TEMPORAL VOXEL CONE TRACING WITH INTERLEAVED SAMPLE PATTERNS

BY

SangHyeok Hong

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
awarded by DigiPen Institute of Technology
Redmond, Washington
United States of America

March
2015

Thesis Advisor: Gary Herron

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS TITLED

Temporal Voxel Cone Tracing with Interleaved Sample Patterns

BY

SangHyeok Hong

HAS BEEN SUCCESSFULLY COMPLETED ON March 12th, 2015.

MAJOR FIELD OF STUDY: COMPUTER SCIENCE.

APPROVED:

Dmitri Volper date	Xin Li date
Graduate Program Director	Dean of Faculty
Dmitri Volper date	Claude Comair date
Department Chair, Computer Science	President

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

THESIS APPROVAL

DATE: March 12th, 2015

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

SangHyeok Hong

ENTITLED

Temporal Voxel Cone Tracing with Interleaved Sample Patterns

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

Gary Herron date Thesis Committee Chair	Xin Li date Thesis Committee Member
Pushpak Karnick date Thesis Committee Member	Matt Klassen date Thesis Committee Member

Contents

Table of Figures

Abstract

The aim of this thesis is to investigate and determine new effective methods for global illumination using voxel cone tracing. It should introduce the reader to the topic of global illumination, covering basic knowledge from scratch to understand the trend of real-time global illumination algorithms used in games. Existing algorithms that has been devised for global illumination such as Precomputed Radiance Transfer (PRT), Light Propagation Volume (PRT), and Voxel Cone Tracing (VCT) will be explored, identifying the positive and negative aspects of each algorithm. A new approach to the voxel cone tracing using interleaved sampling and voxel mips will be examined and compare to the original voxel cone tracing technique to highlight improvements that has been made.

1. Introduction

1.1 Motivation

The computer graphics in games have been developed over the generations of game consoles like the XBOX and PS series. The previous console generation, which included the PlayStation3 (PS3) and Xbox360, took place during the renaissance period for computer graphics in games. In particular, the hardware architecture of the PS3 utilized 'Cell Processors', which made it possible for rendering engineers to apply many visual techniques previously used in movie industries to games as real-time techniques. The new era of consoles — the PS4 and XBOX ONE — has begun. Some noticeable features of the new consoles include the adoption of an x86 processor and GPU separately in their hardware. This provides developers with a more convenient way to develop games, and it provides lots of memory and high speed utilizing DDR5 Graphics RAM. On the surface, it looks like a big improvement in both hardware and the environments for developing games, compared to previous generation. However, upon closer inspection, we can see that there are no big advances in processing power from the cell processor of the PS3 several years ago. At that time, the cell processor was one of biggest revolution in that period — the hardware itself was superior to desktop processors — whereas the processor in a modern console can't catch up with the combination of a CPU and GPU in desktop. As computer hardware advances faster than console hardware, it becomes increasingly difficult to apply cutting-edge rendering techniques on consoles, since they cannot satisfy growing performance demands. Since it is unlikely that consoles will be able to keep pace with PCs, we should begin to take a different approach to enhancing the visual effects in games. To keep improving graphical quality during the course of a console generation (since the hardware won't improve for many years), we need to look at other aspects of the new consoles' features. The most promising aspect is the memory space allocated in the graphics card; the space has remarkably increased. Taking aspects of memory and processor into account, volumetric rendering techniques are being actively researched. These volumetric techniques can achieve peak performance on the hardware architecture of new consoles.

In computer graphics in games, a high frame rate is very important for rendering in real-time. The real-time global illumination algorithm has taken a significant role in modern rendering engines. Originally, the algorithms used in the animation and movie industries are mostly based on ray-tracing algorithms, which shoot lots of rays to make a realistic scene. But these ray-tracing algorithms can’t be done in real-time, and are often called them “off-line” global illumination techniques. To approximate global illumination, the calculation compressing all data for the rendering equation spatially is necessary. The common techniques are Spherical Harmonics (SH) and Gaussian Lobes in lattices. All algorithms using those techniques are categorized as ‘Volumetric Rendering’.

Currently, the real-time global illumination algorithms widely adapted to rendering engines are Light Propagation Volume (LPV) and Voxel Cone Tracing (VCT). In the past, it has not been possible to calculate global illumination in real-time, so several off-line global illumination algorithms were used. Computing a light map by ray-tracing algorithms or utilizing Precomputed Radiance Transfer (PRT) are classic methods for Global Illumination (GI). As time went on, most developers realized that these methods are too inefficient due to the long rendering times. For example, artists can get in trouble for seeing whether their art works in global illumination; it takes too much time. For this reason, real-time global illumination algorithms have been gradually researched in a more productive way.

Generally, the cycle of the console hardware takes about 6 years. At this point, volumetric rendering will be a key topic for real-time global illumination techniques.

1.2 Overview of the work

To understand real-time global illumination, there is some background to cover first, including the Normal Distribution Function (NDF), shadow mapping, and the basis function for compressing directional data. In this thesis, we will cover those topics in detail. For the compression model for storing directional data, the Spherical Harmonics and Gaussian function will be presented in separate chapters due to their importance in understanding the current state of real-time global illumination − for instance, for their part in Light Propagation Volumes (LPVs) and the Voxel Cone Tracing (VCT).

This paper suggests a new algorithm as a contribution to the VCT, to adapt it to run in real-time. It is titled ‘Temporal Voxel Cone Tracing with Interleaved Sample Patterns’. To summarize, it reduces the number of the voxel cones tracing per pixel by one, and introduces interleaved sampling. Interleaved sampling causes noisy artifacts. To reduce these artifacts, the Reverse Reprojection and Temporal Refinement algorithms are used. The details of these methods will be covered in following chapters.

With this paper, readers can see the trend of global illumination in games and get the overall backgrounds necessary to more easily understand other computer graphics (CG) papers that cover GI.

2. Related Work

2.1 Introduction

This chapter will cover basic materials for understanding algorithms for real-time global illumination that are widely used in the game industries. First, we will begin by reviewing local illumination models − in particular, BRDF (Bidirectional Reflectance Distribution Function). Local illumination models currently play a key role in producing photo-realistic effects in real-time rendering. Shadow mapping is will be next topic covered. Outwardly, it doesn't seem to be related to GI, but it is necessary for generating Virtual Point Lights (VPLs) used in indirect illumination. We are going to frequently deal with the topics of Gaussian Lobes and Spherical Harmonics; these will be referred to as 'Directional Compression Data models'. Efficiently storing directional data into limited channels in a texture is an important portion of the process of GI. To do this, compression techniques are needed, using basis functions with inner products; we generally call this a projection. In this area, there are lots of algorithms that have been researched: Wavelet, Spherical Harmonics, Gaussian Lobes, and more. We will then focus on Gaussian Lobes and Spherical Harmonics, which are widely used in volumetric structures. Considering their importance, we will try to describe as much detail as we can, to prevent readers from having the same trouble as I experienced. The final section will cover major GI algorithms, computed both off-line and in real-time.

2.2 Local Illumination Models

As an introduction, we briefly described what the local illumination models are and how important they are in real-time rendering. The most widely adopted technique is Bidirectional Reflectance Distribution Functions (BRDFs). These functions describe how a given surface point responds to light, depending on the incoming light direction and outgoing view direction in (Hoffman, 2012). But it can be described as 'the approximated distribution equation for the interaction between lights and material models'. It is straight forward, and it conveys a lot of information in one sentence. The BRDF is the combination of a Normal Distribution Function (NDF) and a Shadowing-Masking Function, commonly referred as the Geometric Term. In this chapter, we are only going to cover BRDF in enough detail to provide the background for understanding

NDF. Therefore, readers are recommended to review the basic background for optical physics and mathematics, as well as the shadow masking function (Hoffman, 2012).

2.2.1 NDF (Normal Distribution Function)

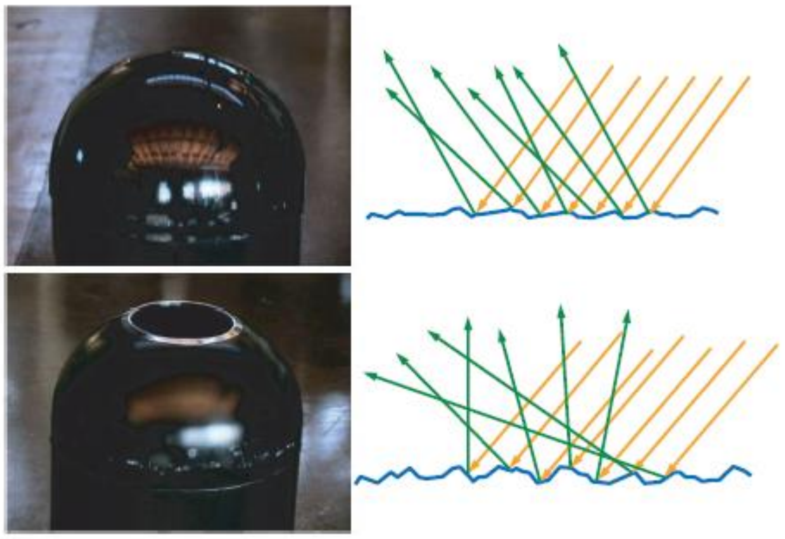


Figure 1. Visible results depending on microfacet NDF (Hoffman, 2012)

The NDF describes a distribution for certain data sets. In BRDF, the NDF is a microfacet NDF, describing the distribution of normals on a micro-surface. The roughness on a microscopic surface determines the material's reflective properties. In other words, the distribution of micro-geometry normals is the main contributor in material models (Figure 1).

To make use of the NDF, it is essential to understand that its graph contains many features describing the material. The ability to analyze a graph of the NDF allows you to define your own material models. We can model any given material with the normal distribution function by determining the mean and variance.

There are various local illumination models currently used in games. All of them are classified by the shape of the NDF graph. Among these are the Blinn Phong, Beckmann and Oyne Neyar models. For metal materials, we use the Phong or Beckmann distributions. The Onye-Neyar distribution is used to represent non-metalic material, but the main purpose for the BRDFs is to model metallic materials, accounting for specular features. As we said earlier, it is necessary to be able to analyze various NDF graphs. Each different local illumination model is defined by the use of a unique NDF. To give you more insight into NDFs, we will look at some examples of the NDF graphs mentioned above.

2.2.2 The Phong distribution

Before looking into the graph, let's first define the Phong NDF:

$$D_p(m) = \frac{\alpha_p + 2}{2\pi}(n \cdot m)^{\alpha_p}$$

The above equation is in normalized form. For details of the equation, refer to the paper (Michael & Shirley, 2000) (Michael, et al., 2000). The primary focus here is on how to read the graph. Let's see the picture (Figure 2).

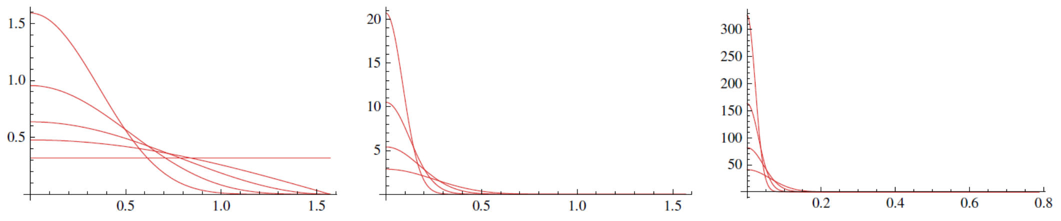


Figure 2. The Phong distribution with logarithmically spaced cosine powers (Hoffman, 2012)

On the left, α_p (cosine power) values vary from 0 to 8. In the middle they range from 16 to 128, and on the right they go from 256 to 2048.

It is not easy to understand what x-axis stands for at first glance. It tells the angle between the normal of the macro-surface and the normal of the micro-surface. For convenience, the angle can be represented as a cosine term. It is also called the "roughness" value. The y value at the origin indicates the number of samples which have the exact same normal as the macro-surface. In other words, the y value at $x = 0$ is the mean value of the Phong NDF. As the x value increases, the y value decreases exponentially with respect to the variance of NDF.

In Figure 2, all distribution functions have different forms in accordance with the α_p. As the α_p gets bigger, the variance gets smaller – the form of graph becomes sharper. In BRDF, a sharp graph of the NDF means that the angle between the micro-scale normal and macro-scale normal is small – it indicates that the surface orientation at microscopic scale is smooth. Generally, we say that the surface is smooth, which results in sharper reflections for incoming rays.

2.2.3 The Beckmann distribution

In the previous example, we dealt with how to analyze the NDF graph. This helps you gain more insight into the NDF. For a more complete understanding, we are going to compare the last example with a different type of BRDF NDF called the Beckmann distribution (Ward, 1992).

Compared to Phong distribution, it has slightly different NDF graph shape. Let's see the Beckmann NDF below:

$$D_b(m) = \frac{1}{\pi\alpha_b^2(n\cdot m)^4} e^{-\left(\frac{1-(n\cdot m)^2}{\alpha_b^2(n\cdot m)^2}\right)}$$

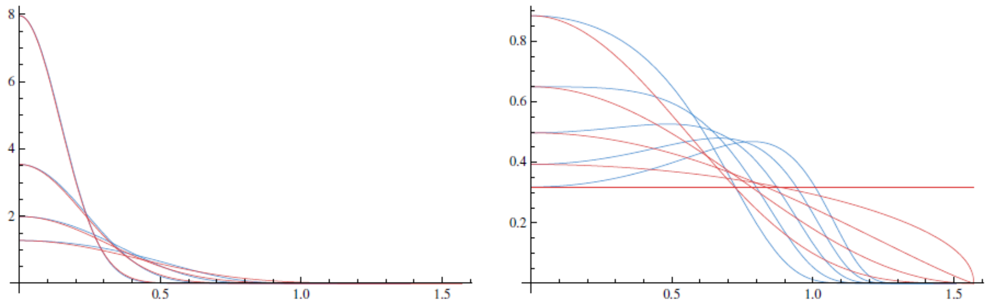


Figure 3. Comparison of Beckmann (blue) and Phong (red) distributions (Hoffman, 2012)

The function looks fairly different from the Phong NDF, and it seems a little bit more complicated. However, we don't have to worry about it. The parameterization for roughness and the normalization of the distribution function make it more intricate. After analyzing the equation and the graph, you will realize that what it try to say is relatively simple, contrary to its appearance.

On the left, we see that the two are very similar for smooth surfaces (values of α_b ranging from 0.2 to 0.5). On the right, we see that they diverge somewhat for moderately rough surfaces (values of α_b ranging from 0.6 to 1.0) (Figure 3).

First we need to clarify the meaning of α_b. Increasing the values of α_b means increasing the average micro-facet slope – the roughness. So, as α_b increases, the variance becomes bigger. Different from the α_p in Phong distribution, α_b is normalized, ranging from 0 to 1.0. As α_b approaches 1.0, the microfacet surface is rougher, otherwise it is smooth.

With the definitions above, see the two graphs above (Figure 3). We can't observe any difference between Phong distribution and Beckmann distribution when the surface is smooth (the left image). However, on the rough surface, there is a noticeable difference. Let us focus on the second graph, because you can understand the first graph in the same manner as before. The peaks on a rough surface are different in the Beckmann NDF – the mean value moves to the right. From this, we can interpret that the distribution of micro-scaled normals is away from the macro-scaled normal. Theoretically, we can say that the Beckmann distribution better describes the condition of micro-facets than the Phong distribution.

After investigating the graph of the Beckmann distribution, we need to concentrate the meaning of a peak in various NDFs – a peak represents the mean angle between micro-scale normals and the macro-scale normal. From this example, we can see that both the variance and the mean value are crucial for describing the state of a surface's micro-facets.

2.2.4 Conclusion

So far, we have looked through the Normal Distribution Functions widely used in Physically Based Shading (PBS). The BRDF is not the only application of the NDF. The NDF is also used to store multiple macro-geometry normals into a voxel in forms of Gaussian Lobe or Spherical Harmonics (Fournier, 1992) (Han, et al., 2007). The principles of NDF for Gaussian Lobe are basically same. Like this, the NDF itself is very important in computer graphics. Especially, in global illumination, the NDF adequately well-describes the behavior of physical lighting using limited bandwidth. It is essential to have a complete an understanding of the NDF to begin to understand BRDF; this is why we have covered and shown examples of key features of the NDF.

2.3 Shadow Mapping

Shadow mapping is one of many real-time algorithms developed to generate shadows in a scene. One might be curious as to why a shadow generation technique appears in a paper about global illumination; this is because we will be utilizing shadow mapping's Virtual Point Lights (VPLs). Most real-time Global Illumination algorithms have a common step − injecting virtual point lights (VPLs) (Keller, 1997) into the scene. To store VPLs, we use a technique called Reflective Shadow Mapping (RSM), which stores all data required for lighting calculation − such as depth, diffuse albedo, and normal − into the texture. As the name implies, RSM is a shadow mapping algorithm, so it is decided to cover this topic. First, we will examine the shadow mapping algorithm briefly. To increase the quality of shadows, we will also look at using Cascade Shadow Maps (Dimitrov, 2007) (Zhang, et al., 2007). When either lights or camera are moved in the scene, we risk producing artifacts in GI where indirectly calculated illumination causes blinking on the rendered frame. This results from inconsistency when injecting light information into the shadow map. To remove these artifacts, we should use stable cascade shadow maps (Valient, 2007).

2.3.1 Shadow mapping

The shadow mapping algorithm is widely used in real-time shadows. In the pursuit of the higher quality shadows, many varieties of shadow mapping have appeared, but the underlying principles remain the same (Figure 4[1]).

1. Generate depth map from light point of view
2. Render the scene and transform world position into light space
3. Compare the transformed depth and depth from depth map
4. Generate shadow mask from comparison

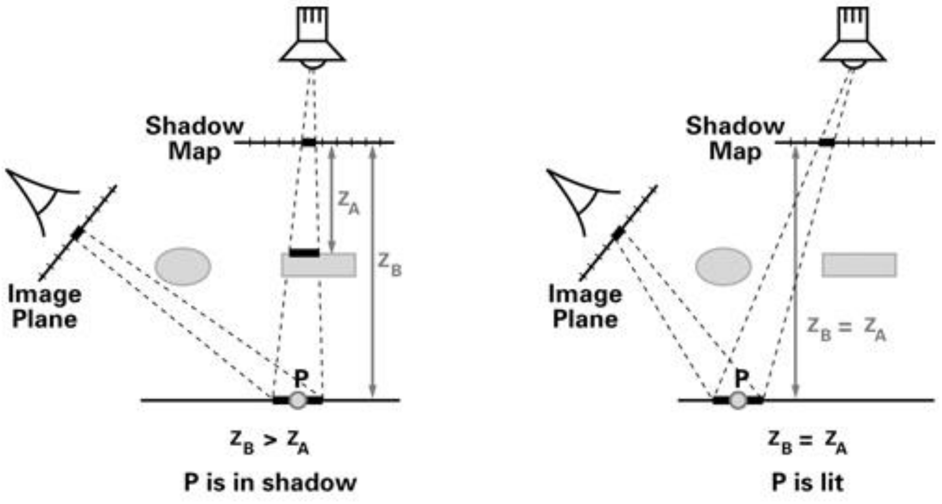


Figure 4. Shadow Mapping Depth Comparison

We will cover the details of using Reflective Shadow Maps (RSM) later. It is enough for now to know that the RSM basically works in same way as shadow mapping.

2.3.2 Cascade Shadow Maps

The cascade shadow maps (Dimitrov, 2007) came out to make up for the artifacts produced when the shadow map tries to cover a large scene in games – blocky artifacts inevitably appear around shadows, resulting from the lack of depth map resolution. Shadows are a big contributor to a realistic environments. As blocky artifacts become noticeable, it can decrease the immersion in the game. In genres like First Person Shooters (FPS) or Adventure Games in which it is important

[1] http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html

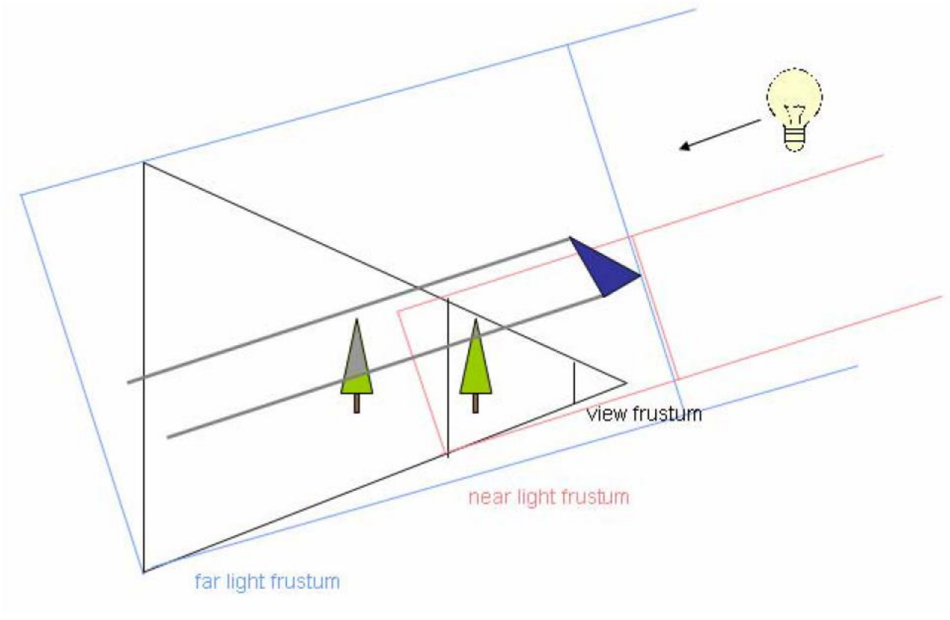


Figure 5. A triangle casting a shadow in multiple depth maps (Dimitrov, 2007)

to present large, realistic scenes to the users, these artifacts provide users the worst experience. Cascaded shadow maping is a popular way to compensate for flaws in shadows.

The idea is to partition the view frustum depending on the distance from the eye's point of view and to write the depths in accordance to each respective view frustum. The objects located in the far view frustum are shaded in a few pixels, while objects near the view frustum are shaded on more pixels of the shadow maps. As a result, we can get high-quality shadows without higher resolution shadow maps.

Figure 5 explains well why the CSMs are able to give better shadow quality. Supposing the same depth map resolution, objects near the light frustum affect a small space, while those far from the light frustum cover a bigger area. For objects far away from a viewer, we don't need to get an accurate depth value. By getting more accurate depth values for objects near the viewer, CSMs increase the overall quality of shadows.

Now we need to think about the chosen method of splitting the view frustum. One of techniques is Parallel-Split Shadow Maps (PSSM) (Zhang, et al., 2007). We can use the practical split scheme:

$$C_i = \rho C_i^{log} + (1 - \rho) C_i^{uni}, \qquad 0 < \rho < 1$$

$$C_i^{log} = n \left(\frac{f}{n}\right)^{\frac{i}{m}}$$

$$C_i^{uni} = n + (f - n)\frac{i}{m}$$

Where C_i^{log} and C_i^{uni} are the split positions in the logarithmic split scheme and the uniform split scheme, respectively.

Sticking to one split scheme might not give the quality that you want. Moderate-sampling requires the combination of logarithmic split scheme and the uniform split scheme in adequate ratios (Figure 6).

