.1109/2945.468400. URL http://dx.doi.org/10.1109/2945.468400.

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.

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.

pike Sloan, Peter. Stupid spherical harmonics (sh) tricks, 2008.

Reeves, William T.; Salesin, David H., and Cook, Robert L. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph.*, 21(4):283–291, August 1987. ISSN 0097-8930. doi: 10.1145/37402.37435. URL http://doi.acm.org/10.1145/37402.37435.

Roble, Doug; Zafar, Nafees bin, and Falt, Henrik. Cartesian grid fluid simulation with irregular boundary voxels. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. doi: 10.1145/1187112.1187279. URL http://doi.acm.org/10.1145/1187112.1187279.

Saito, Takafumi and Takahashi, Tokiichiro. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990. ISSN 0097-8930. doi: 10.1145/97880.97901. URL http://doi.acm.org/10.1145/97880.97901.

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.

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.

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.

Wimmer, Michael; Scherzer, Daniel, and Purgathofer, Werner. Light space perspective shadow maps. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR'04, pages 143–151, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN 3-905673-12-6. doi: 10.2312/EGWR/EGSR04/143-151. URL http://dx.doi.org/10.2312/EGWR/EGSR04/143-151.

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.