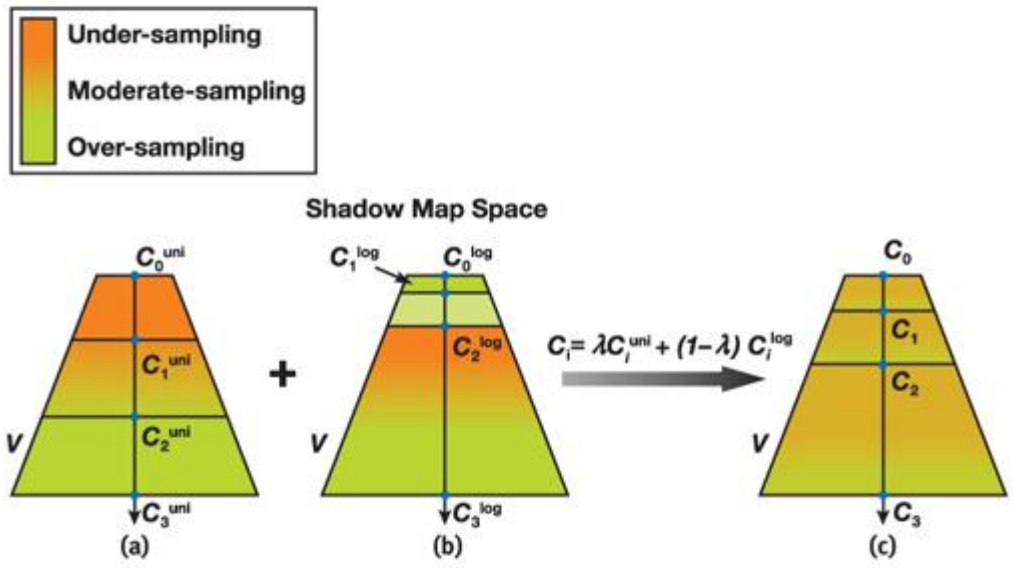


Figure 6. Visualization of the Practical Split Scheme in Shadow-Map Space (Zhang, et al., 2007)

With CSMs, we can get better shadows in a large space. The normal CSMs can give you weird shimmering or swimming artifacts. The artifacts could make gamers notice noisy behavior on shadows. We already had mentioned that shadow mapping is used to inject lighting into volumetric data set sin global illumination. This could lower the quality for injecting photons with RSMs, causing the same shimming artifacts. So we need to stabilize the CSMs.

2.3.3 Stable Cascaded Shadow Maps

The stable cascaded shadow maps (Valient, 2007) is essential to reduce the shimmering effects on CSMs. The shimming effect is caused by inconsistent boundaries in shadows whenever a camera moves. We need to know the main source causing this artifact. The main reason is the discretization of the scene into shadow map pixels.

In Figure 7, let's compare the pictures (a) and (b). With these images, we can understand where the artifact comes from. From (a) to (b), rotation of camera occurs. In each image, we can see the rectangle boundary. Imagine these rectangles as shadow maps. The (b) alignment of shadow maps are dislocated compared to previous status (a). In this phenomena, texels that should have the same values before the camera movement could have different depth values because of shadow map texture aliasing. Whenever you rotate the camera, the shimming takes place by changing depth values on shadow boundaries in depth maps. Whenever we are zoomed in, a similar effect also occurs with the same reason.

How can we remove this artifact? One method is to create a cascade frustum completely enclosed in a square cylinder. We need to make sure our shadow map covers the entire sphere to guarantee that every point in the view frustum is covered by a valid shadow map texel.

The last thing required for stable CSMs is to prevent the situation where, when moving the shadow map, the world space position might fall into a different shadow map texel than it was located in the previous frame – the relative position within the shadow map should stay the same. This process is called Texel Snapping. To do this, we need to place the shadow map corners at discrete positions only, quantizing the step for the shadow map texel.

At the end, the depth maps become stabilized on translation or rotation. In other words, we are able to inject consistent intensity lights via RSMs.

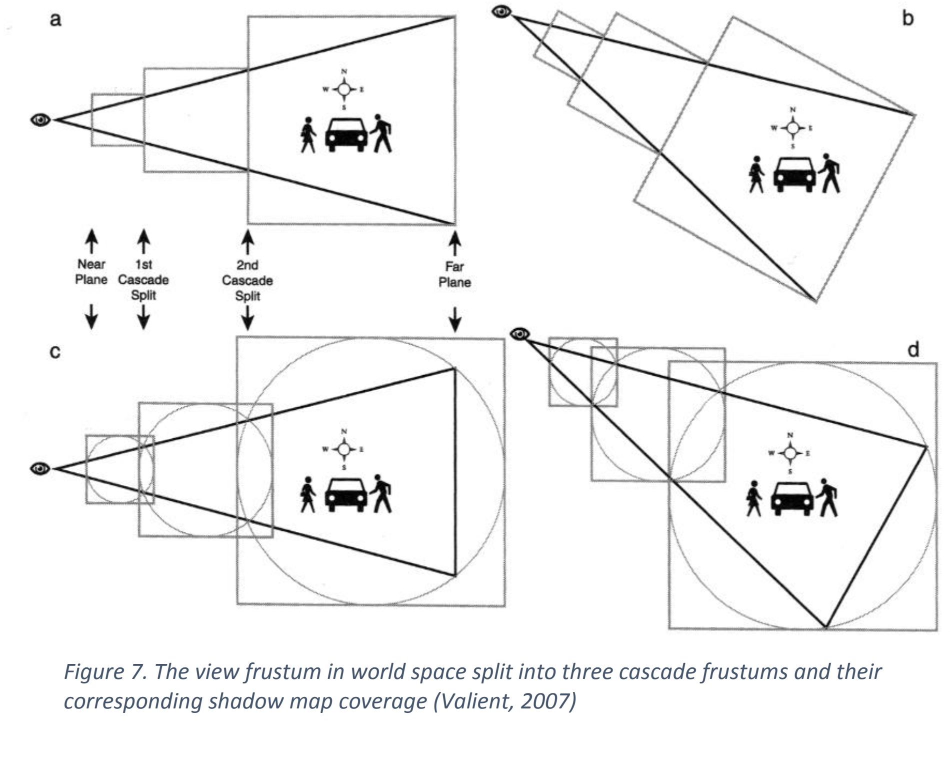


Figure 7. The view frustum in world space split into three cascade frustums and their corresponding shadow map coverage (Valient, 2007)

2.4 Directional data compression models

In volumetric rendering, we need to store multiple layers of directional data − like geometry normals or light directions − into one voxel. As the number of layers to preserve increases, the required memory space is increased exponentially. This is because we keep these layers of data in 3D formats such as 3D textures. Therefore, we need a special methodology where, even though we lose some precision in these data, we can better approximate it with a small memory bandwidth. We can get around this by compressing them into limited channels in GPU memory. Here we are going to look through two representative methods called Spherical Harmonics and Gaussian Lobes. Each one approaches a different principal for storing the data, but they are quite similar in using convolution in signal-space. The details will be covered in following order — first Spherical Harmonics (SH) (Green, 2003) (Schönefeld, 2005) (Slomp, et al., 2006) then Gaussian Lobes (Tsai & Shih, 2006).

2.4.1 Spherical Harmonics

Spherical Harmonics have been used widely in computer graphics, developed in the search for a solution to real-time indirect illumination. The traditional way to calculate indirect lighting is through Ray-Tracing. But this method is computationally very heavy, so it can't afford to be adopted in real-time rendering. For dynamic lighting, it is chosen to store all necessary data for lighting calculation in advance, and process them with dynamic lighting parameters in real-time. To do that, the SH has been used for various global illumination algorithms in computer graphics.

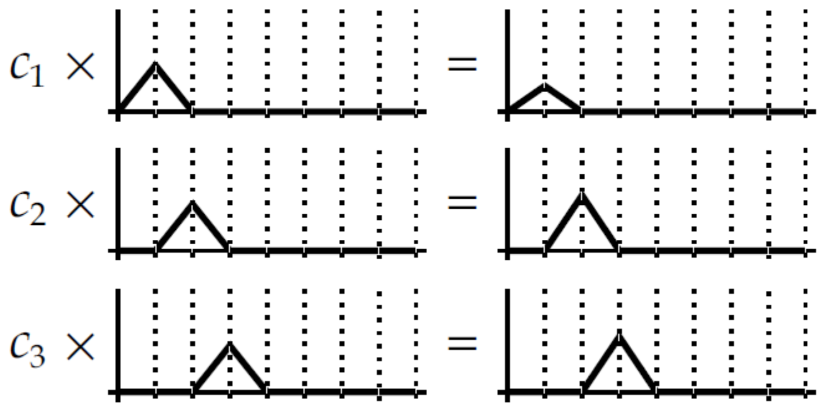


Figure 8. Pieces of basis function and following coefficients (Green, 2003)

2.4.1.1 Introduction

Before looking into the details of Spherical Harmonics, we need to know the basis functions. The basis function can be thought of a small piece of signal that can be combined or scaled to produce an approximation of the original one – it could be straight forward to regard a basis function as unit vector in linear algebra. The combination of different sets with unit vectors produces any arbitrary vector. Likewise the combination of the basis function B_i and coefficient C_i can approximate any original function $\mathrm{F(x)}$ (Figure 8).

The projection of some original function produces a set of coefficients, where each coefficient corresponds to each basis function (Figure 9). What is the projection in function space? It is easier to understand it as the projection of vectors in Linear Algebra. Geometrically, the projection of a vector A into vector B results in the coefficients on vector B, which means the combination of the coefficients and vector B can restore vector A – the dot product for projection in Linear Algebra.

The biggest difference between geometric projection and functional projection is the accuracy of the reconstructed data. For geometric projection, we can restore the original vector by computing the dot-product of the basis vector and the corresponding coefficients, because the vector in use is composed of a limited tuple of floating values (e.g. 3D or 2D vectors in computer graphics). On the other hand, the function is fundamentally continuous. That is to say, we can't recover the exact function by the inner-product. The inner-product is the projection of the original function onto basis functions, and is similar to dot-product. So, we need to approximate the ordinary function provided in limited space. For indirect diffuse lighting, we don't need to be

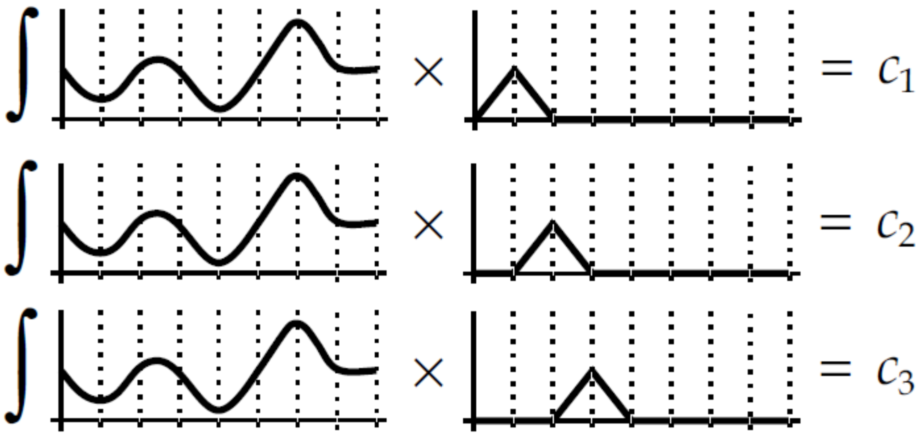


Figure 9. Produce coefficients projecting into basis functions (Green, 2003)

accurate for global illumination calculation. The details will be covered in the global illumination chapters after introducing the SH.

With piecewise projections, we could get partial functions to approximate the original function. The sum of partial functions give us the approximated ordinary function (Figure 10).

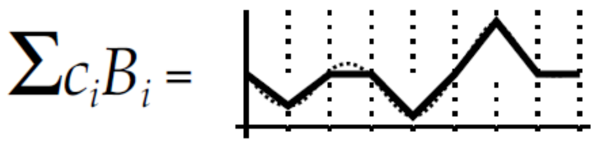


Figure 10. The sum of piecewise functions result in approximated function (Green, 2003)

In the examples, we have used a set of linear basis functions to produce piecewise linear approximations. At last, these approximations reproduce the original input function. Among various types of basis functions, the orthogonal basis function is used in the SH. When we integrate the product of any two of them, if they are the same, we can get a constant value, and if they are different we get zero:

$$\int_{-1}^{1} F_m(x)F_n(x)dx = \begin{cases} 0\ for\ n\ \neq m \\ c\ for\ n = m \end{cases}$$

We can make more a rigorous rule, that the integration of the product of two approximated piecewise functions must return 0 or 1 by the orthonormal basis functions. It means that we can break a basis function into its component sine waves.

One of these polynomial families for the SH is the Associated Legendre Polynomials. To understand Spherical Harmonics, we need to know some basic properties of Associated Legendre Polynomials. There are three following properties:

1. $(\mathrm{l}-\mathrm{m})P_l^m = x(2l-1)P_{l-1}^m - (l+m-1)P_{l-2}^m$
2. $P_m^m = (-1)^m(2m-1)!!\,(1-x^2)^{m/2}$
3. $P_{m+1}^m = x(2m+1)P_m^m$

The three rules for the Associated Legendre Polynomials are essential to produce Spherical Harmonic basis functions. Keeping them in mind, let's start to deal with the SH.

2.4.1.2 Spherical Harmonics

Normally when discussing SH, we refer to the 3D version of the Fourier Transform. To understand SH, understanding the background of the Fourier Transform helps. The Fourier Transform is the transformation of discrete data set in the spatial domain into an analog data set in frequency space. From the point of view of approximating data into the signal dimension form, the Fourier Transform looks similar to Spherical Harmonics. So we can say that the SH is the 3D form of Fourier Transform. But for someone who doesn't have sufficient knowledge of the Fourier Transform, it is hard to get complete insight into the projection of a data set into signal space. In fact, the SH can be described by one sentence: the SH is a compression technique that stores the directional data with limited memory bandwidth via the projection onto the piecewise spherical harmonic basis function. With this definition, we don't need to care about what the Fourier Transform is. But it is highly recommended to read related papers for the Fourier Transform to get a deeper understanding of the SH (Schönefeld, 2005) (Sloan, 2013).

We continuously said that the SH is the projection of direction data onto basis functions. Precisely, the SH is only interested in approximating real functions over the unit sphere. As a domain, the sphere is a complete data set over all unit directions. Basically, the SH is based on treating this data as a function, rather than geometric data. But if we reflect the SH concept into geometric principles in Linear Algebra, we can get the idea easier. Imagine the basis functions as unit vectors and the original function as input data. We can think of the projection as storing input data in accordance to direction. Like in Linear Algebra, where we transform one vector currently in local space to another local space, we can restore the original function with the SH coefficients and SH basis functions. Until now we have reviewed the SH using a suitable geometric analog, but now let's see the spherical harmonic basis function:

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2}K_l^m \cos(m\varphi)\, P_l^m(\cos\theta), m > 0 \\ \sqrt{2}K_l^m \sin(-m\varphi)\, P_l^{-m}(\cos\theta), m < 0 \\ K_l^0 P_l^0(\cos\theta), m = 0 \end{cases}$$

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-\ |m|)!}{(l+\ |m|)!}}$$

The above is the basic form of the SH basis functions. If you look closely, we can notice the Associated Legendre Polynomials. By combining the Associated Legendre Polynomials, the SH functions can have orthonormal properties (Schönefeld, 2005).

As parameters of SH function, it have two inputs (θ, φ). As basis functions over the sphere, we use spherical coordinates. We can also produce the SH function in Cartesian coordinates, which we will cover later. One last thing to beware of is the range of m. Different from the associated Legendre polynomials, m takes signed integer values from –l to l. For convenience, to store the

SH function in limited channels like float3 or float4, we can flatten them into a vector, which we can also define like this:

$$y_l^m = y_i \; where \; i = l(l+1) + m$$

When we calculate indirect diffuse lighting, we don't consider that to be very accurate because diffuse lighting itself is low-frequency. When storing SH coefficients and SH functions in the render-targets, it is very useful to convert them into a 1D form for linear combination.

From the previous definition, the SH function is composed of sine and cosine wave. The cosine and sine wave is a cycle over negative -1 to positive +1. The green indicates positive values, and the red indicates negative values (Figure 11) (Figure 12).

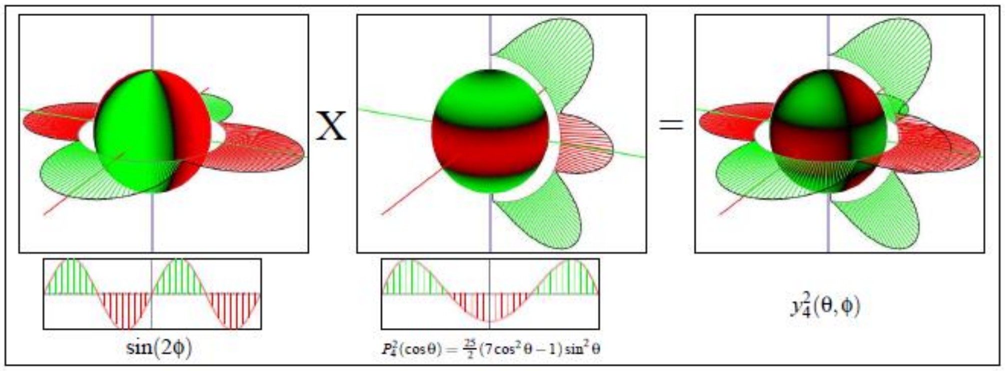

X
=
$\sin(2\phi)$
$P_4^2(\cos\theta) = \frac{15}{2}(7\cos^2\theta - 1)\sin^2\theta$
$y_4^2(\theta,\phi)$

Figure 11. Demonstration of the functional dependencies of both coordinate axes (Schönefeld, 2005)

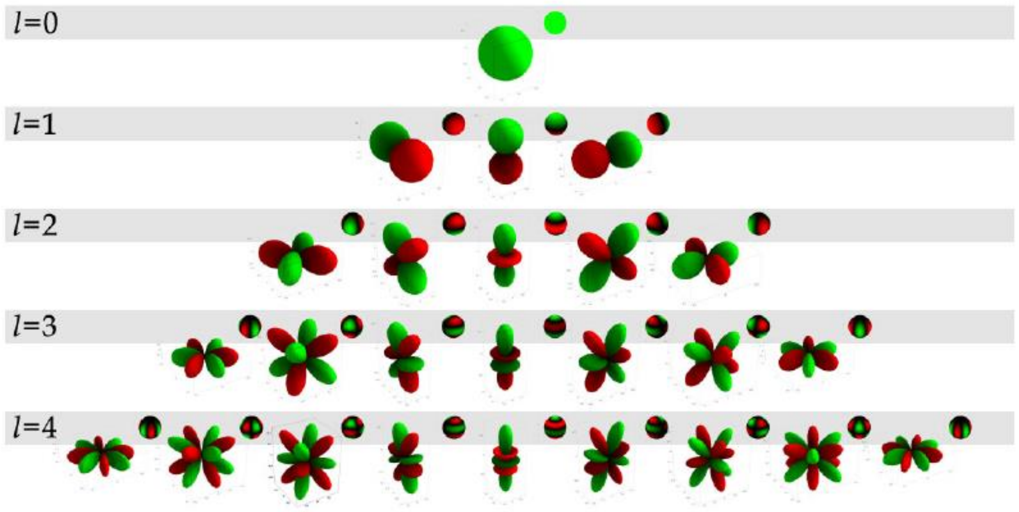


Figure 12. 5 SH bands plotted as unsigned spherical functions by distance from the origin and by color on a unit sphere (Green, 2003)

One thing to be mentioned is that each band is approximated as one cycle per sphere – the linear combinations give us very good approximations of the cosine term in the diffuse surface reflectance model.

2.4.1.3 SH Projection

The SH Projection is performed to produce the coefficients on the SH basis and to restore the original function with a linear combination of the coefficients and SH functions. We already have covered this in a previous chapter. Before formulizing SH projection in equations, let's see a concrete example:

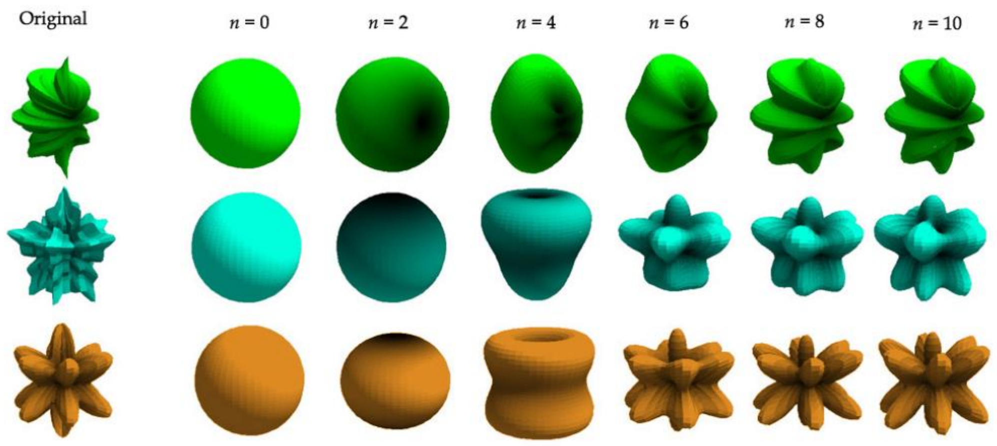


Figure 13. SH projection of functions with increasing orders of approximation (Green, 2003)

Figure 13 describes one property of SH approximation. With higher frequency SH bands, it better approximates the original function. But this accompanies an increase in memory space, as there are more SH coefficients to be saved. For indirect illumination for diffuse terms, it is enough to consider using at most n = 2, because the diffuse term doesn't need to be accurate; even a mid-frequency specular term is enough for n = 2. But when we consider high-frequency specular term, we should consider having more bands for SH coefficients.

2.4.1.4 SH Properties

We can classify SH properties into four categories: Orthonormal, Rotational Invariant, Transfer matrix and Rotation in the SH. These properties are important to utilize the SH properly.

The Spherical Harmonics are not only orthogonal but also orthonormal. When two SH basis functions are integrated $y_i y_j$ for any i and j, the following rule is expected.

$$y_i(x) y_j(x) = \begin{cases} 1, i = j \\ 0, i \neq j \end{cases}$$

And the SH is also rotational-invariant. If a function f is a rotated copy of function g then it meets:

$$\tilde{f}(x) = \tilde{g}(R(x))$$

In other words, the projection of a rotated function f gives exactly the same result as if we had rotated the input to f before the SH projection. In indirect diffuse lighting, regardless what happens to an SH function (like rotating the scene or character), the light intensity has no change.

The third feature is closely related the second one, the orthonormality. Suppose that we are integrating a lighting function and a transfer function. The transfer function often mirrors shadowing, reflectance or refraction (Ramamoorthi & Hanrahan, 2001) (Oat, 2004). We can think of participants for global illumination. To integrate the lighting function and transfer function over the sphere, we can do our calculation with the coefficients derived from the SH projection like this:

$$\int \tilde{L}(s)\tilde{T}\,(s)ds = \sum_{i=0}^{n^2} L_i T_i$$

The integration over the sphere is simplified into a single dot-product. In general, we say this form is a convolution of the SH coefficients. The convolution represents a dot-product, meaning integration over a certain domain. Like this, the calculation of integrations can be reduced by the convolution of two SH coefficients. In this case, we need the Triple Tensor for the SH. We will cover this later in the Precompute Radiance Transfer section.

We can also convert the spherical harmonics in spherical coordinates into the SH in Cartesian coordinates like this

	$m = -2$	$m = -1$	$m = 0$	$m = 1$	$m = 2$
$l = 0$			$\frac{1}{2}\sqrt{\frac{1}{\pi}}$		
$l = 1$		$\frac{1}{2}\sqrt{\frac{3}{\pi}}\frac{y}{r}$	$\frac{1}{2}\sqrt{\frac{3}{\pi}}\frac{z}{r}$	$\frac{1}{2}\sqrt{\frac{3}{\pi}}\frac{yx}{r}$	
$l = 2$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{yx}{r^2}$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{yz}{r^2}$	$\frac{1}{4}\sqrt{\frac{5}{\pi}}\frac{2z^2 - x^2 - y^2}{r^2}$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{zx}{r^2}$	$\frac{1}{2}\sqrt{\frac{15}{\pi}}\frac{x^2 - y^2}{r^2}$

$$\mathrm{r} = \sqrt{x^2 + y^2 + z^2}\ (usually\ r = 1)$$

The detail of derivation refers to (Sloan, 2008). The Cartesian version of the SH is useful a symbolic expression of the projection for indirect lighting.

One of the difficult part in the SH is rotating SH coefficients. Basically the SH function is rotationally invariant, but how do we rotate an SH projected function? In here we also focus on the orthogonality, implying that the coefficients don't interact each other between different bands of SH functions. So we can construct the rotation matrix as a sparse block diagonal sparse matrix to map an un-rotated SH coefficients vector into a rotated one:

$$Z_{\propto} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \cos\propto & 0 & \sin\propto & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\sin\propto & 0 & \cos\propto & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cos 2\propto & 0 & 0 & 0 & \sin 2\propto \\ 0 & 0 & 0 & 0 & 0 & \cos\propto & 0 & \sin\propto & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\sin\propto & 0 & \cos\propto & 0 \\ 0 & 0 & 0 & 0 & -\sin 2\propto & 0 & 0 & 0 & \cos 2\propto \end{bmatrix}$$

For Euler rotation $R_{SH}(\alpha, \beta, \gamma)$, we have several options – XYZ rotation, ZYZ rotation or ZYX rotation. We concentrates on ZYZ rotation for the merit that we can get away with the only two rotations. For y-axis rotation, we can express z-axis rotation with constant x-axis rotation around $-90°$ and $90°$ like this:

$$R_{SH}(\alpha,\beta,\gamma) = Z_{\alpha}X_{-90°}Z_{\beta}X_{90°}Z_{\gamma}$$

By this we only need two type of rotation formats – z-axis and x-axis. But for x-axis rotation, we don't need to calculate x-axis rotations for each SH rotation. It is constant rotation matrix.

Bringing constant x-axis rotations into SH rotation for first two bands, it looks like this:

$$R_{SH} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \begin{pmatrix} \cos\alpha\cos\gamma \\ -\sin\alpha\sin\gamma\cos\beta \end{pmatrix} & \sin\alpha\sin\beta & \cos\alpha\sin\gamma + \sin\alpha\cos\gamma\cos\beta \\ 0 & \sin\gamma\sin\beta & \cos\beta & -\cos\gamma\sin\beta \\ 0 & \begin{pmatrix} -\cos\alpha\sin\gamma\cos\beta \\ -\sin\alpha\cos\gamma \end{pmatrix} & \cos\alpha\sin\beta & \cos\alpha\cos\gamma\cos\beta - \sin\alpha\sin\gamma \end{bmatrix}$$

We can make ZYZ rotation into analytic solution. This form helps make debugging easier, but we should change the rotation version to ZYZ rotation for your engine. Considering performance in calculating these SH rotation matrix, adopting ZYZ rotation is sufficiently worthwhile.

However, rotating the SH projected function is still computationally heavy. There is another method to relieve the calculation using Zonal Harmonics (Nowrouzezahrai, et al., 2012).

2.4.1.5 Conclusion

We have covered an overview of Spherical Harmonics. We tried to explain the SH in detail for its importance in global illumination. You have enough insight to read real-time global illumination algorithms. Some applications of the SH will be handled in chapter following Gaussian Lobes.

2.4.2 Gaussian Lobes

The Gaussian Lobe is one of techniques used to store multiple directional data into voxels, introducing the Normal Distribution Function (NDF). We covered the NDF in the previous local illumination model section. The NDF in the BRDF describes the behavior of light rays after interacting materials in terms of diffuse or specular reflectance. The Gaussian Lobe uses the NDF to store the directional data into voxels for the same reasons, accounting for the interaction between lighting and materials.

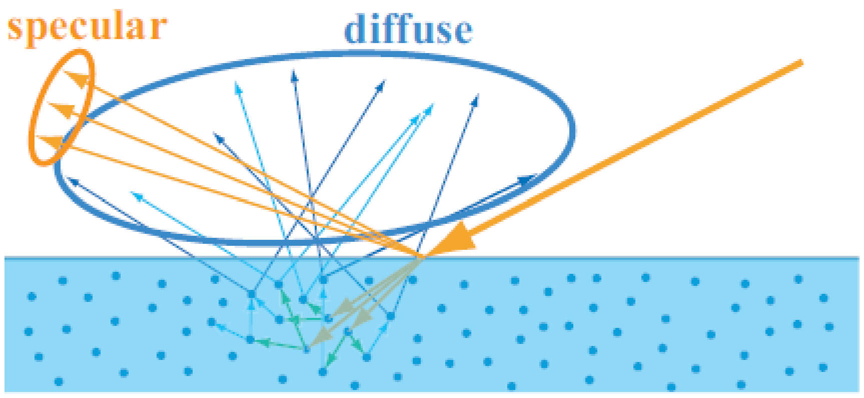


Figure 14. The diffuse and specular ray behaviors in microfacet on the material (Hoffman, 2012)

In Figure 14, we can see specular and diffuse behavior on the material microscopically. When we gather all these rays into one point from micro-scale to macro-scale, we can observe the Gaussian Lobe shapes (Figure 15).

The NDF explains the shape of Gaussian Lobes. For example, when the variance of the NDF is bigger, the shape of Gaussian lobe is wider. As the variance is smaller, the shape becomes sharper. We can generalize that the diffuse reflectance has a high variance – resulting in a wide Gaussian Lobe – while specular reflectance has small variance – producing a narrow Gaussian lobe. For diffuse terms, we generally say the cosine lobe only depends on the surface normal and light direction.

What is difference between the Spherical Harmonics (SH) and the Gaussian Lobes? The SH writes the data over an omnidirectional domain – the unit sphere domain – with basis functions, whereas the Gaussian lobes use the main direction vector as the mean value, and the variance is obtained by using the NDF. To describe multiple layers of directional data, we can use a mixture of multiple Gaussian Lobes (Xu, et al., 2013) (JiapingWang, et al., 2009). They are different in the ways they store data, but for the purpose of storing the directional data in limited memory space, the point is the same.

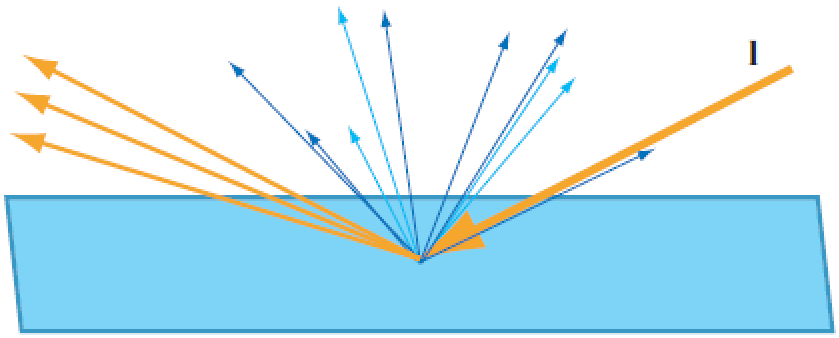


Figure 15. Diffuse and Specular Gaussian lobes in BRDF (Hoffman, 2012)

2.4.2.1 The Normal Distribution Function in Gaussian Lobes

In this section, we look into the NDF of a Gaussian Lobe in detail. As we guess from the name, the Gaussian Lobe mainly uses a Gaussian distribution function. The definition is:

$$\mathrm{f(x)} = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2 / (2\sigma^2)}, \qquad \textit{where } \mu \textit{ is mean and } \sigma^2 \textit{ is variance}$$

The basic properties for global illumination are the same as the NDF that we covered in the local illumination model (BRDF). As the variance increases, the distribution of values deviates farther from the mean value. If the $\sigma^2 = 0$, all values are in the mean value.

Before going over an example of Gaussian Lobes in global illumination, we should clarify the different types of Gaussian Lobes first. There are two types of Gaussian lobes – one for isotropic shape and the other for anisotropic shape. The biggest difference is that the isotropic lobes are symmetric around the origin, but anisotropic lobes are not symmetric (Tsai & Shih, 2006) (Hsu & Kakade, 2012) (Figure 16).

For the isotropic Gaussian lobe, it is enough to express its Gaussian distribution function in 1D, exploiting the symmetry around its direction. But for anisotropic Gaussian Lobe, it is necessary to use multivariate variables to represent multi-directional distribution.

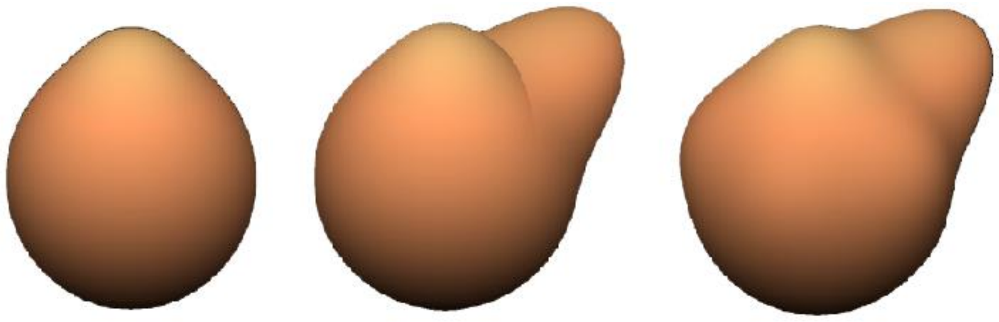


Figure 16. Left is isotropic Gaussian Lobe, right two images are anisotropic Gaussian Lobes; the middle is mixture of two Gaussian Lobes and the right one is the mixture of four Gaussian Lobes. (Tsai & Shih, 2006)

In global illumination, the Gaussian Lobe is one option to store a 3D directional data set. So, suppose that we are going to store all the 3D normals falling on a voxel. With the isotropic model, we could set distance from the average normal direction as a variable for the Gaussian distribution (Toksvig, 2004). When the distance between each normal in a voxel and the main direction of Gaussian Lobe is zero, the variance is zero. As this distance increases, the variance of the NDF does too. In global illumination, the diffuse reflectance takes a wide range of reflection, using a cosine lobe with a high variance, while the specular reflectance tends to have narrow range of reflection indicating a small variance. In general, not all lighting behaviors will be as predictable as isotropic. To describe a more accurate shape for reflectance, it is sometimes useful to use anisotropic Gaussian lobes. But for real-time rendering, the isotropic model provides a satisfactory approximation.

We can find a similar pattern presented in the SH – if we increases the number of bands in computing the SH, it becomes more accurate but reduces overall efficiency of both memory space and the performance of calculations. Both isotropic and anisotropic models, follow the same pattern. If we decided to adopt the anisotropic model, it would take more memory space to store multiple means and variances for multivariate Gaussian distributions and heavy calculations for higher Gaussian distributions.

2.4.2.2 Construct Gaussian Lobes

One of ways to construct Gaussian lobes is to average a set of vectors (Toksvig, 2004). The variance of a Gaussian Lobe is calculated by the length of the averaged vectors. With the variance, an NDF is constructed, which produces the isotropic Gaussian Lobe model. The variance is approximated to:

$$\sigma_N^2 = \frac{1 - |N_a|}{|N_a|}, \qquad \text{where } N_a \text{ } is\ average\ directional\ data$$

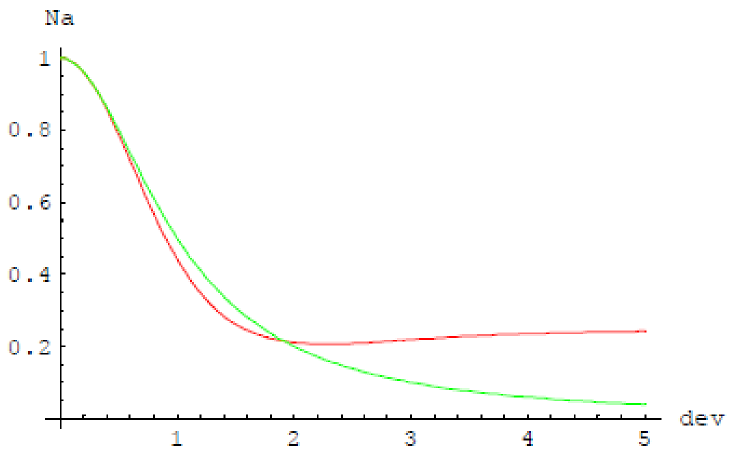


Figure 17. Average normal as function of Standard Deviation, (Red), and Approximation (Green) (Toksvig, 2004)

This is useful when you store multiple layers of directional data into each voxel. For example, we could put all geometric normals into the distribution. First, we average the normals and get the length of the average normal. Submitting the length as an input for the above equation, we can get variance. Finally, we can model the isotropic Gaussian Lobe.

2.4.2.3 Convolution

How does one integrate different two Gaussian lobes? For instance, we could have a Gaussian lobe for geometry normals and a BRDF Gaussian lobe. The integration of two Gaussian lobes can be calculated by their convolution (Han, et al., 2007) (Figure 18).

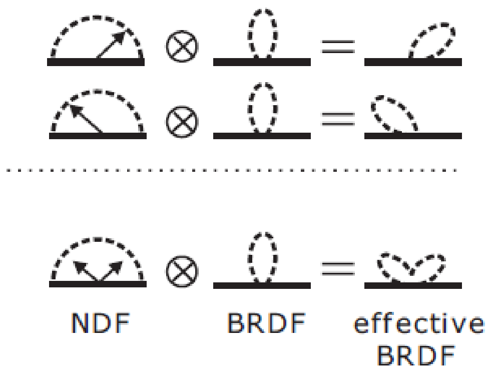


Figure 18. Convolution between NDF and BRDF (Han, et al., 2007)

The BRDF Gaussian lobe is fixed in the local frame. If we want to make an effective BRDF over geometry normals stored as a distribution, the convolution gives exactly what we desire (Figure 18). In another example, we can get the cosine term in a voxel used for lighting calculation by convolving the geometry normal Gaussian lobe and the light direction Gaussian lobe.

For our complete understanding, take the last example. Suppose we have two normals in a voxel. Two normals are in opposite directions, which causes a large variance. That is to say, convolving any Gaussian lobe with this Gaussian lobe gives you a result close to zero. This means the effective probability for the convolution is almost none. Convolving two different lobes results in another new lobe. To make use of the resultant Gaussian lobe, the probability density value from the Gaussian distribution function should be utilized. The detail for this will be covered in Voxel Cone Tracing (VCT).

2.4.2.4 Conclusion

In this section, we have described Gaussian lobes. The Gaussian lobe is a popular method to describe the directional behavior of light, along with the Spherical Harmonics (SH). For performance concerns, the isotropic lobe model is more widely used in real-time global illumination than the anisotropic lobe model. In the case of volumetric rendering, the Gaussian Lobe model has started to receive lots of interest. In spite of the model's limitations in real-time use, it gives well-approximated data in each voxel. We will see the potential for real-time global illumination in the VCT.

2.5 Real-Time Global Illumination

2.5.1 Precomputed Radiance Transfer (PRT)

Precomputed Radiance Transfer (PRT) (Slomp, et al., 2006) (Sloan, et al., 2002) started from the inspiration to allow lighting calculations in real-time. In the past, complicated lighting calculations were too tough for the hardware to handle. To overcome this limitation, the PRT came out. In this section, we are going to cover the general PRT algorithm. As the representative application of Spherical Harmonics (SH) in Global Illumination (GI), we can understand its usage further. In this chapter, we will overall examine the GI theories from scratch. After this chapter, we can get more complete insight into realistic GI.

2.5.1.1 Introduction

Before going into the PRT, we need to know what the PRT exactly is. When illustrating a realistic scene, there are many complicated interactions between light sources and objects in the scene. We can classify these interaction into shadows, inter-reflection or refraction, caustics and subsurface-scattering. It is too computationally heavy for the hardware to calculate them all in real-time. For the real-time implementation, we need to make some assumptions and approximate them by precomputing the lighting calculations in advance. In PRT, it is assumed that all objects to be precomputed are rigid. The requirement that objects are static becomes the main limitation, but we can achieve it in real-time.

We want to emphasize the importance of the mathematical theories behind PRT. There are several things that need to be mentioned. The first one is the rendering equation. This equation appears often in lots of global illumination papers, which implies its importance:

$$\mathrm{L}(\mathrm{p} \rightarrow \vec{d}) = L_e(p \rightarrow \vec{d}) + \int_{\Omega} f_r(p, \vec{s} \rightarrow \vec{d}) L(p \leftarrow \vec{s})\, H_{N_p}(-\vec{s})\mathrm{ds}$$

The rendering equation contains all components necessary to calculate realistic lighting. $\mathrm{L}(\mathrm{p} \rightarrow \vec{d})$ is a total light intensity in direction of $\vec{d}$ from the point p (Figure 19).

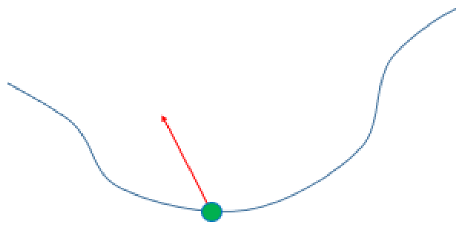


Figure 19. Radiance leaving point p (green) in direction (red)

Total light intensity consists of emissive lighting on the point p, where $\int_{\Omega} f_r(p, \vec{s} \rightarrow \vec{d}) L(p \leftarrow \vec{s})\, H_{N_p}(-\vec{s})\mathrm{ds}$ represents the total intensity over hemisphere of the point p. Now decompose the second component further. The intensity gathered from all directions over a hemisphere can be divided into three parts – first is the BRDF, second is light intensity in a direction $\vec{s}$, and the last part is a cosine term. In the cosine term, we take negative operator on the direction $\vec{s}$. This is because we set $\vec{s}$ as the incoming direction towards point p. To describe the inter-reflectance, we need to take the theory called Neumann Expansion into account in our rendering equation. It looks like this:

$$\mathrm{L}(\mathrm{p} \rightarrow \vec{d}) = L_0(p \rightarrow \vec{d}) + L_1(p \rightarrow \vec{d}) + \cdots$$

Based on the above equation, let's define each element.

$$L_0(p \to \vec{d}) = \int_\Omega f_r(p, \vec{s} \to \vec{d}) L_{env}(\vec{s}) V(p \to \vec{s})\, H_{N_p}(-\vec{s}) \mathrm{ds}$$

Can you find the difference between the previous rendering equation and this equation? In fact, we substitute the incoming intensity for the combination of incoming lighting over the environment map and a visibility function, a binary data set to determine whether a point is visible or not. The $L_0(p \to \vec{d})$ is direct lighting which came from light source – in this case, image-based lighting (Figure 20). How about $L_1(p \to \vec{d})$? Before knowing what the value of L_1 indicates, let's define the equation:

$$L_1(p \to \vec{d}) = \int_\Omega f_r(p, \vec{s} \to \vec{d}) L_0(p \leftarrow \vec{s})(1 - V(p \to \vec{s})) H_{N_p}(-\vec{s}) \mathrm{ds}$$

In the place of L_{env}, the previously calculated L_0 value is inserted, and the visibility function is replaced by the inverse visibility function. We need to determine whether $L_0(p \to \vec{s})$ is different from $L_0(p \to \vec{d})$. The $L_0(p \leftarrow \vec{s})$ describes the incoming intensity in a direction $\vec{s}$ toward the point p combined with the direct intensity of the intersected point from p in the direction $-\vec{s}$. The inverse visibility function accounts for one bounce – the bounced light is shadowed from the view of point p (Figure 20).

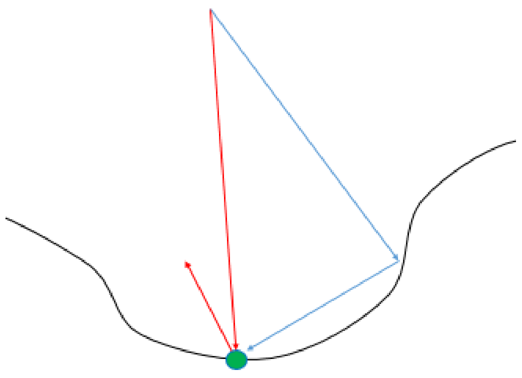


Figure 20. Path that takes 1 bounce (red) and path that takes 2 bounces (blue)

After comprehending this equation, we can generalize the i^{th} bounce model like this:

$$L_i(p \to \vec{d}) = \int_\Omega f_r(p, \vec{s} \to \vec{d}) L_{i-1}(p \leftarrow \vec{s})(1 - V(p \to \vec{s})) H_{N_p}(-\vec{s}) \mathrm{ds}$$

This model explains that the intensity of previous bounced light contributes outgoing radiance on an arbitrary point.

We can accomplish the rendering equation by ray-tracing or path-tracing. But these methods require lots of computation, which is not able to be done in real-time. The PRT enables calculation in real-time by precomputing incoming radiance with a transfer matrix, which will be covered later.

The PRT gives you fuller insight into Spherical Harmonics (SH). We know the SH is composed of a Legendre polynomial basis, in which basis functions are all orthogonal and orthonormal to each other. Other than that, the new basis function called the "dual basis function", should be introduced. As the word implies, the dual basis function is the inner product for two different basis functions, which are not orthogonal. But the dual basis function makes it possible to be orthogonal for those bases. Let's clarify this with sequential equations:

$$\langle B_i(t), B_j(t) \rangle = ?$$

For $B_i(t)$ and $B_j(t)$, which are non-orthogonal, what is the inner-product? We can't compute the projection directly. We first need to convert it into a suitable form that enables projection. The dual basis makes it possible for the projection:

$$\langle B_i(t), \widetilde{B_k}(t) \rangle = \begin{cases} 1, & i = k \\ 0, & else \end{cases}$$

In the above equation, $\widetilde{B_k}(t)$ is the dual basis; this differs the projection of $B_j(t)$ onto $B_i(t)$, as the orthogonality is preserved with $B_i(t)$. The dual basis is derived from two different basis functions. We can refer to the dual basis as the linear combination of basis functions, $B_i(t)$.

The dual basis is computed like:

$$A_{ij} = \int B_i(t) B_j(t)\, dt$$

$$\widetilde{B_k}(t) = \sum_j A_{kj}^{-1}\, B_j(t)$$

We will apply the dual basis functions to make a transfer matrix for glossy reflectance.

In basis functions, we have several options. The representative methods are wavelet and the SH. We have already learned SH, so from now on, we will regard SH as our basis function.

2.5.1.2 Diffuse PRT

The diffuse term has some key features – diffuse lighting is low-frequency and view-independent. Making use of these features, we can precompute the diffuse term using the Spherical Harmonics (SH).

Considering the direct light equation for $L_0(p)$, suppose all incoming light is coming from the environment map:

$$L_0(p) = \frac{P_d}{\pi}\int_{\Omega} L_{env}(\vec{s})V(p \to \vec{s})H_{N_p}(-\vec{s})d\vec{s}$$

The L_0 is diffuse lighting from $L_{env}(\vec{s})$ over the hemisphere. As we said earlier, the diffuse lighting is sampled at a low-frequency, which implies that we don't need accurate approximation. Therefore, we can approximate it with a few bands of the SH. This accelerates the calculation and reduces the memory bandwidth needed to store the SH coefficients. We define the SH coefficients for incoming light as $L_{env}(\vec{s})$:

$$L_{env}(\vec{s}) \approx \sum_{i}^{n} L_i Y_i(\vec{s})$$

The radiance from environment map is able to approximate the linear combination of the SH coefficients and the SH basis. If more accuracy is required, more coefficients and basis are needed, but we don't need that much precision.

Combining the two equations, we derive the final form, containing not only the direct diffuse term, but also shadow term.

$$L_0(p) = \frac{P_d}{\pi} L_i \int_{\Omega} Y_i(\vec{s})V(p \to \vec{s})H_{N_p}(-\vec{s})d\vec{s}$$

From here, we can define new terms called transfer coefficients, which will be stored in a transfer matrix. By precomputing the transfer matrix, dynamic lighting, as well as complicated light behaviors, can be computed in real-time.

$$T_{p,i}^0 = \frac{P_d}{\pi}\int_{\Omega} Y_i(\vec{s})V(p \to \vec{s})H_{N_p}(-\vec{s})d\vec{s}$$

$$L_0(p) = L_i T_{p,i}^0$$

With the transfer matrix ready to use, the only thing left is simple multiplication of the matrix and a vector (Sloan, et al., 2002). To add inter-reflectance, it is necessary to introduce the Neumann equation into the transfer matrix.

$$\mathrm{L}(\mathrm{p} \leftarrow \vec{d}) = L_0(p \to \vec{d}) + L_1(p \to \vec{d}) + \cdots$$

$$\mathrm{L}(\mathrm{p}) = \sum_{i}^{n} L_i (T_{p,i}^0 + T_{p,i}^1 + \cdots)$$

$$\mathrm{L}(\mathrm{p}) = \sum_{i}^{n} L_i \, T_{p,i}$$

With multiple transfer matrices, inter-reflection represents a linear combination of the transfer function and the SH coefficients of incoming light intensity. For dynamic lighting, the light coefficients are calculated like this every frame:

$$L_i = \int L_{env}(\vec{s})Y_i(\vec{s})d\vec{s}$$

How can we precompute the transfer coefficients? First, let's define the complete form with a complete cosine term for diffuse lighting.

$$T_{p,i}^0 = \frac{P_d}{\pi}\int_{\Omega} Y_i(\vec{s})V(p \rightarrow \vec{s})max(\overrightarrow{N_p} \cdot -\vec{s}, 0)d\vec{s}$$

By numerically evaluating the equation we can get:

$$T_{p,i}^0 = \frac{P_d}{\pi}\frac{4\pi}{N}\sum_{j=0}^{N-1} V(p \rightarrow \vec{s_j})\max(\overrightarrow{N_p} \cdot -\vec{s_j}, 0)Y_i(\vec{s})$$

Each $\vec{s_j}$ is uniformly distributed over the hemisphere. After generating a discrete set of directions, we can begin ray-tracing on each direction. How can we get discrete set of directions?

$$\vec{s_j} = (\theta, \varphi)$$

$$\theta = 2\cos^{-1}(\sqrt{1-x}), \varphi = 2y\pi$$

$$x, y \in [0, 1]$$

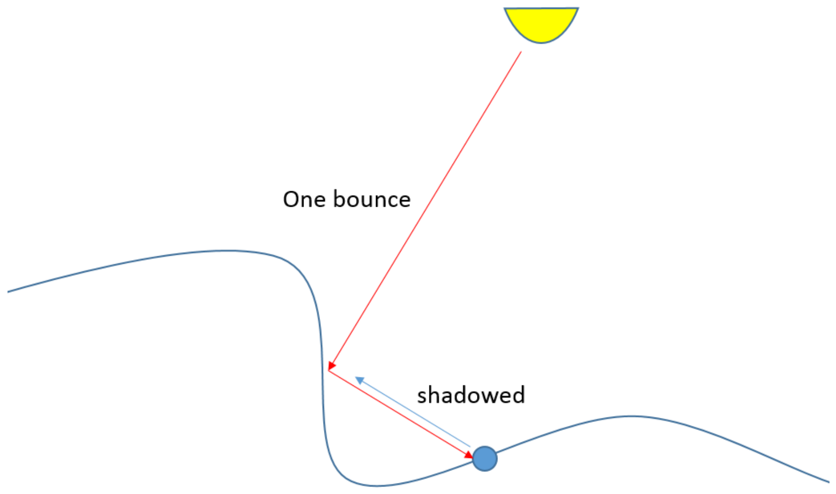


Figure 21. The point p receives one bounce radiance from the direction (red arrow) shadowed (green arrow)

The above explains how to compute each direction over the hemisphere in spherical coordinates from randomly generated x and y values ranging from 0 to 1.

To derive multiple bounce transfer matrices, first define one bounce transfer matrix, and then generalize the i^{th} bounce transfer matrix. One bounce diffuse radiance comes from the shadowed area of the point (Figure 21).

$$L_1(p) = \frac{P_d}{\pi}\int_{\Omega} L_0(p \leftarrow \vec{s})(1 - V(p \rightarrow \vec{s}))max(\overrightarrow{N_p} \cdot -\vec{s}, 0)d\vec{s}$$

To account for one bounce inter-reflectance on a shadowed area, the inverse visibility term is defined in the above equation. With generalized one-bounce incoming radiance, we can generalize the incoming radiance at the i^{th} bounce like this:

$$T_{p,i}^{b} = \frac{P_d}{\pi}\frac{4\pi}{N}\sum_{j=0}^{N-1} T_{p,i}^{b-1}(1 - V(p \rightarrow \vec{s_j}))\mathrm{max}(\overrightarrow{N_p} \cdot -\vec{s_j}, 0)$$

$$T_{p,i} = \sum_{b=0}^{B-1} T_{p,i}^{b}$$

The sum of transfer coefficients can describe any behavior of the incoming radiance, such as inter-reflectance, shadowing, and the diffuse term. It also makes it possible to have arbitrary and dynamic real-time lighting. However, it is only possible on static geometry and low-frequency lighting. How about high-frequency lighting like specular lighting? In the following section, we are going to deal with this in detail.

2.5.1.3 Glossy PRT

The glossy Precomputed Radiance Transfer (PRT) provides view-dependent, glossy reflection (or shiny BRDF). Like diffuse PRT, it has the same limitations, allowing only rigid scenes and objects. For rendering glossy surfaces in global illumination, transferred incident radiance is introduced. Transferred incident radiance is an input for calculating the final bounce radiance, used in computing the outgoing radiance towards the eye. The transferred incident radiance is calculated by the multiplication of the transfer matrix generated off-line with the incident light coefficients approximated by the SH. Additionally (not only for implementing glossy effect), we will adopt a method of neighborhood transfer – where lighting data is stored in free space —commonly called parameterized radiance volume. A similar technique is using irradiance volumes (Tatarchuk, 2005) (Greger, et al., 1998). By setting the irradiance approximation in a free space, the outgoing radiance of a glossy surface makes it possible to place moving objects with global illumination in a static scene. The final goal for the glossy PRT is to use the transfer matrix as a linear operator for incoming light sources.

Let's define the transferred incident radiance as $L_{xfer}(p \leftarrow \vec{w})$, describing all light arriving at p, coming from $L_{env}(\vec{w})$ (Figure 22).

The purpose of using the transferred incident radiance instead of calculating a final outgoing radiance is to get a more accurate outgoing radiance on an arbitrary point. The outgoing radiance is high-frequency radiance for the purpose of glossy reflection. Directly storing outgoing radiance in limited bands with SH loses lots of precision. However, by interpolating multiple less accurate incoming radiances, we can produce a high-frequency glossy outgoing radiance. When we are generating samples of incoming radiance, we don't know which exact points will be shaded. By sampling in a free space, the neighborhood transfer provides an approximation of the transferred incident radiances.

The overview for glossy PRT looks like this:

1. Precompute transfer matrix off-line
2. Use them at run-time to determine the transferred incident radiance for surfaces to be shaded
3. Integrate the incident radiance with BRDF and the cosine term to get outgoing radiance

The transferred incident radiance is approximated from environment lighting.

$$L_{xfer}(p \leftarrow \vec{w}) \approx \sum_{i=1}^{m} L_i^p Z_i(\vec{w})$$

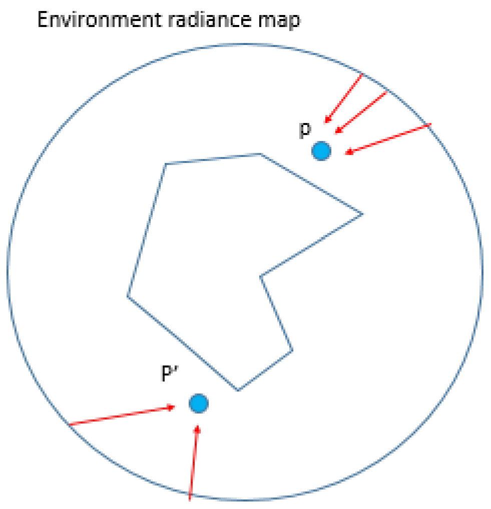


Figure 22. Multiple transferred incident radiance points p and p'

The above equation is the approximation that discretizes the transferred incident radiance. We can convert it to vector form like this:

$$L_{xfer}(p \leftarrow \vec{w}) \approx z(\vec{w})^T \cdot L^p$$

As you can see, the transferred incident radiance is approximated like the dot-product of two vectors – the basis functions and basis coefficients for incoming lights. For incoming lights, we can parameterize the vector-matrix multiplication.

$$L^p = T^p L$$

The T^p is the transfer matrix, which describes the behavior of light before it arrives at arbitrary point P. For instance, it describes direct lighting with shadowing and multiple bounces.

$$\mathrm{L} = \sum L_{env} Y_i(\vec{w})$$

We can also represents the incoming light in the SH basis. The projection of incoming radiance vectors onto the $Y_i(\vec{w})$ basis creates the SH coefficients. The resultant coefficients are able to be multiplied by the transfer matrix. The transfer matrix is composed of basis functions for each column. Each column looks like the basis created from i^{th} light basis functions. The details of entries into the transfer matrix are covered in the following section.

2.5.1.4 Entries of the Transfer Matrix

Each element in the transfer matrix is defined:

$$T_{ji}^p = \int_\Omega L_{xfer}^i(p \leftarrow \vec{w}) \tilde{z}_j(\vec{w}) d\vec{w}$$

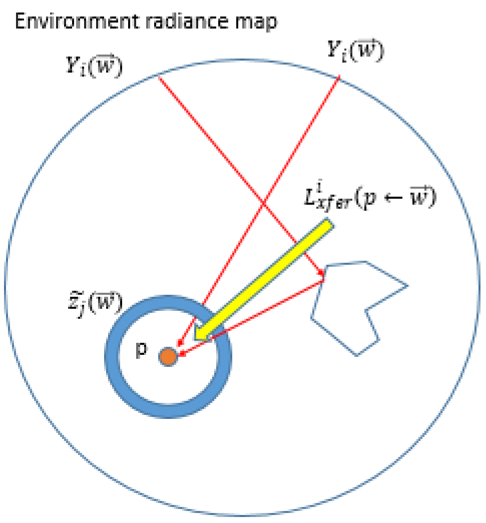


Figure 23. Description for each element in transfer matrix

There are a few terms to define in the equation. The $L^i_{xfer}(p \leftarrow \vec{w})$ is the light from the lighting basis $Y_i(\vec{w})$ that reaches the point p either directly or through any number of bounces off the object (Figure 23).

We see a new basis $\widetilde{z_j}(\vec{w})$. It is the dual basis that we covered in this section's introduction. To recap, the dual basis function enables orthogonality for two different basis functions, mapping one to the other. In here, the dual basis $\widetilde{z_j}(\vec{w})$ projects an arbitrary basis function onto $z_j(\vec{w})$. Combining the two terms, describes the projection coefficient of the transferred incident radiance on the dual basis. We can represent different entries in the transfer matrix as an inner-product.

$$T^p_{ji} = \langle L^i_{xfer}(p \leftarrow \vec{w}), \widetilde{z_j}(\vec{w}) \rangle$$

To adopt multiple bounce inter-reflection, there are various options for calculating the transferred incoming radiance, such as Monte-Carlo Path-Tracing, Photon Mapping, or Progressive Radiosity. We should be careful to avoid using negative lighting values in SH (Sloan, 2008).

It is still hard to understand how to generate entries for the transfer matrix. The transfer matrix for direct lighting will be a good starting point. Let's define the transfer matrix for direct lighting step by step.

$$L_0(p \leftarrow \vec{w}) = L_{env}(\vec{w}) V(p, \vec{w})$$

The equation for direct lighting on the point P in the direction of $\vec{w}$ looks like this. We don't need the full range of lighting. An approximated coefficient on the lighting basis $Y_i(\vec{w})$ is enough. Extracting lighting coefficients, the terms left looks like this:

$$\mathrm{L}^i_0(p \leftarrow \vec{w}) = Y_i(\vec{w}) V(p, \vec{w})$$

The inner-product of radiance from i^{th} lighting basis and j^{th} transferred incident radiance dual basis produces entry (i, j) of direct lighting transfer matrix:

$$T^p_{0,ji} = \langle \mathrm{L}^i_0(p \leftarrow \vec{w}), \widetilde{z_j}(\vec{w}) \rangle$$

$$T^p_{0,ji} = \int_{\Omega} Y_i(\vec{w}) V(p, \vec{w}) \widetilde{z_j}(\vec{w})\, d\vec{w}$$

The direct lighting transfer matrix represents the sum of all directions over the effective sphere, combining the direct lighting basis $Y_i(\vec{w})$, the incoming transfer radiance basis $\widetilde{z_j}(\vec{w})$ and the visibility function. The transfer matrix will transform the direct lighting coefficient to the transferred incident radiance on an arbitrary point P on the SH basis $z_j(\vec{w})$. Let's organize it:

$$\mathrm{L}(\mathrm{p} \leftarrow \vec{w}) \approx \sum_{j=1}^{m} L_j^p z_j(\vec{w}) = \sum_{j=1}^{m} z_j(\vec{w}) \left[\sum_{i=1}^{n} L_i T_{0,ji}^p \right]$$

With the SH coefficients of incoming lights L_i and direct lighting transfer matrix $T_{0,ji}^p$, we can approximate the transferred incident radiance by the simple linear operations of matrix and vector multiplication.

So far we have looked through the direct lighting transfer matrix. But for glossy surfaces, we need to account for multiple bounces. As we discussed earlier, there are several ways to approximate multiple bounces for the transferred incident radiance, $L_{xfer}^i(p \leftarrow \vec{w})$ like photon mapping. But in this thesis, we are going to use progressive method, using the incorporated transfer matrix for multiple bounces.

The main reason for using the progressive method with the transfer matrix is to approximate the heavy calculations of global illumination as linear operators through vector-matrix multiplication. First we define the complete form of the rendering equation for transferred incident radiance (Figure 24).

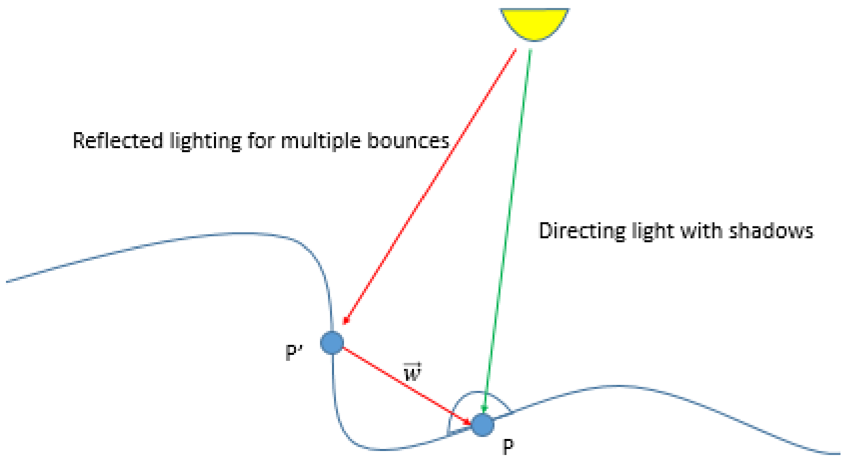

Reflected lighting for multiple bounces
Directing light with shadows
P'
$\vec{w}$
P

Figure 24. Two components of progressive method with transfer matrix

$$L_{xfer}(p \leftarrow \vec{w}) = L_{env}(\vec{w}) V(p, \vec{w}) + \int_{\Omega(p')} L_{xfer}(p' \leftarrow \vec{w}') f_r(p', \vec{w}' \rightarrow -\vec{w}) \cos\theta' \, d\vec{w}'$$

We divide this equation into two parts – direct lighting with shadows and reflected lighting for multiple bounces. Let's look at an example that should prove useful in illustrating the process.

The first part is straightforward to understand. We already have covered this. The problem is the second part of the equation, which represents indirect radiance from the point P' shadowing in the direction $\vec{w}$ toward the point P (Figure 24).

It consists of the direct lighting of P', a BRDF from $\vec{w}'$ to $-\vec{w}$ describing glossy reflectance, and lastly a cosine term. It is integrated over the hemisphere on the point P'. To sum up, the equation describes indirect lighting from P' toward P. For multiple bounces, we can convert it into terms of the Neumann Series like this:

$$L_{k+1}(p \leftarrow \vec{w}) = \int_{\Omega(p')} L_k(p' \leftarrow \vec{w}) f_r(p', \vec{w}' \rightarrow \vec{w}) \cos\theta' \, d\vec{w}'$$

$$\mathrm{L}(\mathrm{p} \leftarrow \vec{w}) = L_0(p \leftarrow \vec{w}) + L_1(p \leftarrow \vec{w}) + \cdots$$

To derive the transfer matrices for all bounces, we need to project these radiance values into the dual basis of the point P, $\widetilde{z_j}(\vec{w})$. Before getting into this, we should recap the definition of the transferred incident radiance.

$$L_{xfer}(p \leftarrow \vec{w}) \approx \sum_{i=1}^{m} L_i^p Z_i(\vec{w})$$

$$L_j^p = \int_{\Omega} L_{xfer}(p \leftarrow \vec{w}) \, \widetilde{z_j}(\vec{w}) d\vec{w} := \langle L_{xfer}(p \leftarrow \vec{w}), \widetilde{z_j}(\vec{w}) \rangle$$

Now let's project multiple bounces of indirect lighting of the transferred incident radiance to the dual basis $\widetilde{z_j}(\vec{w})$.

$$L_j^p = \langle L_0(p \leftarrow \vec{w}), \widetilde{z_j}(\vec{w}) \rangle + \langle L_1(p \leftarrow \vec{w}), \widetilde{z_j}(\vec{w}) \rangle + \cdots$$

After generalizing the equation:

$$L_{k+1,j}^p = \langle L_{k+1}(p), \widetilde{z_j}(\vec{w}) \rangle = \langle \int_{\Omega(p')} L_k(p' \leftarrow \vec{w}) f_r(p', \vec{w}' \rightarrow \vec{w}) \cos\theta' \, d\vec{w}', \widetilde{z_j}(\vec{w}) \rangle$$

The k^{th} bounces incoming radiance based on $\widetilde{z_j}(\vec{w})$ is calculated with the previously bounced incident radiance $L_k(p' \leftarrow \vec{w})$. The problem is how to get the radiance after the previous bounce, since we don't inherently know all these radiances.

At the beginning in this section, we said that we need to do neighborhood transfer by setting sparse points to store the transferred incident radiance. With these data, we can interpolated the radiance after the k^{th} bounce like this:

$$L_k(p' \leftarrow \vec{w}') = \sum_{s=0}^{S} w_s L_k(P_s \leftarrow \vec{w}')$$

To get k^{th} transferred incident radiance, we interpolated several samples near the point P' with an appropriate weighting value. Now we have the incident radiance from the previous bounce. Putting it all together:

$$L_{k+1,j}^{p} = \langle L_{k+1}(p), \widetilde{z_J}(\vec{w}) \rangle$$

$$L_{k+1,j}^{p} = \langle \int_{\Omega(p')} \left(\sum_{s=0}^{S} w_s L_k(P_s \leftarrow \vec{w}') \right) f_r(p', \vec{w}' \rightarrow \vec{w}) \cos\theta' \, d\vec{w}', \widetilde{z_J}(\vec{w}) \rangle$$

For all samples to interpolate the previous radiance, we can approximate the SH coefficients with new basis functions:

$$L_k(P_s \leftarrow \vec{w}') \approx \sum_{a=0}^{A} L_{k,a}^{P_s} Z_a(\vec{w}')$$

For calculation convenience, we should represent the linear operator as matrix-vector multiplication:

$$L_{k,a}^{P_s} = \sum_{i=0}^{I} L_i T_{k,ai}^{P_s}$$

Finally, we can get this:

$$L_{k+1,j}^{p} = \langle \int_{\Omega(p')} \sum_{s=0}^{S} w_s \left(\sum_{a=0}^{A} \left(\sum_{i=0}^{I} L_i T_{k,ai}^{P_s} \right) z_a(\vec{w}) \right) f_r(p', \vec{w}' \rightarrow \vec{w}) \cos\theta' \, d\vec{w}', \widetilde{z_J}(\vec{w}) \rangle$$

Rearrange the equation:

$$L_{k+1,j}^{p} = \sum_{i=0}^{I} L_i \langle \int_{\Omega(p')} \sum_{a=0}^{A} \left(\sum_{s=0}^{S} w_s T_{k,ai}^{P_s} \right) z_a(\vec{w}) \, f_r(p', \vec{w}' \rightarrow \vec{w}) \cos\theta' \, d\vec{w}', \widetilde{z_J}(\vec{w}) \rangle$$

From this equation, we can derive the transfer matrix:

$$T_{k+1}^{p} = \langle \int_{\Omega(p')} \sum_{a=0}^{A} \left(T_{k,ji}^{p'} \right) z_a(\vec{w}) \, f_r(p', \vec{w}' \rightarrow \vec{w}) \cos\theta' \, d\vec{w}', \widetilde{z_J}(\vec{w}) \rangle$$

$$\text{where } T_{k,ji}^{p'} = \sum_{s=0}^{S} w_s T_{k,ai}^{P_s}$$

The equation

$$\sum_{a=0}^{A} \left(T_{k,ji}^{p'} \right) z_a(\vec{w})$$

is the transferred incident basis function $Y_i(\vec{w})$ captured at P_1 and P_2 by basis function $Z_a(\vec{w})$ interpolated to P'.

The complete transfer matrix is derived from the multiplication of matrices that represents each bounce of the transferred incident radiance. The compound transfer matrix is:

$$T^p = T_0^p + T_1^p + \cdots$$

We have derived the transfer matrix describing direct lighting with shadows and multiple bounce inter-reflectance with BRDF. We can also apply this to any mechanism related to global illumination using a transfer matrix, such as caustics.

For now, we can get the transferred incident radiance containing multiple interactions on various materials from the light SH coefficients based on the SH. With the incident radiance, we go further to define outgoing radiance. To complete the outgoing radiance, the BRDF term is considered:

$$L_{out}(p \to \vec{w}_{out}) = \int_{\Omega} L_{xfer}(p \leftarrow \vec{w}_{in}) f_r(p, \vec{w}_{in} \to \vec{w}_{out}) \cos\theta' \, d\vec{w}_{in}$$

The BRDF is constant for all surface points when calculating the outgoing radiance. So it is convenient to transform the global frame to the tangent frame to simplify the calculation of the BRDF term. We can achieve this by rotating into the tangent frame. To account for various effects, we generate a transfer matrix. For rotation, the rotation matrix is helpful as linear operator. But the incoming light source is parameterized as the SH coefficients. For these coefficients, it is necessary to get specialized rotation matrix called the SH rotation matrix. As we covered in the SH chapter, the SH rotation is mostly sparse. To make use of this, we can reduce the complexity of the matrix calculation. Using the rotation matrix, we can represents the outgoing radiance in matrix-vector multiplication form:

$$L^{R,p} = R^p L^p = R^p T^p L$$

The transferred incident radiance is in the SH coefficients, so we need to restore it by projecting the SH basis $z(\vec{w})$:

$$L_{xfer}^{R}(p \leftarrow \vec{w}) = \sum_{j=0}^{m} L_j^{R,p} z(\vec{w})$$

In the linear operator form:

$$L_{xfer}^{R}(p \leftarrow \vec{w}) = z(\vec{w})^T R^p T^p L$$

The rotation into tangent frame is implicit in the compound transfer matrix. The transferred incident radiance is calculated by a simple linear operator. With arbitrary radiance, we are able to get the final outgoing radiance. Before doing that, notice that the BRDF term and cosine term can be turned into another transfer matrix. But we need to know that the frequency for usage of this matrix is greater than that of the transferred matrix for incident radiance. The matrix for transferred incident radiance statically resides in the free space, and when calculating the radiance for each surface point, it is constant. Whereas the matrix for outgoing radiance is changed frequently depending on the normal because of rotating to the tangent frame for the BRDF term. So we handle them separately.

First let's apply the transferred incident radiance to the rendering equation for the outgoing radiance:

$$L_{out}(p \to \vec{w}_{out}) = \int_{\Omega} \left[\sum_{j=1}^{m} L_j^{R,p} Z_j(\vec{w}) \right] f_r(p, \vec{w}_{in} \to \vec{w}_{out}) \cos\theta' \, d\vec{w}_{in}$$

$$L_{out}(p \to \vec{w}_{out}) = \sum_{j=1}^{m} L_j^{R,p} \int_{\Omega} Z_j(\vec{w}) f_r(p, \vec{w}_{in} \to \vec{w}_{out}) \cos\theta' \, d\vec{w}_{in}$$

Except for the lighting source coefficient $L_j^{R,p}$, the rest of terms are view-dependent, and can be presented in a new basis function u:

$$L_{out}(p \to \vec{w}_{out}) = \mathrm{u}(\vec{w}_{out})(C^p B^p) R^p T^p L$$

In the equation, new matrices is appeared. The first C^p is compound BRDF matrix and the other B^p is basis change matrix. For basis change matrix, further description is necessary. We additionally parameterized new basis functions on the outgoing radiance. A while ago, for the BRDF term, we added a rotation matrix. It is essential to restore the frame before projecting the outgoing basis function. This is what the basis change matrix is for.

We said that we need to manage two different transfer matrices for outgoing and incident radiances. Each matrix looks like this:

$$T_{outgoing} = C^p B^p R^p$$

$$T_{incident} = T^p$$

The glossy PRT is more complicated than the diffuse PRT. But after precomputation of the transfer matrix in free space, we can approximate high quality surfaces in global illumination. To wrap up this section, let's summarize the steps for glossy PRT.

1. Parameterize the distant light source onto SH basis function
2. Precompute the transfer matrix T^p for the transferred incident light
3. Generate the outgoing radiance transferred matrix
   - The rotation matrix to tangent frame

- The compound BRDF matrix
- The basis change matrix

4. Apply the outgoing matrix and the incident matrix to the distant light coefficients

2.5.1.5 Compression

In preceding sections, we have covered diffuse PRT and glossy PRT. In these PRTs, we had stored the precomputed transfer matrices. The storage of these matrices causes issues with memory space and performance. To approximate effectively, we should use at least a 25x25 matrix for glossy surfaces. As we said earlier, as opposed to diffuse lighting, specular lighting requires more accurate precision for its natural, high-frequency lighting. If we were to add spectra RGB for color bleeding, it would require 3x more memory space. For real-time global illumination, this much memory space is a poor trade-off. In performance, glossy PRT is also not GPU-friendly, which results in slow calculation. We should optimize the algorithm enough to warrant using the PRT in real-time rendering.

To reduce memory consumption, compression is the best option to choose. For best performance, we need to set additional constraints like static lighting sources and a static view direction. With compression, we should render the compressed representation directly for effectively encoding and decoding. The compression gives us not only a reduction on disk space but also an improvement on run-time memory access efficiency, which improves cache coherency.

Vector Quantization (VQ) is a basic option to consider for compression. Before applying the VQ, we should know the VQ in general terms. First, see the picture below (Figure 25).

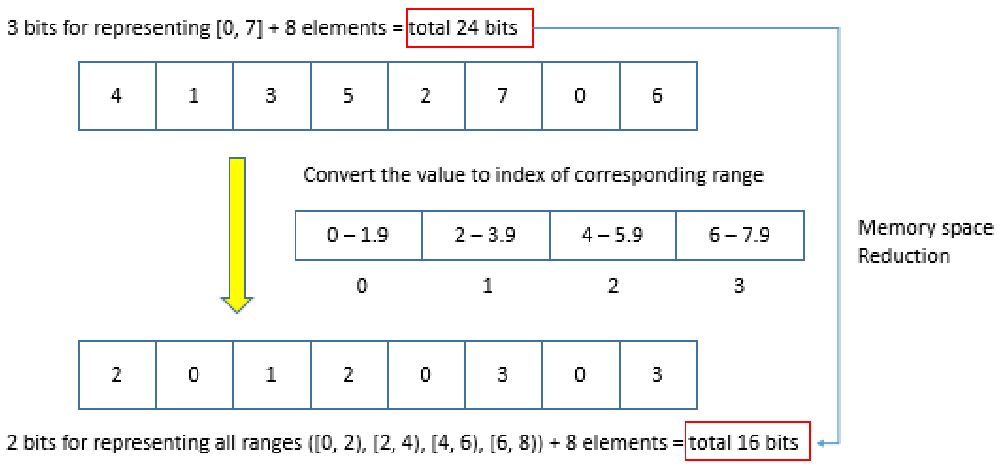

3 bits for representing [0, 7] + 8 elements = total 24 bits
4 1 3 5 2 7 0 6
Convert the value to index of corresponding range
0 – 1.9
2 – 3.9
4 – 5.9
6 – 7.9
0 1 2 3
Memory space Reduction
2 0 1 2 0 3 0 3
2 bits for representing all ranges ([0, 2), [2, 4), [4, 6), [6, 8)) + 8 elements = total 16 bits

Figure 25. Vector Quantization description

Suppose we have 8 values ranging from 0 to 7. Each value needs 3 bits to represent the values ranging from 0 to 7. A total memory space of 24 bits is required for storing all the data. To compress this, we can use the VQ. First we divide the range up into sections of equal size. In the figure, we have split up the range into pairs. Next, substitute each value with the mean value of belonged range, looping each value in a set. At the end, we get a compressed set, preserving its precision as much as possible.

In vector space, the basic concept is not different. For the PRT, we can compress outgoing radiances by the VQ. The samples over the surface are grouped into a small number of samples, and the samples are replaced with the cluster's mean. The one thing to care about is that all data should be in signal space before clustering; in the PRT, the signal describes the surface normal (Figure 26).

$$M_p \approx \widetilde{M}_{C_p} = M_{C_p}$$

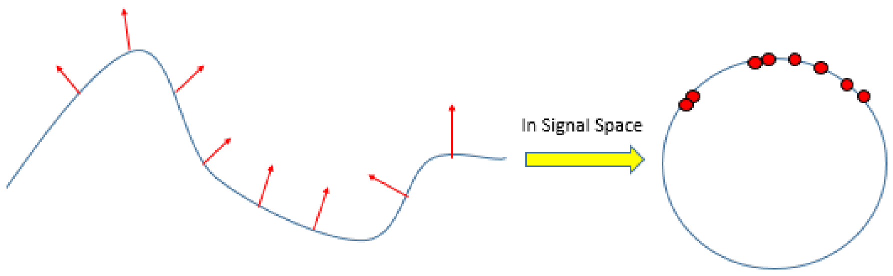


Figure 26. Surface normal in signal space

Applying the VQ only causes noticeable artifacts. For smooth transitions on outgoing radiances, we need another compression technique called Principle Component Analysis (PCA) (Sloan, et al., 2003). The PCA computes an optimal lower−dimensional linear approximation to the signal to at least mean squared error.

$$M_p \approx \widetilde{M}_p = M^0 + \sum_{i=1}^{N} w_p^i M^i$$

Each outgoing radiance on the surface is approximated to the addition of clustered mean values M^0 and linear combination of a set of weights w_p^i and PCA basis M^i. Using the PCA, we reduce the memory space; the matrix per point on the surface is bigger than the weight coefficients and the constant PCA mean and PCA basis. But the PCA coefficients, weights w_p^i are a 1D manifold, which is not adequate for approximation in the global frame. It could lose radiance energy in the outgoing radiance, but it is well-approximated locally. For the PCA compression locally, we introduce CPCA (Clustered Principle Component Analysis). The CPCA computes a subspace in each local frame. The data to be stored is composed of a cluster index and a set of projection weights for the PCA basis (Figure 27).

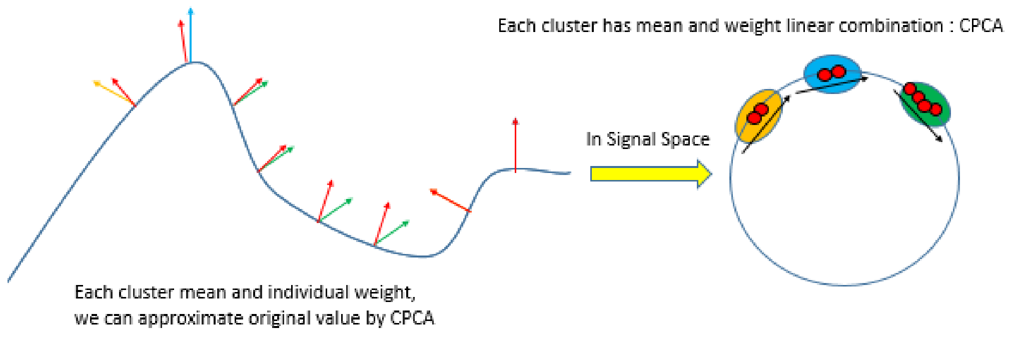

Each cluster has mean and weight linear combination : CPCA
In Signal Space
Each cluster mean and individual weight,
we can approximate original value by CPCA

Figure 27. CPCA description

The CPCA is seen as hybrid of VQ and PCA – clustering with VQ and smoothing with PCA locally. For each frame, the CPCA looks like this:

$$M_p \approx \widetilde{M}_{C_p} = M_{C_p}^0 + \sum_{i=1}^{N} w_p^i M_{C_p}^i$$

Plug the CPCA into the rendering equation of outgoing radiance:

$$e_p(v_p) = u(v_p)^T M_p L$$

$$e_p(v_p) = u(v_p)^T \left(M_{C_p}^0 + \sum_{i=1}^{N} w_p^i M_{C_p}^i \right) L$$

The equation can be optimized further by setting restrictions on the scene. First, we will make the assumption that the light is static – there are no changes to its intensity and position:

$$e_p(v_p) = u(v_p)^T \left(e_{C_p}^0 + \sum_{i=1}^{N} w_p^i (e_{C_p}^i) \right)$$

The cluster mean and PCA basis becomes constant by projecting lighting coefficients on both terms. By confining the view direction, we are able to increase the performance:

$$e_p = \mathrm{A} + \sum_{i=0}^{N} w_p^i B$$

$$\mathrm{A} = u(v_g)^T e_{C_p}^0 \quad \mathrm{B} = (v_g)^T e_{C_p}^i$$

The values of A and B are precomputed. As a result, the outgoing radiance is produced by the simple linear combination of colors, implying the independence of the view direction and lights' order. By placing these restrictions on the view direction and lights, we can get better performance, as well as memory space reduction.

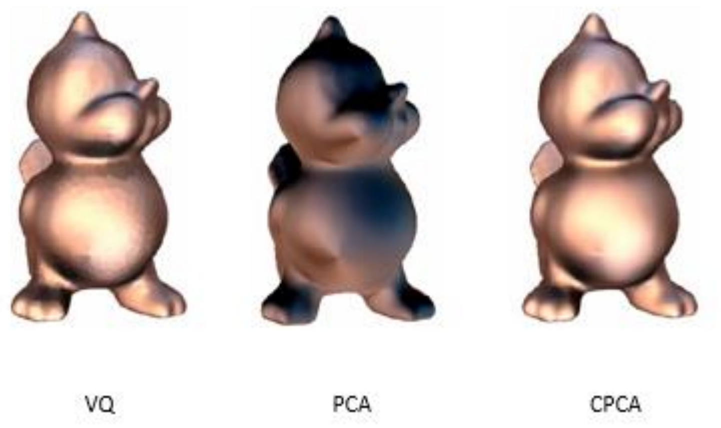


Figure 28. Comparison VQ, PCA and CPCA (Sloan, et al., 2003)

2.5.1.6 Conclusion

We went through all the materials required in implementing the PRT for diffuse and glossy surfaces. The PRT is a big contributor to understanding how Spherical Harmonics (SH) is used mathematically or graphically. The current trend in global illumination used in games is steadily moving towards voxelized global illumination methods, such as voxel cone tracing or light propagation volume. The major reason for this is that the preprocessing phase takes too much time. Taking too much time on precomputation results in inefficient cycles between artists and programmers. So the voxelized global illumination algorithms which do the whole process in real-time are gaining more interest than other methods. But considering the quality produced by Global Illumination, these algorithms can't catch up with the PRT. The PRT is still widely used to enhance the graphical quality in games. As a technique heavily based on SH, it is worthwhile to cover the PRT in detail.

2.5.2 Light Propagation Volumes

2.5.2.1 Introduction

As a global illumination algorithm designed by Crytek, Light Propagation Volumes (LPV) has been successfully implemented into CryEngine, their custom game engine (Kaplanyan, 2009) (Kaplanyan & Dachsbacher, 2010). LPV computes the intensity of each cell in the lattice. The lighting is sampled across this lattice capturing the Virtual Point Lights (VPLs) in screen-space usually stored in textures. The main advantage here is that we can sufficiently compute it from scratch for every frame. The light propagation algorithm enables the indirect lighting of full dynamic scenes. But it can suffer from ray effects by the discretization of the direction and smearing lights by repeating averages. Because of this, it is hard to apply this algorithm to high-frequency lighting like specular lighting. However, in low-frequency lighting these errors are tolerable. The advent of LPV affects all aspects of the production on games. The LPV makes it possible to compute indirect diffuse lighting in real-time on fully dynamic scenes. Unlike with the PRT, artists do not need to wait for their works to produce global illumination; artists can get direct feedbacks from their art assets instantly. This accelerates the whole process of the development of games. To produce Global Illumination (GI), the diffuse term requires the heaviest computation, since it has to trace lots of rays to get a realistic result. On the other hand, the specular term should be more accurate than the diffuse term because of its natural, high-frequency lighting. It can be handled by so-called screen space techniques like Screen Space Reflection (SSR), which doesn't impose the same performance cost on the calculation.

2.5.2.2 Overview

In the LPV, two voxel grid structures are needed – one for indirect light intensity and the other for potential occlusion, which is approximated by volumetric fuzzy occlusion. All data in a lattice is stored in the form of Spherical Harmonics (SH) coefficients. As a voxel based data structure, using SH coefficients is the best option, as previously noted.

The steps can be overviewed like this:

1. Initialize the voxel grids for the LPV and render the Reflective Shadow Maps (RSMs)
2. Inject Virtual Point Lights (VPLs) from RSMs into the LPV
3. Propagate the indirect light intensity in voxels, accumulating intermediate results
4. Render the scene with the LPV

We will separately cover each step in detail.

2.5.2.3 LPV Initialization

Before propagating the indirect light intensity, we need access to the indirect illumination stored in a volumetric structure. We could get this by rendering Reflective Shadow Maps (RSMs) (Keller, 1997). Each pixel in RSMs represents an indirect light source. We refer to these surfels as Virtual Point Lights (VPLs). We can represent each surfel like this:

$$I_p(w) = \Phi_p \langle n_p | w \rangle_+, where\ \langle \cdot \mid \cdot \rangle_+ : dot\ product\ clamp\ to\ 0\ if\ it\ is\ negative$$

For each generated VPL, the intensity is the flux cosine term of the surfel's normal and the incoming light direction. To approximate indirect illumination with directional information, we accumulate indirect diffuse lighting with Spherical Harmonics (SH) coefficients. This is necessary to optimize the algorithm in terms of performance and memory space. For each VPL, we project its intensity onto the SH basis.

$$C_{l,m} = \sum_{i=0}^{4} I_p(w) Y_i(w)$$

When we accumulate the SH lighting into the voxel, it can suffer from self-lighting or self-shadowing due to its discretization. To avoid this side-effect, it is recommended to move each VPL from RSM by half of the cell spacing along its normal when we are injecting the VPLs. We already

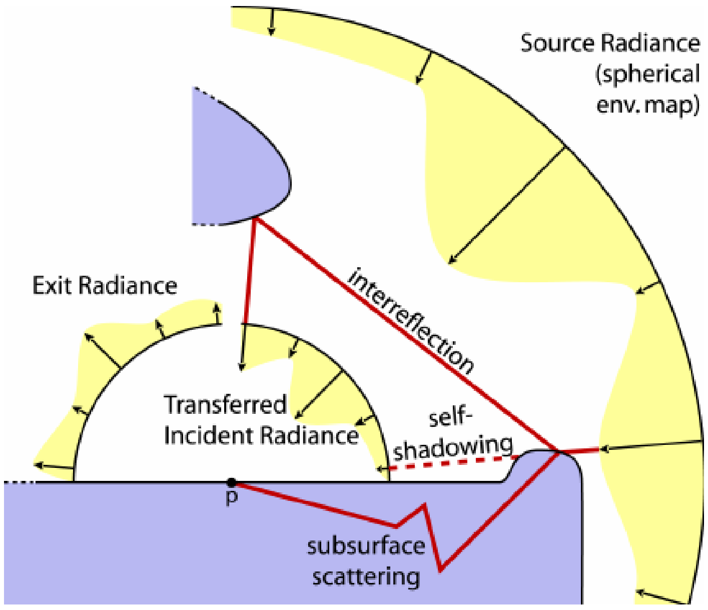


Figure 29. Various light sources interactions (Sloan, et al., 2002)

know a VPL's orientation by its normal. Zonal harmonics can be used as an alternative to the cosine term (Sloan, 2008). With zonal harmonics, the cosine term is constant. The only thing to do is to rotate ZH in accordance to its tangent space. After determining the grid cell from the depth value in the RSM, accumulate the calculated intensity of the SH approximations.

We can accumulate values − not only form indirect light sources, but also direct light sources − into the lattice. For example, we can accumulate illumination from area lights, environment maps, point lights or lighting emitted from a particle system (Figure 29).

Any type of light source is accumulated after projecting its intensity on the SH basis. For preventing self-shading, we should also snap to half the grid size.

2.5.2.4 Scene Geometry Injection

When propagating light from one grid to another, improve image quality by taking into account geometry blocking, which follows from indirect shadows. To get close to the quality of the PRT without precomputation, the Geometry Volume (GV) is essential for dynamic scenes. We use the G-Buffer and RSM to sample a rendered scene's surface, which increases performance by doing calculation in screen-space. In the case of a more complex scene, we should use multiple RSMs and G-Buffers sampled by depth-peeling.

How can we accumulate the occlusion into a cell? In LPV, an algorithm called Fuzzy Occlusion is used to accumulate the blocking potential of surfels in a grid cell, which is represented as the probability of blocking light from a certain direction going through that cell:

$$\mathrm{B}(\vec{w}) = \frac{A_s\langle n_s|\vec{w}\rangle}{S^2}$$

The parameters in the above equation need some additional description. The A_s is the area of single surfel s. The n_s is the normal of the surfel. Lastly, the upper case S is the grid size of the LPV. The blocking potential of a surfel is calculated as the relation between the grid size and surfel size.

We accumulate the SH projections of blocking potential into GV at the same resolution as the lattices storing light intensity. The Spherical Harmonics (SH) allows the geometry potential to be stored in directional properties in limited memory channels. However, sampling with the same resolution causes discretization during the propagation. To reduce the discretization, we shift by half a cell and accumulate the blocking potential to the GV. Consequently, this produces a smooth transition between each blocking potential and it reduces the discretization.

When we accumulate the occlusion from multiple G-buffers and RSMs, we should pile up the surfels from different render targets separately. First, generate several GVs and accumulate surfels from different buffers to separate GVs. After that, merge the multiple GVs into one using a maximum operator on the SH coefficients. For further optimization, we can separately store the GV for static objects in advance. Hence, the only initialization for GVs that is required is for dynamic objects, which increases performance.

2.5.2.5 Propagation Schemes

The light propagation through the scene is computed using successive local iterative steps. The initial light propagation volume has indirect lighting sources. These light sources need to be propagated to approximate one bounce of light on the scenes. We maintain two type of volumes – one for the intensity and the other for the occlusion. The intensity volume is the one that is propagated along the axial directions in 6 ways (Figure 30).

The intensity in the initialized LPV is a SH approximation. We can restore the intensity from SH coefficients:

$$\mathrm{I}(\vec{w}) \approx \sum_{l,m} C_{l,m} Y_{l,m}(\vec{w})$$

To propagate indirect light, we compute the flux on each of the faces of the adjacent destination cell. To compute the total flux reaching a given face:

$$\Phi_f = \int_{\Omega} I(\vec{w}) V(\vec{w}) d\vec{w}$$

For total flux, we consider the visibility function over the face from the intensity source. Similar to the SH approximation of the intensity, we can approximate the visibility function on the SH basis:

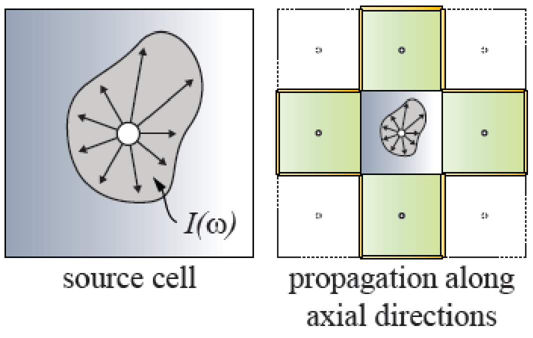


Figure 30. The light propagation in 2D (Kaplanyan & Dachsbacher, 2010)

$$V_{l,m} = \sum_{i=1}^{4} V(\vec{w})Y_i(\vec{w})$$

With SH approximations of the intensity and visibility, we can shrink the calculation for total flux to a single dot-product:

$$\Phi_f = \int_{\Omega} I(\vec{w})V(\vec{w})d\vec{w} = C_{l,m} \cdot V_{l,m}$$

However, approximating the visibility function on the SH basis can be very inaccurate for low-order SH approximation. To overcome this, we can use high-order SH approximation, but this costs lots of memory space and performance degradation. Keeping in mind the LPV, I suggest a different approach to this problem.

The different approach is to represent visibility for each face to the destination cell by a solid angle:

$$\Delta w_f = \int_{\Omega} V(\vec{w})\, d\vec{w}$$

We determine the central direction w_c of the visibility cone. With the calculated visibility cone, we calculate the flux reaching to the face (Figure 31):

$$\Phi_f = \frac{\Delta w_f}{4\pi} I(w_c)$$

After calculating the flux for all faces in a destination cell, we reproject each face's flux to the center of the grid. To reproject the flux, we reflect across the outgoing direction. For this, we reproject the flux on the direction vector from the center to the face (Figure 31):

$$\Phi_f = \int_{\Omega} \Phi_l \langle n_l | \vec{w} \rangle_+ d\vec{w}$$

$$\Phi_l = \frac{\Phi_f}{\pi}$$

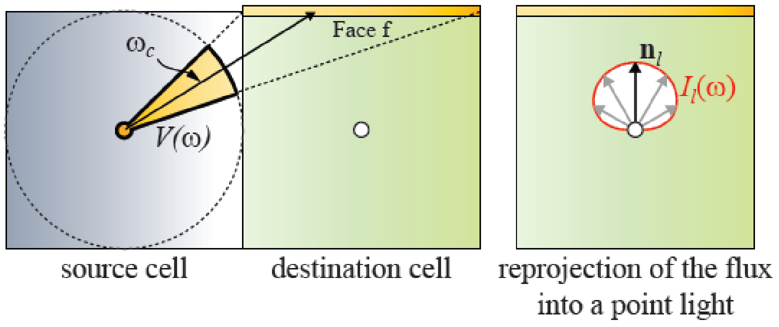


Figure 31. Compute the flux onto the faces the destination cell (Kaplanyan & Dachsbacher, 2010)

To model light propagation, we can make use of the GV that we created. The GV is shifted half a grid square relative to the intensity LPV. When we propagate the LPV, we will interpolate tri-linearly a blocking potential and attenuate the intensity according to GV's coefficients. To prevent overlapped occlusion, we do not consider the first propagation.

The determination for the number of iterations in light propagation depends on the resolution of light propagation volumes. At higher resolutions, the process requires more iteration, while for lower resolution, it takes fewer less iterations.

2.5.2.6 Indirect illumination with the LPV

We render the scene with the lattices for Light Propagation Volume (LPV) and Geometry Volume (GV). The SH coefficient can be interpolated linearly. Using trilinearly interpolated SH coefficients from LPV, we evaluate the intensity for the negative orientation of the surface. To prevent self-shadowing or self-illumination, we suppose that the distance between a cell's center and the surface to be lit is half of the grid size. But the LPV still suffers from same problem, and is susceptible to light-bleeding. A damping factor based on the directional derivative of the intensity distribution can greatly save us from these issues. Before choosing a damping factor, we compute the directional derivative for SH coefficients and determine whether the blocker exists. Suppose the surface location x, normal n, tri-linearly interpolated SH coefficients $C_{l,m}$ and the directional derivative$\nabla_n C_{l,m}$. The directional derivative can be calculated as the difference of values between opposite directions (Figure 32).

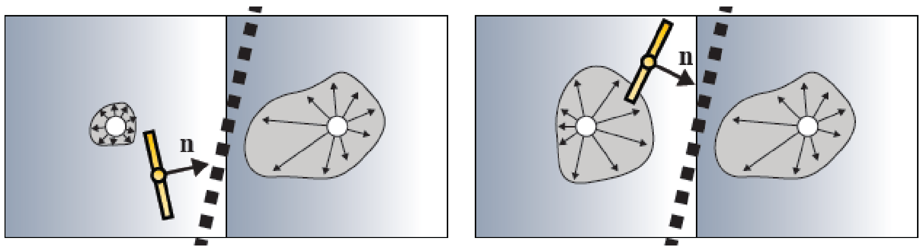


Figure 32. Detect the discontinuity in the intensity by computing its gradient in normal direction n (Kaplanyan & Dachsbacher, 2010)

With the derivative, we can determine the existence of blockers between two cells. If it is large, then the SH coefficient should be deviating – the blocker exists. The proper damping value is multiplied with the SH coefficients. By doing this, we can attenuate the intensity by means of fuzzy occlusion from the Geometry Volume (GV).

2.5.2.7 Cascaded Light Propagation Volumes

In the previous chapter, we covered the Cascaded Shadow Maps. The Cascaded Light Propagation Volumes algorithm shares similar supporting principles – it provides high spatial resolution in parts of the scene close to camera, while covering distant parts with a lower resolution. Each proper resolution for LPV is 32x32x32 (Figure 33).

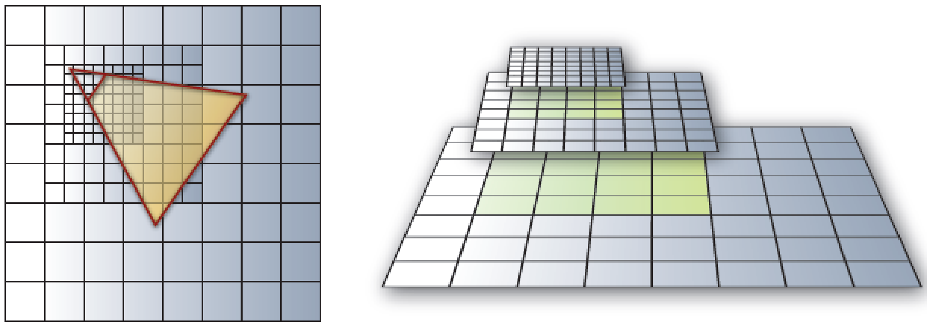


Figure 33. Cascaded Light Propagation Volumes (Kaplanyan & Dachsbacher, 2010)

The propagation scheme is same under different cascaded volumes. We propagate lights individually in a process called 'Nested Gird Propagation'. At all boundaries of a fine grid, we create a smooth transition to the next coarser grid by interpolation.

Cascaded LPV mainly causes two side-effects: spatial discretization which stems from single LPV, and inconsistency or flickering. To reduce spatial discretization, we can apply a similar solution on artifacts in Cascaded Shadow Maps. We make sure our shadow map covers the entire sphere, snapping one cell grid to multiples of the grid size. Most of time, flickering is caused by a lack of samples. Therefore, an increase in the number of sampling surfaces and the injection of a huge number of VPLs can relieve this problem.

2.5.2.8 Implementation

To reduce the memory space used, we use 2 SH bands per color channel. The 2 SH bands are enough for diffuse indirect illumination. Using more than 2 SH bands is wasteful in performance and memory space. Conveniently, 2 SH bands also perfectly fit in texture channels. It requires 4 coefficients, exactly matching the number of channels in RGBA textures. If we use a 32 bit RGBA texture format, it significantly degrades quality. We already use limited SH bands, so to prevent further loss in precision, we need to use a 16 bit floating point texture to store SH coefficients.

For Cascaded Light Propagation Volumes, the three cascaded grids are enough to cover the scene. The parts of the scene not covered by these grids can be rendered via screen-space global illumination (SSGI) (Ritschel, et al., 2009), which doesn't need to be accurate, because the remaining objects are far from the camera. We also need cascaded grids for geometry volumes. With this method, GV provides more precise blocking potential. We said that the number of iterations for light propagation depends on the resolution of the grids used. For 32^3grids for LPV, 8 iterations is sufficient in most scenes.

2.5.2.9 Conclusion

LPV empowers the rendering engine to do the whole process of global illumination from scratch in real-time. Even though it only controls low-frequency lighting, it lightens the heaviest part of Global illumination. At the end, it speeds up production speed for artists by providing them with immediate feedback. Nowadays, most well-known rendering engines like CryEngine, Unreal Engine, and many others adopt LPV for diffuse indirect illumination. A new era of consoles came out, but the processors inside haven't improved a lot, especially compared to the advances made in the previous generation. For this generation, LPV techniques will continue to influence the future of rendering with global illumination.

2.5.3 Voxel Cone Tracing

Voxel Cone Tracing (VCT) (Crassin, et al., 2011) is the algorithm that enables us to approximate real-time global illumination from a voxelized scene. This voxelization allows the calculation for indirect illumination to be independent of the geometric complexity of the scene. Once the scene is voxelized, the complexity of scene doesn't affect performance, which contributes to the feasibility of real-time global illumination. Compared to the Light Propagation Volumes (LPV), the independency can approximate indirect high-frequency specular lighting in VCT. For the VCT, the construction of the Sparse Voxel Octree (SVO) (Crassin & Green, 2012) (Laine & Karras, 2010) is fundamental. First we are going to review the sparse voxelization, and based on the voxelized scene, we will find the way to apply VCT on an SVO.

2.5.3.1 Sparse Voxelization

There are many ways to voxelize the scene. Here, we will stick to using sparse voxelization for memory space efficiency. For a more efficient search time, we manage the set of voxels in octree form. By constructing the Sparse Voxel Octree (SVO), we can achieve a reduction of search time. In this chapter, we will look though the voxelization pipeline and how to efficiently generate the SVO.

2.5.3.1.1 One-pass Voxelization Pipeline

The traditional algorithm for voxelization is done by running multiple passes along the six main axes. An increase in the number of rendering passes is closely tied to a decrease in performance. Reducing the number of passes is one possible optimization in rendering. It is possible to voxelize the scene in one-pass. In this section, we are going to see this algorithm (Crassin & Green, 2012).

First, each triangle is projected orthogonally along the dominant axis of its face normal. The dominant axis is one of positive three main axes of the coordinates that maximizes the number of pixel fragments on the projecting plane. The dominant axis is chosen dynamically by a per-triangle unit in a geometry shader:

$$l_{\{x,y,z\}} = \left|N_p \cdot V_{\{x,y,z\}}\right|$$

The $l_{\{x,y,z\}}$ is the projected areas between triangle's normal N_p and the three main axes $V_{\{x,y,z\}}$. From the three main axes, the one axis that maximizes projected area is selected as the dominant axis (Figrue 34).

Once the axis is determined, project each triangle onto the screen by the orthogonal projection matrix along the chosen axis in the geometry shader. The projected fragments become scene voxels. For these voxels, the resolution is determined by the viewport resolution. To set a

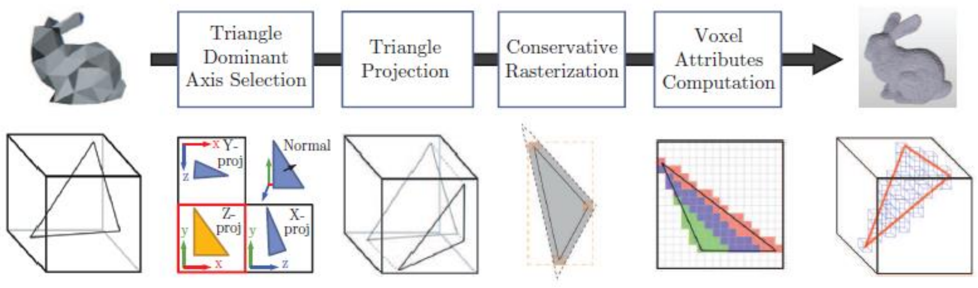

Triangle Dominant Axis Selection
Triangle Projection
Conservative Rasterization
Voxel Attributes Computation
Y-proj
Normal
Z-proj
X-proj
x
y
z

Figure 34. One-pass voxelization pipeline (Crassin & Green, 2012)

resolution appropriate for voxelization, set the viewport size properly. Normally, the converted voxels from the scene are written into a 3D texture directly. As an optimization, disabling the depth and color writes and depth testing reduces unnecessary calculations during the voxelization pass.

As the input of pixel shaders, the rasterized fragments generate corresponding voxels. To complete the generation of voxels, we need to store the following attributes into a voxel: 2D fragment position, depth, and screen space derivatives. It is straightforward to understand that the 2D pixel position and depth are needed to find the 3D texture coordinate for a voxel, but it is somewhat less apparent that the screen space derivatives are used for compacting the voxel resolution.

In the end, individually generated voxels basically contain 3D coordinates stored in a compact integer by using each 10 bits for x, y and z. They also include additional attributes needed for the use of the SVO (Sparse Voxel Octree). For instance, the indirect illumination needs normals, light intensity, and light direction. For ambient occlusion, each voxel should have extra blocking potential attributes. The resultant voxel fragments are written into 3D textures.

2.5.3.1.2 Conservative Rasterization

Without conservative rasterization, voxelization may cause holes between voxels. In other words, those voxels which don't completely cover the edges of a triangle will result in a set of sparsely generated voxels. A conservative rasterization process includes all pixels for which the intersection between a pixel cell and the polygon is non-empty, which clears up these cavities between voxels.

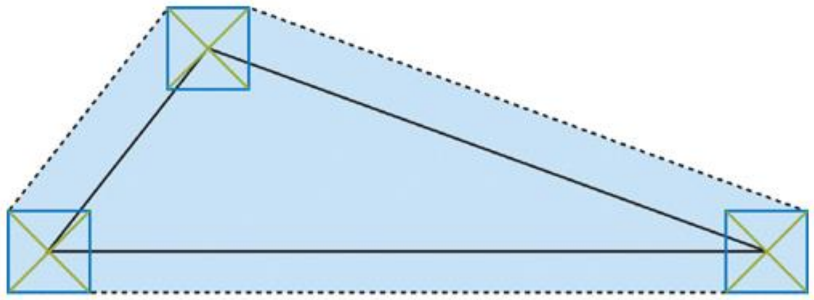


Figure 35. Overestimated Conservative Rasterization (Hasselgren, et al., 2005)

In conservative rasterization, there are two types of operations – one for overestimation and one for underestimation (Hasselgren, et al., 2005). For our case, the overestimation fills cracks resulting from scene voxelization (Figure 35)

An increase of geometric complexity is inevitable with conservative rasterization. To make use of shaders, we enlarge a bounding triangle for every input triangle in each of semi-diagonal directions, instead of computing the optimal bounding polygon in as many as 9 vertices in vertex shader or geometry shader. However, this causes poor bad approximation for acute−angled triangles. To work around this problem, an alternative interpretation is introduced in the pixel shader – the intersection of a bounding polygon and the axis–aligned bounding box (Figure 36)

We can do conservative rasterization following steps:

1. Define three edge equations, computing the normal of the line through each edge of the input triangle.
2. Move three line equations by the worst case, the semi-diagonal four directions
3. After calculating three line equations, compute the intersection points of each pair of edges to get the vertices of the bounding triangle
4. Calculate the AABB translating the original triangle vertices by semi-diagonal vectors
5. Bounding test with the AABB in the pixel shader

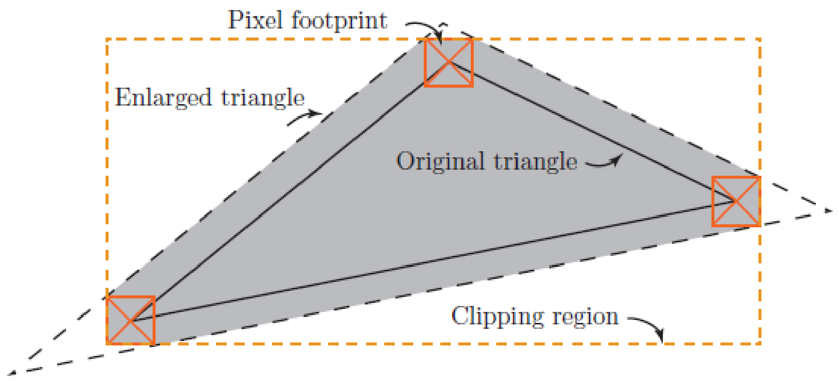


Figure 36. Improved Conservative Rasterization (Crassin & Green, 2012)

The hardest thing in these steps is to calculate the intersection points after translating the three edges. We can get the vertices of the bounding triangle by solving two quadratic equations. We simplify the calculation into a single cross product. First define the line equation:

$$(a, b) \cdot x + c = 0$$

The ordered pair (a,b) is the normal of the edge, and the $-c$ is distance from the origin to the edge. We convert the equation into the form:

$$c_{moved} = c - v \cdot (a, b)$$

Some new terms are introduced in the above equation. The c_{moved} is the distance moved from the original edge to the translated edge by the projected vector from its normal to semi-diagonal vectors v. The intersection point is derived from two translated edge equations:

$$E_0: (a_0, b_0) \cdot (x, y) + c_0 = 0$$
$$E_1: (a_1, b_1) \cdot (x, y) + c_1 = 0$$

We can rearrange the equations to solve two linear equations:

$$E_0: -\frac{a_0}{b_0}x - \frac{c_0}{b_0} = y$$
$$E_1: -\frac{a_1}{b_1}x - \frac{c_1}{b_1} = y$$

For the convenience, we substitute the terms in each equation like this:

$$\mathrm{a} = -\frac{a_0}{b_0}, b = -\frac{a_1}{b_1}, c = -\frac{c_0}{b_0}, d = -\frac{c_1}{b_1}$$

The value x and y can be derived like:

$$\mathrm{x} = \frac{(d-c)}{(a-b)}, y = a\frac{(d-c)}{(a-b)} + c$$

For x and y, derive each elements in the equations:

$$\mathrm{a - b} = \frac{a_1}{b_1} - \frac{a_0}{b_0} = \frac{a_1 b_0 - a_0 b_1}{b_1 b_0}$$

$$\mathrm{d - c} = \frac{c_0}{b_0} - \frac{c_1}{b_1} = \frac{c_0 b_1 - c_1 b_0}{b_1 b_0}$$

$$\mathrm{x} = \frac{(d-c)}{(a-b)} = \frac{c_0 b_1 - c_1 b_0}{a_1 b_0 - a_0 b_1}$$

$$\mathrm{y} = a\frac{(d-c)}{(a-b)} + c = \frac{a_0 c_1 - a_1 c_0}{a_1 b_0 - a_0 b_1}$$

Finally we know the (x, y):

$$(\mathrm{x}, \mathrm{y}) = \left(\frac{c_0 b_1 - c_1 b_0}{a_1 b_0 - a_0 b_1}, \frac{a_0 c_1 - a_1 c_0}{a_1 b_0 - a_0 b_1}\right)$$

Now we prove that the intersection point is same with the cross-product of two vectors: $n_1 = (a_0, b_0, c_0)$ and $n_2 = (a_1, b_1, c_1)$. Let's compute the cross-product of the two vectors:

$$n_1 \times n_2 = \begin{vmatrix} x & y & w \\ a_1 & b_1 & c_1 \\ a_0 & b_0 & c_0 \end{vmatrix}$$

$$n_1 \times n_2 = \begin{vmatrix} b_1 & c_1 \\ b_0 & c_0 \end{vmatrix} i + \left(-\begin{vmatrix} a_1 & c_1 \\ a_0 & c_0 \end{vmatrix} j\right) + \begin{vmatrix} a_1 & b_1 \\ a_0 & b_0 \end{vmatrix} k$$

After calculating the cross-product, divide the z component over the x and y components.

$$(n_1 \times n_2).\mathrm{xy} = \; (n_1 \times n_2).\mathrm{z}$$

$$(n_1 \times n_2).\mathrm{xy} = \left(\frac{c_0 b_1 - c_1 b_0}{a_1 b_0 - a_0 b_1}, \frac{a_0 c_1 - a_1 c_0}{a_1 b_0 - a_0 b_1}\right)$$

We can see that it exactly matches the intersection points that we got previously. We prove that the intersection point is written by the cross-product of two vectors defined as edge equation in a form of $(n_x, n_y, c_{dist}) = (n_x, n_y) \cdot x + c_{dist}$.

After finding the bounding triangle and the AABB, we do a boundary test in the pixel shader by finding the intersection of the bounding triangle and the AABB.

2.5.3.1.3 Inserting Attributes into voxel fragments

Once voxel fragments are generated in the pixel shader, the 3D coordinates and appropriate attributes of a voxel are written directly into the 3D texture. However, multiple voxel fragments from different triangles can fall into the same destination in an arbitrary order. The parallel processing of the GPU gives rise to unpredictable order of execution of voxel fragments in pixel shader. This causes several issues that must be solved: the writing order, flickering, and non-time coherent results.

Atomic operations work out the side effects caused by inconsistency of thread execution on the GPU. Relying on atomic operation, all attributes are averaged into the same voxel. The restriction on using an atomic operation is only valid to 32-bits signed/unsigned integer type resources. But we want to write the values into RGBA8 or RGBA16F/32F type of textures. We need another operation rather than atomic arithmetic operations. The compare-and-swap atomic operation is the thing that we are looking for. By looping on each pixel−triggered thread, write the attributes until there are no more conflicts and the values with which we have computed the sum have not been changed by another thread. This doesn't produce any conflicts between running threads using RGBA8 or RGBA16F/32F textures, but it is slower than doing atomic arithmetic operations like atomic addition.

Suppose we want to use RGBA8 texture to save on memory space. Only 8 bits are available per color channel, which makes it easy to overflow values when summing attributes in the voxel. We must compute the average incrementally each time whenever a new voxel is inserted into the voxel:

$$C_{i+1} = \frac{iC_i + x_{i+1}}{i+1}$$

When we average the values incrementally, we store the counters in the alpha channel of an RGBA texture to improve accuracy.

We project the value range from [0, 1] to [0, 255]. During the conversion, we might lose some precision; however, we can prevent conflicts between different threads using atomic operations which are only available for integer−type resources.

2.5.3.2 Sparse Voxelization into an Octree

At this point, the voxelized scene is ready for use. But, the search time causes major performance drops. The octree structure reduces the search time from n^2 to $\log(\mathrm{n})$. In other words, the search time depends on the depth of the octree. For instance, transition to the octree makes it possible to approximate global illumination in real-time rendering. The voxelized scene is stored inside the 3D texture. Storing each level on separate 3D texture is wasteful. To save on memory bandwidth, a compact pointer-based structure is introduced. The voxel octree based on a compact pointer-based structure is called a Sparse Voxel Octree (SVO). We are going to see the detail of SVO step by step to give the full insight on how this data structure is used in volumetric rendering.

2.5.3.2.1 Octree Structure

The Sparse Voxel Octree (SVO) is a point-based structure – it has linear memory organization, rejecting empty voxels and storing valid voxels in linked−list form. The picture below shows memory organization for each nodes in the octree (Figure 37).

In the figure, a node which has child nodes points to a group of eight child nodes. By doing this, we reduce the memory requirement from eight pointers for child nodes to a single pointer. Each group has 2^3 nodes. The SVO outline is maintained by buffer objects rather than any type of textures.

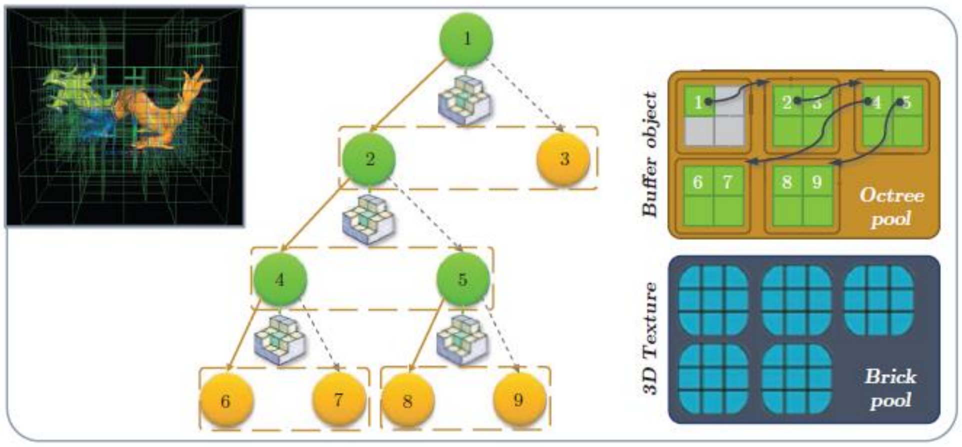


Figure 37. Illustration of Sparse Voxel Octree (SVO) (Crassin & Green, 2012)

Most attributes in a voxel need an interpolation scheme to produce smooth changes in values. If we use hardware filtering provided by the Graphics Device Interface (GDI), we can accelerate the performance rather than doing custom filtering in shaders. To do this, we introduce 3^3 bricks. The reason for having 3^3 bricks instead of 2^3 bricks is to efficiently account for boundary interpolation. Our octree structure is basically sparse and linear memory organization. To interpolate the values in a voxel, searching adjacent voxels for boundary values is wasteful. To eliminate this process, we utilize this kind of 3^3 bricks structures. See the below illustration for 3^3 bricks (Figure 37). By moving the nodes to the corners of bricks, we make use of hardware tri-linear interpolation. It increase the usage of memory, but our voxelized scene is mostly sparse. Considering the resultant performance enhancements, the additional memory space used is sufficiently tradable.

2.5.3.2.2 Sparse Voxel Octree (SVO) Construction Overview

With a voxelized scene, we need additional steps to construct the sparse octree.

First, we outline the octree with a top-down scheme. One level at a time, progressively subdivide nodes marked nodes which are marked non-empty into 2^3 subnodes. Each successive step increases the octree resolution. For each level, the non-empty nodes are detected by looping all voxels comprising the scene over the current level. At the end of a level (having reached the bottom), the voxel fragment's attributes are written to the leaf nodes. After that, it starts to mip-map the values from the bottom to the top.

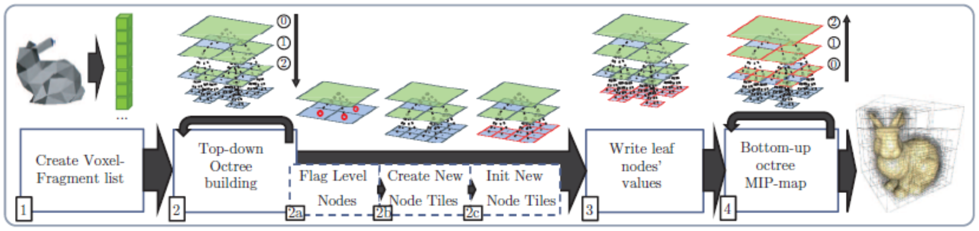


Figure 38. Description SVO construction (Crassin & Green, 2012)

To fit the scene voxels into this algorithm, it is better to store all voxels into a single list instead of in 3D textures. We are going look through how to create the voxel fragment lists for the SVO.

2.5.3.2.3 Voxel Fragment List Construction using atomic operation

The voxel fragment list is the data set used for subdividing the tree during the octree building process. Each level of the octree should determine which nodes are non-empty, and which are allocated. The voxel fragment list is handy for looping whole voxel fragments rather than finding and looping non-empty voxels in 3D textures. Writing the attributes in pixel shader during the voxelization over the scene, we encounters the case where multiple voxel fragments from different triangles fall into same position in the 3D texture. To resolve this, we use the atomic swap−compare operation for averaging the values. For the voxel fragment list, we don't need to care about the conflicts caused by this situation. We simply put all voxel fragments into the single buffer. For here, we make sure that each voxel fragment has a unique index to allow concurrent access by thousands of threads. The atomic counter operation is used for assigning a unique index for each voxel fragment in the list.

2.5.3.2.4 Node Subdivision

When the node is marked as a non-empty node, this node should be subdivided into 2^3 nodes. Whether a node is non-empty is determined by each thread looping though nodes residing in the current depth of the octree per entry of the voxel fragment list. This process is repeated from top to bottom, setting the most significant bit of the children pointer of the node. For marking, we don't need to be worried about conflicts between different threads. As stated before, we simply set the most significant bit, which removes interlocks between threads and increases performance.

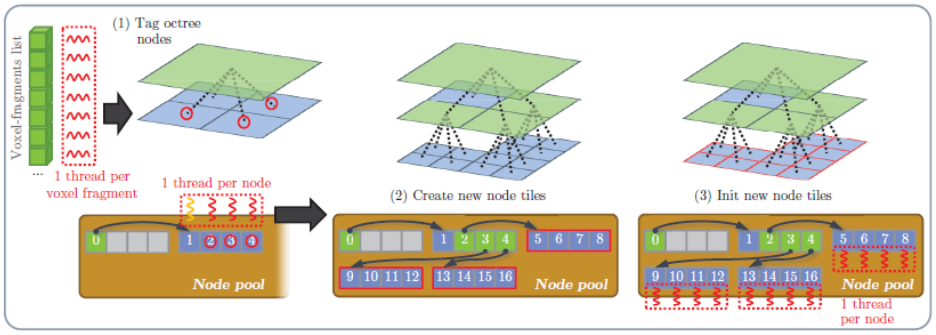


Figure 39. SVO Node Subdivision (Crassin & Green, 2012)

Whenever subdividing the 2^3 node tiles, allocate them into the octree pool and link them properly. For allocating the subnodes, using a separate scheme for executing threads is one optimization for equalizing the termination time for threads.

After competing thread subdivision, the new allocated nodes are initialized. We also allocate one brick per group of child nodes to store all appropriate attributes. The thread scheduling scheme is applied to same scheme in subdividing the node. We trigger all threads for each node at the current depth of the octree. Finally we are ready to write all values into leaf nodes and mip-map the values from bottom to top.

2.5.3.2.5 Writing and Mip-Mapping Values

With the constructed SVO, the leaf nodes of the SVO should be filled up before mip-mapping the values. However, we store all attributes into a separate data structure called the brick pool. Each brick consists of 3^3cubic voxels for hardware filtering. But we assign 3^3 voxels in the brick from values from 2^3nodes. The input for the brick is insufficient. We need another strategy to insert the values properly. See the picture below:

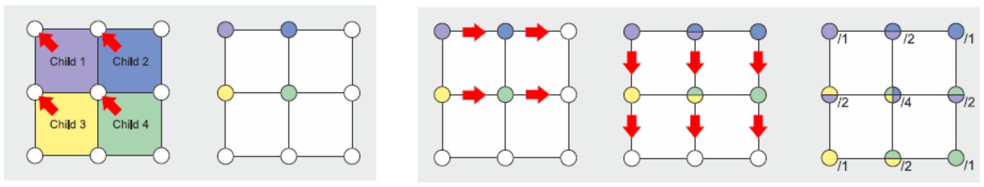


Figure 40. Filling attributes from leaf nodes to octree nodes (Drınovsky, 2013)

In the figure, we initialize all values from the nodes to the brick starting from the left-top corner. We need three passes along three main axes (the x, y and z axes). For each axis, we move the values along the axis and average two values, putting the result into middle voxel. We repeat this process for the rest of main axes.

The same thing that we had done in the simple voxelization pipeline is considered to write the values into leaves of the SVO. Multiple voxels in the voxel fragment list fall into same node. In this case, we use the same scheme used to prevent conflicts from multiple threads writing to the same voxel. When we writes the attributes, we generate the same number of threads as the number of elements inside the voxel fragment list. It is natural that multiple voxels will try to assign their values in the same node. The atomic compare-and-swap operation works to average the values without any conflicts.

After filling up leaf nodes and bricks, we start mip-mapping from bottom to top. Each step, we trigger as many threads as the number of non-empty nodes in the current depth of the octree. Each triggered thread is totally independent from the others, so we don't need to worry about any conflicts.

2.5.3.2.6 Draw Indirect

The generation of threads is incurred by drawing each pixel in the fragment shader. When triggering threads, the transferrence of geometric information to the GPU memory each time is wasteful for memory bandwidth. To do this, OpenGL 4.2 or DirectX 11 supports a draw-indirect method. We can use this to produce threads without any draw calls which requires copying geometry to the GPU. With this method, we can reap the benefits of hardware rasterization like triggering exact threads. We are also free from any disadvantages.

2.5.3.3 Voxel Global Illumination

The geometry independency benefits from the Sparse Voxel Octree (SVO) helps to increase the performance of global illumination algorithms, especially indirect illumination, which is regarded as the heaviest calculation in GI. For indirect illumination, voxel cone tracing gives us additional effects with less cost, including soft shadow cone tracing, ambient occlusion, and indirect specular illumination; it is superior to other global illumination algorithms such as Imperfect Shadow Maps (ISMs) or Light Propagation Volumes (LPVs).

A few months ago, NVIDIA announced their real-time global illumination tech demo named VXGI (Voxel Global Illumination) (Panteleev, 2014). A studio named Q-Games has successfully adopted a modified voxel cone tracing method called Cascaded Voxel Cone Tracing for use in their games, *The Tomorrow Children* (McLaren, 2014). Even though the Unreal Engine dropped the VCT a while ago, they tried to implant the VCT into their engine (Mittring, 2012). Recently, various VCT algorithms haven been researched and widely adopted into rendering engines to give a more realistic experience to users. Next, we are going to look at the detail of the original Voxel Cone Tracing (VCT) algorithm (Crassin, et al., 2011).

2.5.3.3.1 Algorithm Overview

We have already covered how to construct the Sparse Voxel Octree (SVO). So, suppose that the SVO is ready to use. For generating indirect light sources called Virtual Point Lights (VPLs), we render the Reflective Shadow Maps (RSMs) from the light's point of view, baking the irradiance and light direction to corresponding nodes in the SVO. The directional opacity and radiance are stored in the form of Gaussian Lobes. We had already covered the Gaussian Lobe (Toksvig, 2004).

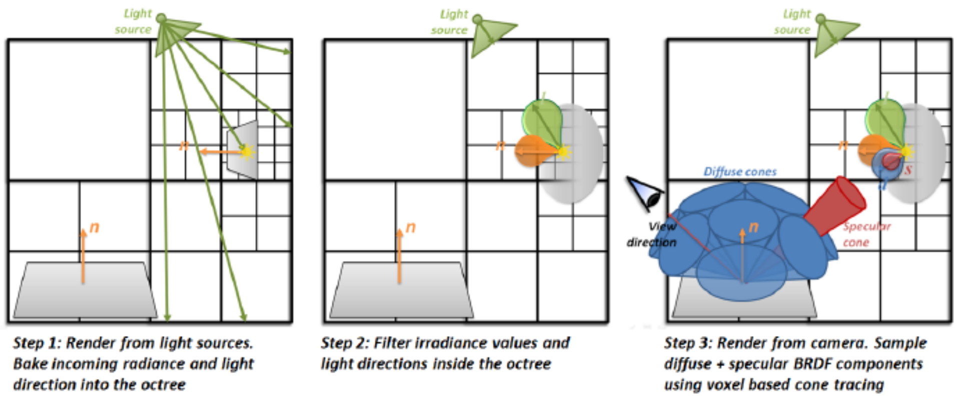

Light source
n
Step 1: Render from light sources. Bake incoming radiance and light direction into the octree
Light source
n
Step 2: Filter irradiance values and light directions inside the octree
Light source
Diffuse cones
View direction
n
Specular cone
S
Step 3: Render from camera. Sample diffuse +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

Figure 41. Voxel Cone Tracing Overview (Crassin, et al., 2011)

The wide lobes are used to store diffuse radiance distributions for incoming lights, and the narrow lobes are used to represent the indirect specular radiance. After injecting all VPLs from RSMs, start to filter all attributes residing in the current level to the upper level by mip-mapping the values in the octree. The final step is to render the scene from the camera view and to sample diffuse and specular BRDF components using voxel cone tracing to perform the final gathering. For diffuse indirect lighting, to account for the wide cosine lobe, we need to trace roughly more than 5 or 6 cones to cover the hemisphere. For specular lighting, tracing one narrow cone with the aperture derived from the specular exponent for glossy material is enough.

2.5.3.3.2 Structure Description

The structure in the SVO allows us to query filtered scene information such as radiance, irradiance, occlusion, or geometric normals distribution. The SVO allows us to handle larger and more complex scenes without incurring further costs. We already covered the SVO structure in previous chapters, so we will only review it briefly.

For GPU representation, we keep two memory pools. The first is the octree node pool in linear GPU memory, grouped into 2^3node tiles which are pointed to a single index, rather than all eight indices. Each node has a pointer to a brick in the texture object. When voxelizing the scene, most of voxels remain empty. This implies that a higher resolution is more efficient in terms of memory space with a sparse memory representation. The second is the brick pool structure, a single brick contains a total of 3^3 voxels for interpolating the boundary without finding adjacent nodes. Setting the voxel centers to be centered at the node corners, we can make use of hardware filtering (Figure 42).

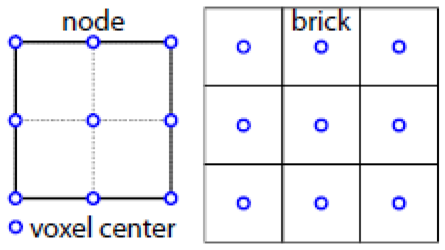


Figure 42. Brick pool structure (Crassin, et al., 2011)

Compared to adding separate boundary data structures like keeping additional pointers toward neighbors, it requires less-memory consumption and improve the performance by hardware filtering.

2.5.3.3.3 Mip-Mapping

The mip-mapping is an intriguing part of the voxel cone tracing algorithm because of the brick pool. An individual brick in the pool shares its boundaries sparsely, which makes it hard to mip-map the values via hardware filtering. We do mip-mapping programmatically in the shader.

For smooth filtering, we want to do Gaussian Filtering. Defining the Gaussian kernels and convolving it to the bricks is irritating. If we make use of overlapped boundaries, we can do the filtering easily. For ease of explanation, we consider the 2D case (Figure 43). Suppose that we want to do mip-map the center voxels. Before filtering, take some time to look at the picture above. When we add the contributing voxels to the mip-mapped center voxel, the center voxel has multiplicity 4, edge voxels have multiplicity 2 and corner voxels have single multiplicity. It looks like the exact Gaussian distribution. After adding the values like this, we divide the total multiplicity over 9 voxels and sum them up, producing the Gaussian average. This also calculates the average easily with parallelizable, GPU friendly-calculations.

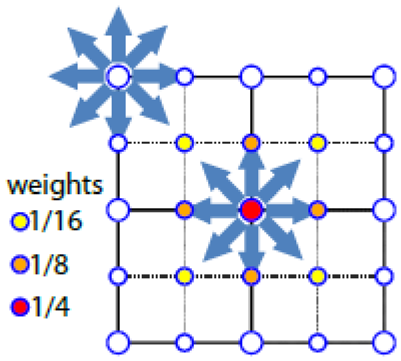


Figure 43. SVO Mip-mapping (Crassin, et al., 2011)

2.5.3.3.4 Voxel Representation

Most attributes stored in a voxel are directional data. To store directional data in limited channels in texture, we keep it in the form of the Normal Distribution Function (NDF). We can apply it to store the normal or light direction for indirect illumination. But storing arbitrary distributions is too memory intensive. To reduce the bandwidth, we make the assumption that all distributions can be represented as isotropic Gaussian lobes characterized by the average vector D and a standard deviation σ. To ease interpolation, we calculate the variance with the average vector (Toksvig, 2004):

$$\sigma^2 = \frac{1 - |D|}{|D|}$$

For indirect illumination, we also store the blocking potential in form of visibility. In other words, this represents the percentage of blocked rays. In the Light Propagation Volumes (LPVs), the potential occlusion is the same in terms of storing opacity for each voxel with the isotropic Gaussian Lobes representation. To avoid the loss of accuracy of the visibility value, rather than storing a single opacity value, we will also use the NDF for opacity. The color or normal values stored in voxels are weighted by an opacity in order to correctly take its visibility into account.

2.5.3.3.5 Voxel Shading

The voxel shading is the approximation of shading for each voxel. Each voxel contains a set of attributes as Gaussian lobes. We will translated it into convolutions for different Gaussian lobes. For instance, the convolution between the normal and the BRDF gives us the probability density of the BRDF over the effective normal (Figure 18).

For any attribute for a voxel, we can represent the Gaussian lobe such as the BRDF or span of view cones as the following:

$$\sigma_n^2 = \frac{1 - |N|}{|N|}$$

For the view cone, we can calculate the deviation for the intermediate angles (Figure 44):

$$\sigma_v = \cos\varphi\,, where\ \varphi\ is\ view\ cone\ apature$$

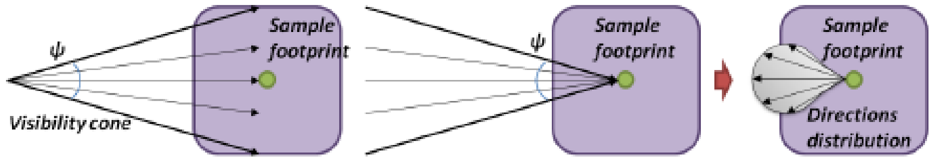


Figure 44. The description of the direction distribution (Crassin, et al., 2011)

2.5.3.3.6 Dynamic Updates

To speed up the construction or updating of the Sparse Voxel Octree (SVO), we need to focus on the fact that in many cases, large parts of the scene are usually static or updated locally upon user input. Only fully-dynamic objects need a per-frame SVO construction. We could store the octree structure for these objects separately, but rather than that, we will keep them in same octree structures for ease of traversal and unified filtering. But the update frequency for semi-static objects and dynamic objects is different. A different time-stamp mechanism is required to differentiate them. To prevent overwriting static octree nodes and bricks, we put them at the end of the node pool and brick pool.

2.5.3.3.7 Voxel Cone Tracing

The traditional method for global illumination is through the ray-tracing. The ray-tracing method triggers lots of sampling rays. Casting more sampling rays results in a more realistic scene, but it is not appropriate to use in real-time rendering. It is too expensive. Fortunately, global illumination is spatially and directionally coherent; to exploit this feature, voxel cone tracing is introduced. For further efficiency, the SVO is used for the voxel cone tracing to approximate the result for all rays more efficiently.

The voxel cone tracing algorithm steps along the cone axis and performs the lookups in our hierarchical representation at a level in accordance to the cone radius (Figure 45).

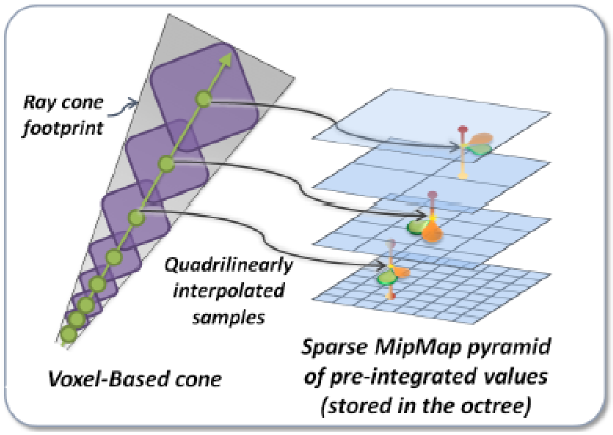

Ray cone footprint
Quadrilinearly interpolated samples
Voxel-Based cone
Sparse MipMap pyramid of pre-integrated values (stored in the octree)

Figure 45. Illustration for Voxel Cone Tracing (Crassin, et al., 2011)

A smooth variation is achieved by quad-linear interpolation. Even though it is not correctly calculated in mathematics, this approximation produces a plausible result. Stepping along the cone, the classical emission-absorption optical model is used to accumulate the values along the cone. Suppose the current occlusion α and color value c. In each step, we retrieve from the appropriately filtered scene information the occlusion α_2 for new outgoing radiance c_2, then update the values with front−to−back accumulation like this:

$$c = \alpha c + (1-\alpha)\alpha_2 c_2$$

$$\alpha = \alpha + (1-\alpha)\alpha_2$$

To ensure good integration quality, we can do correction upon updating the occlusion. The distance d' between successive sample locations along a ray does not coincide with the current voxel size d.

$$\alpha_s{}' = 1-(1-\alpha_s)^{\frac{d'}{d}}\ for\ smaller\ step$$

With this correction, we can increase the quality of filtered values for small steps along the cone. This calculated occlusion is used not only for stepping along the cone, but also for ambient occlusion.

2.5.3.3.8 Ambient Occlusion

Ambient occlusion is a simple application of voxel cone tracing. By covering this topic, we can get some insight into the usefulness of voxel cone tracing. There is no difference between the diffuse indirect lighting and generating cones over the hemisphere. This implies that while approximating diffuse indirect illumination, we can get ambient occlusion without any additional costs.

Suppose that there is ambient occlusion $\mathrm{A(p)}$ at a surface point p.

$$\mathrm{A(p)} = \frac{1}{\pi}\int_{\Omega} V(p,w)(\cos w)dw$$

$$\mathrm{V(p,w)} = \begin{cases} 1, & \text{if the ray intersects the scene} \\ 0, & \text{otherwise} \end{cases}$$

We weight the blocking potential by a function $\mathrm{f(d)}$ which decays over the distance:

$$\mathrm{f}(r) = \frac{1}{(1+\gamma r)}$$

$$\alpha_f(p+rw) = f(r)\alpha(p+rw)$$

Until now, we have dealt with the ambient occlusion calculated from ray-tracing or path-tracing. To approximate it via the voxel cone tracing, partition the hemisphere into the sum of serial integrals.

$$\mathrm{A(p)} = \frac{1}{N}\sum_{i=0}^{N} V_c(p,\Omega_i)$$

$$V_c(p,\Omega_i) = \int_{\Omega_i} V_{p,\theta}(\cos\theta)d\theta$$

The definition $V_c(p,\Omega_i)$ resembles the cone over the hemisphere (Figure 46).

We can approximate ambient occlusion contribution by voxel cone tracing. In other words, accumulating the blocking potential accounts for the weight $\mathrm{f(r)}$, which implies a decay over the

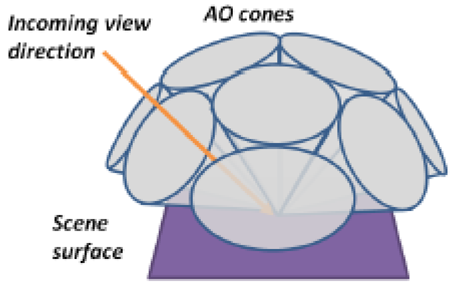


Figure 46. Ambient Occlusion (AO) by Voxel Cone Tracing (Crassin, et al., 2011)

distance. The sum of the contributions of all cones gives the approximation of the ambient occlusion on the point p.

2.5.3.3.9 Indirect Illumination

The voxel cone tracing algorithm is a method of one-bounce indirect illumination. For each voxel, we write the irradiance, rather than the outgoing radiance. By doing this, we can achieve one-bounce indirect illumination by gathering the light intensity with the voxel cone tracing. For efficiency, we use the G-buffer from deferred rendering to avoid hidden surface cone-tracing in the final rendering pass. The diffuse indirect illumination is the one of heaviest calculations in the Voxel Cone Tracing (VCT). Eliminating the cones for diffuse lighting on those surfaces hidden from the camera reduces the bottleneck in the performance.

For indirect illumination, we inject the VPLs into voxel fragments. To do this, we rasterize the scene from each light's point of view. Each pixel represents a photon that we want to bounce in the scene. We store the properties of these photons as a directional distribution and a scalar energy proportional to the subtended solid angle of the pixel as seen from the light. To splat the photons, we use fragment shader to trigger a thread for each Reflective Shadow Maps (RSMs) pixel. Then, we traverse the SVO from top to bottom and find the leaf node where the light intensity will be inserted. Just in case, the light intensity is written into the voxels sparsely. In other words, we can see cracks between voxels. This is caused by the lack of RSM resolution (Figure 47[2]).

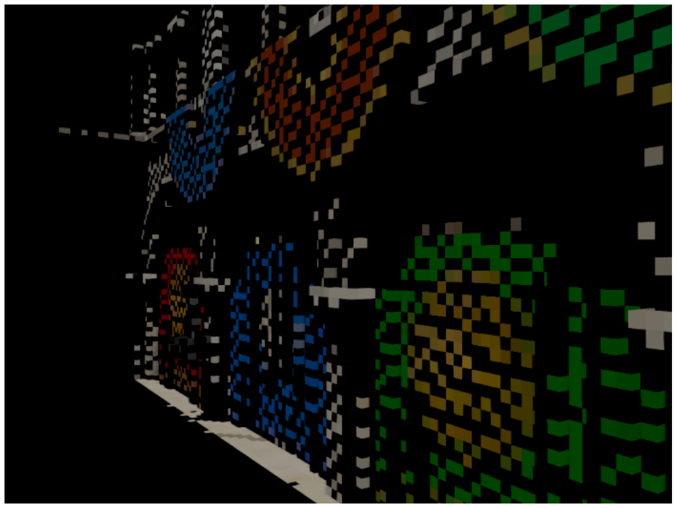


Figure 47. Artifacts from lack of RSM resolution

[2] http://simonstechblog.blogspot.com/2013/01/implementing-voxel-cone-tracing.html

It can differ depending the complexity of the voxelized scene, but normally a resolution 2x that of the highest resolution of the SVO is enough for the resolution of RSMs. Injecting the photons produce the same race condition problems found in voxelization, where multiple voxels from different triangles fall into the same location. To handle this, we should use an atomic operation.

There is another hurdle to overcome - the voxels are repeated for adjacent bricks. We need a more efficient transfer scheme. To efficiently average the boundary of each brick, a new algorithm is introduced. Let's examine the left picture of Figure 48.

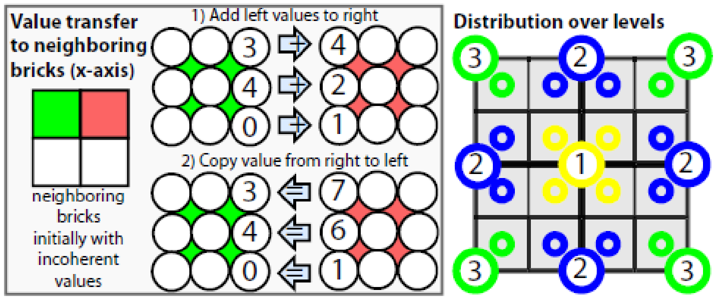

Value transfer to neighboring bricks (x-axis)
1) Add left values to right
3
4
0
4
2
1
neighboring bricks initially with incoherent values
2) Copy value from right to left
3
4
0
7
6
1
Distribution over levels
3
2
3
2
1
2
3
2
3

Figure 48. Boundary values transfer scheme (Crassin, et al., 2011)

We do six passes, two (negative and positive) for each main axis (x, y and z). Along the positive main three axes, we average the left values and right values. Along the negative main axes, we copy the right values to the left values.

For further optimization when mip-mapping the values from bottom to top, there is room to overcome inefficiencies due to thread scheduling; the computation cost of each thread differs depending on which voxel is being processed, leading to unbalanced scheduling. We can mitigate this by introducing three separate passes to make sure that all threads have roughly the same cost. Let's examine the right picture in Figure 48.

In the picture, each pass is represented by a different color. For the first pass, we do mip-mapping on the center of the voxel, which needs 9 voxels to average. In the second pass, we do this on the four face centers of the voxel – each face center needs 6 voxels from child nodes. Lastly, for the third pass, all four corners needs 4 voxels to average. This roughly manages to balance the thread termination. We can easily extend our 2D case to the 3D version.

After injecting the VPLs into leaf nodes, we filter the intensities from the leaf nodes to the root node. During the filtering, we launch threads for all nodes, even including those that did not contain any photon, where the filtering was applied to zero values. In fact, the voxels to be filtered by photons are only a small part of the scene. It is crucial to avoid the filtering of zero values. We can do this by launching a thread per RSM pixel and finding the node in the level on which the filtering should be applied and executing it. As a result, we can reduce the number of threads to get close to the optimal number set of threads set for each filtering pass.

2.5.3.3.10 Anisotropic Voxels for Improving Cone-tracing

Up to now, we have covered the isotropic voxel cone tracing for indirect lighting. This method causes some quality problems. Consider the following red-green wall problem to understand these artifacts.

The problem occurs whenever two opaque voxels with different colors are averaged in the upper level of the octree, as illustrated in this picture (Figure 49).

We can see the artifact here; their colors are mixed as if the two walls were semi-transparent. Since we store not only the intensity but also the opacity, the opacity has same problem. Over the upper levels, it becomes fully transparent from semi-transparent. To prevent this and enhance the visual quality, the anisotropic voxel representation for irradiance is introduced. Instead of storing a single non-directional value, we store 6 channels of directional values, in the x, y, z, -x, -y, and -z directions. To filter anisotropic voxels by mip-mapping, we need a pre-integration step (Figure 49).

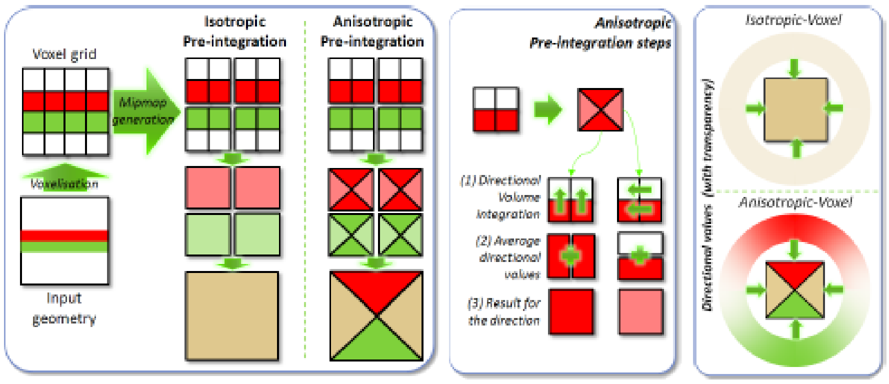

Voxel grid
Mipmap generation
Voxelisation
Input geometry
Isotropic Pre-integration
Anisotropic Pre-integration
Anisotropic Pre-integration steps
(1) Directional Volume integration
(2) Average directional values
(3) Result for the direction
Directional values (with transparency)
Isotropic-Voxel
Anisotropic-Voxel

Figure 49. Illustration Anisotropic Voxels (Crassin, et al., 2011)

The three steps required for pre-integration are:

1. Direction of volume integration
2. Average direction of values
3. Store the average into a separate channel for directional value

By storing anisotropic voxels, we can fully interpolate directionally. It is easy modification to store the voxels in our bricks, but it takes an additional 1.5x memory consumption. Taking the memory space into an account, we will consider the introduction of anisotropic distribution.

2.5.3.3.11 Conclusion

The voxel cone tracing algorithm was incorporated into the Unreal Engine 4 (UE4), but the UE4 decided to drop the technique for the VCT. Currently, it is too computationally heavy to construct the octree for the new generation of consoles, the PS4 and XBOX ONE. But for the representative game engine named ‘Panta Rhei’, Capcom has implanted the modified VCT for their global illumination calculation. Even though, they gave up on the SVO (Sparse Voxel Octree) for the same reasons as UE4, they increased the performance of the voxel cone tracing algorithm without the octree structure by making use of 3D texture mip-mapping hardware filtering and reducing the texture resolution. Over the LPV for which the indirect low-frequency lighting is available, the VCT can well-approximate plausible indirect mid−to−high frequency lighting. If we research and develop an innovative algorithm for processes on the SVO, the Sparse Voxel Octree Global Illumination (SVOGI) might garner attention once more, as it did the first time it appeared in the field of Global Illumination (GI) algorithms.

2.6 Summary

We have covered all necessary background for global illumination, as well as real-time global illumination techniques widely used in rendering engines. Currently, the Light Propagation Volume (LPV) and the Voxel Cone Tracing (VCT) are influential techniques. The VCT can represent a wider range of frequency lighting than indirect illumination. However, the VCT is less efficient than the LPV in terms of performance. In next chapter, we are going to analyze these problems and suggest a new way of using VCT named ‘Temporal Voxel Cone Tracing with Interleaved Sample Patterns’.

3. Problem Analysis

The Voxel Cone Tracing in (Crassin, et al., 2011) is too heavy to apply using the current hardware of the new generation of consoles. It is the main reason that Epic Games dropped this technique from their engine, Unreal Engine 4. In this section, we are going to analyze which sources cause poor performance for the VCT and suggest solutions to make this technique feasible in real-time rendering, which will be covered in detail in the following chapter.

3.1 The Sparse Voxel Octree (SVO)

In the paper (Crassin, et al., 2011), constructing a Sparse Voxel Octree (SVO) is a necessary preparation before sampling voxels hierarchically in the voxel cone tracing. The construction of the SVO takes too much time, which is the main performance flaw. Specifically, the SVO requires custom mip-map filtering on update, because it is comprised of sparsely allocated bricks. This prevents us from using hardware filtering in mip-mapping. The poor cache usage of the GPU causes a further decrease in performance, compared to the hardware filtering. If we can use the hardware filtering for mip-mapping multiple resolution 3D textures for voxel storage, we can get a speed-up from not only the mip-mapping, but also the hardware quad-linear filtering, which more efficiently samples voxels in cone tracing. In the end, we get various merits in terms of performance.

To support hardware filtering, we must make the choice to dispose of the octree structure and instead utilize the 3D mip-textures for storing voxels. Therefore, it may require more memory than the sparse voxel octree approach. By adopting this, it gives us huge speed benefits − enough to consider applying the voxel cone tracing algorithm for indirect diffuse illumination on current generation consoles, taking into account that these consoles support lots of memory, comparable to a high-end desktop. So, it is decided to drop the SVO to improve performance by sacrificing memory storage.

3.2 The Indirect Diffuse Illumination

The diffuse indirect illumination is the heaviest part of Voxel Cone Tracing (VCT). At the very least, each pixel should span 6 cones to approximate the reference of the path-tracing. To tell the truth, using even 6 cones for indirect diffuse illumination is insufficient; the aperture for each cone is 60 degrees, which has great potential to skip thin geometries. Therefore, using more cones approaches a better approximation of the reference.

The diffuse voxel cone tracing gives us the free volumetric Ambient Occlusion (AO) superior to the Screen-Space Ambient Occlusion (SSAO) (Panteleev, 2014) (Figure 50).

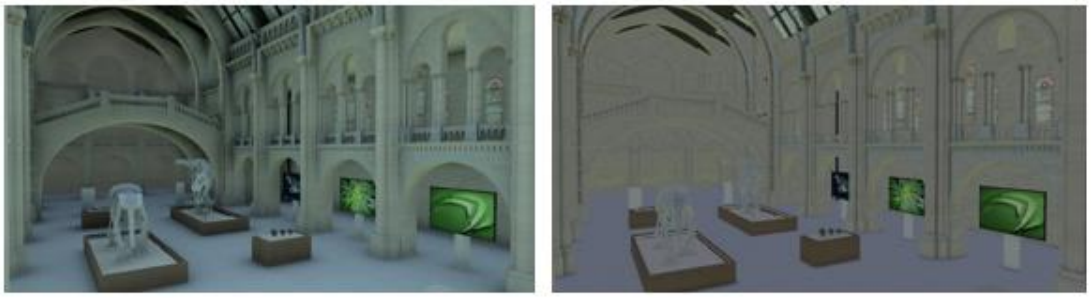


Figure 50. Volumetric Ambient Occlusion (Left) and SSAO (HBAO+) (Right) (Panteleev, 2014)

This follows the same principle as ray-tracing; as the number of cones increases, a better approximated AO of the scene is guaranteed. The importance of the number of cones cannot be overstated. However, it is computationally too heavy to generate such a large number of cones per pixel, so it is suggested to use Interleaved Sampling for the VCT (Keller & Heidrich, 2001) (Segovia, et al., 2006) (Wald, et al., 2002).

In the interleaved sampling algorithm, we group $\mathrm{n} \times \mathrm{m}$ pixels on the Geometry Buffer (G-Buffer). Each pixel in an individual group will be assigned to a different cone direction, but each group has same pattern as all other groups. The interleaved sampling gives rise to structured noise artifacts. The bilateral blur removes the artifact with a discontinuity buffer (Figure 51).

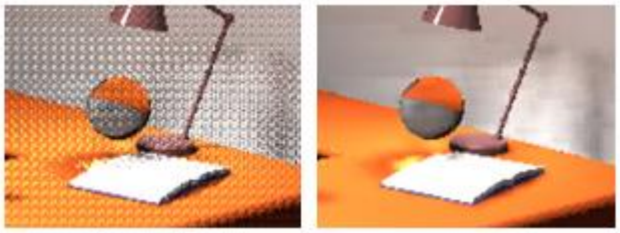


Figure 51. 5x5 interleaved sampling (Left) and 5x5 interleaved sampling 5x5 discontinuity buffer (Right) with Virtual Point Lights (VPLs) (Wald, et al., 2002)

We can hugely reduce the computational complexity by assigning each pixel to trace one cone, but we can approximate the outcome after doing $\mathrm{n} \times \mathrm{m}$ cone tracing per pixel.

3.3 Unstable Interleaved Sampling for Global Illumination

When we use interleaved sampling in voxel cone tracing, we encounter artifacts caused by the inconsistencies in sampling different cone direction for the same world position in the screen-space. Specifically, a pixel from the same position can assigned a different cone direction and trace the cone in a different frame. During the cone tracing, all pixels corresponding to the same world position should be assigned the same direction. As a result, this inconsistency causes noisy artifacts (like sinuous movement) whenever the camera moves or rotates. To reduce these artifacts, we need to use the Reverse Reprojection (Scherzer, et al., 2007) (Nehab, et al., 2007) and Temporal Filtering (Bavoil & Andersson, 2012) (Mattausch, et al., 2014).

3.4 Conclusion

We have looked through the problems that we could face using the improved voxel cone tracing algorithm. For each problem, the appropriate solutions should be applied. In the next chapter, we are going to deal with them in detail, covering the problem analysis further.

4. Contribution

In chapter 3, we went through the problems with the improved VCT and suggested some brief solutions. First, we are going to suggest a new, more concise voxel cone tracing algorithm by removing the Sparse Voxel Octree (SVO), enabling hardware filtering in the mip-mapping of voxel data and having the cone tracings sample voxels hierarchically. A similar concept is introduced in (McLaren, 2014)(Rauwendaal, 2014). They suggest removing the SVO to improve performance and make the overall steps simpler. The extra indirections for volume access decreases the performance to sample voxels (Rauwendaal, 2015). In this part, we are going to compare the results between SVO and Voxel Mips. Considering a section for SVO construction results in (Drınovsky, 2013), you can observe that the Voxel Mips is more efficient than the SVO in speed.

Next, the interleaved sampling for the VCT will be introduced to accelerate the calculation of the indirect diffuse illumination. For interleaved sampling, we need to split the geometry buffer (G Buffer) by arbitrary patterns like $\mathrm{m} \times \mathrm{n}$. The splitting method by one pass is cache inefficient for texture access. To make it more performant, the two pass algorithm is introduced (Segovia, et al., 2006). The two pass algorithm makes memory access coherent by subdividing the G Buffer in $\mathrm{p} \times \mathrm{q}$ blocks in advance. The interleaved sampling always comes with bilateral blurring to remove structured artifacts caused by the interleaved sampling.

By introducing interleaved sampling, we may face the several artifacts discussed in the previous chapter. To resolve the artifacts, we will introduce the reverse reprojection and temporal refinement algorithms. The reverse reprojection reuses the results which were already calculated in the previous frame, but it can cause another artifact called ‘trailing artifacts’. To remove the artifacts, the temporal filtering algorithm is used. The temporal filtering converges the previously calculated results to current results with some ratio. By doing that, we can remove the trailing artifacts. The details will describe the experience implementing these algorithms alongside sufficient images captured from the voxel cone tracing demo. From now on, we will refer to the improved voxel cone tracing technique as ‘Temporal Voxel Cone Tracing with Interleaved Sample Patterns’.

4.1 The Voxel Mips for the Voxel Cone Tracing

The development of a new method of voxel cone tracing (VCT) is motivated by the inefficiency of the Sparse Voxel Octree (SVO). It erases the heaviest process in constructing the SVO and adopts new voxel data structures called 'Voxel Mips', or multiple−resolution 3D textures. It enables the use of hardware filtering for mip-mapping, which boosts performance. The new way to represent voxel data will be constructed in the following way (Figure 52):

1) One-pass simple voxelization at full-resolution
2) Inject the direct illumination data into appropriate voxels in the highest resolution texture in the voxel mips from the Reflective Shadow Maps (RSMs)
3) Generate the voxel mips with the hardware's linear filtering

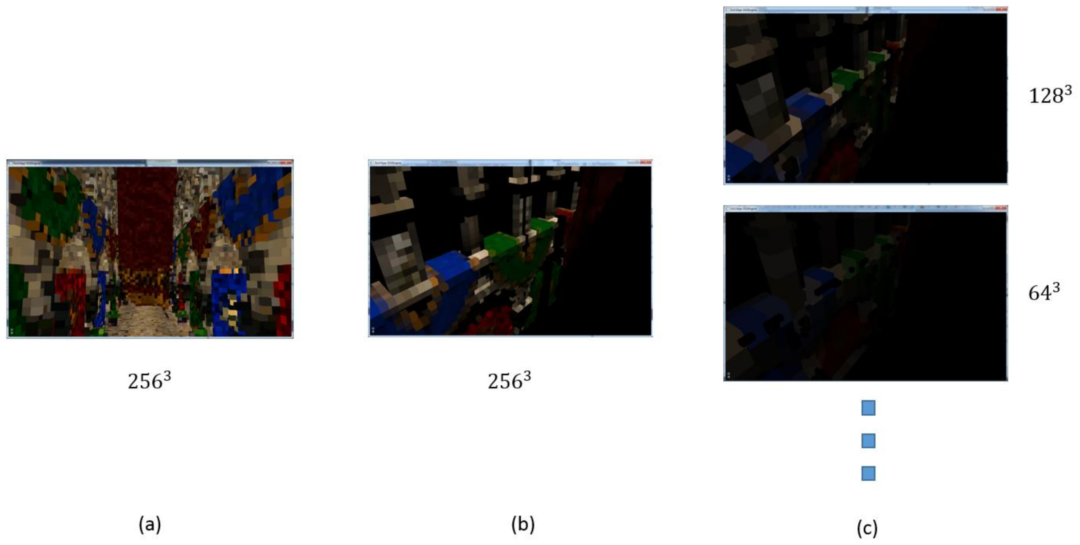


Figure 52. Illustration of the voxel mips, (a) The one-pass simple voxelization of the scene, (b) injecting the direct illumination into highest voxel mips, (c) Generate the voxel mips with hardware filtering

By removing the SVO, the overall number of steps required to voxelize the scene is reduced. The details of the one-pass voxelization method and the injection of the Virtual Point Lights (VPLs) from RSMs remain as described in previous sections. The main difference here is that, instead of using an SVO that requires custom mip-mapping (which is the heaviest performance hot spot (Figure 53)), we only execute the generation of the voxel mips with the hardware mip-mapping as an alternative to the voxel octree structure (Figure 52 (c)).

A similar way to construct a dense voxel representation is described in (McLaren, 2014) (Rauwendaal, 2014). The voxel mip has a similar approach, representing the voxels as cascaded volumes in (McLaren, 2014). By this approach, we can keep the steps to construct voxel structures simpler and make it more performant (Rauwendaal, 2015).

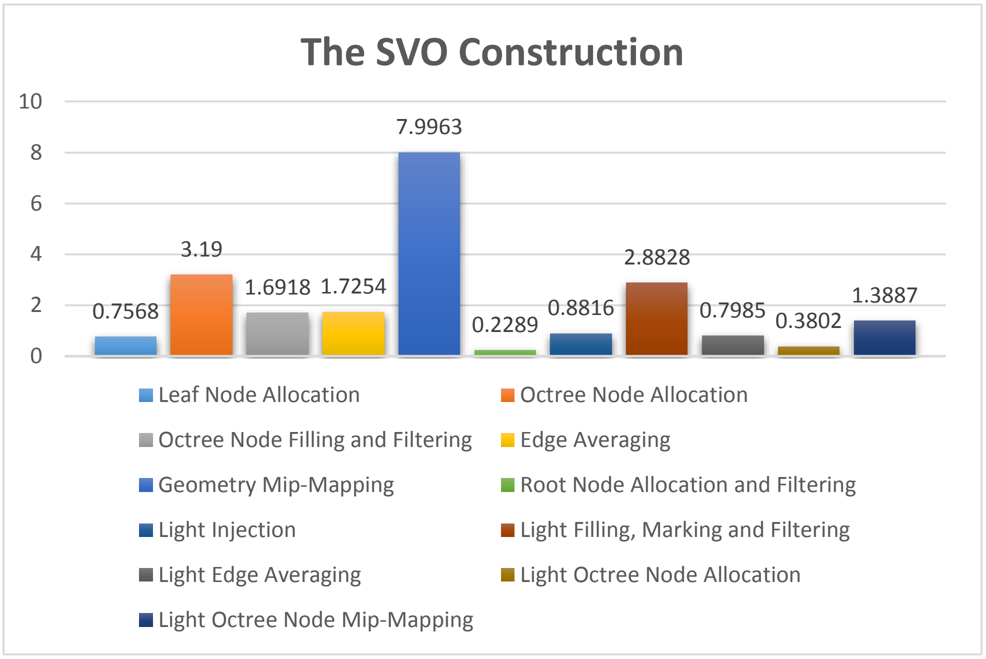


Figure 53. Time Complexity for the SVO Construction

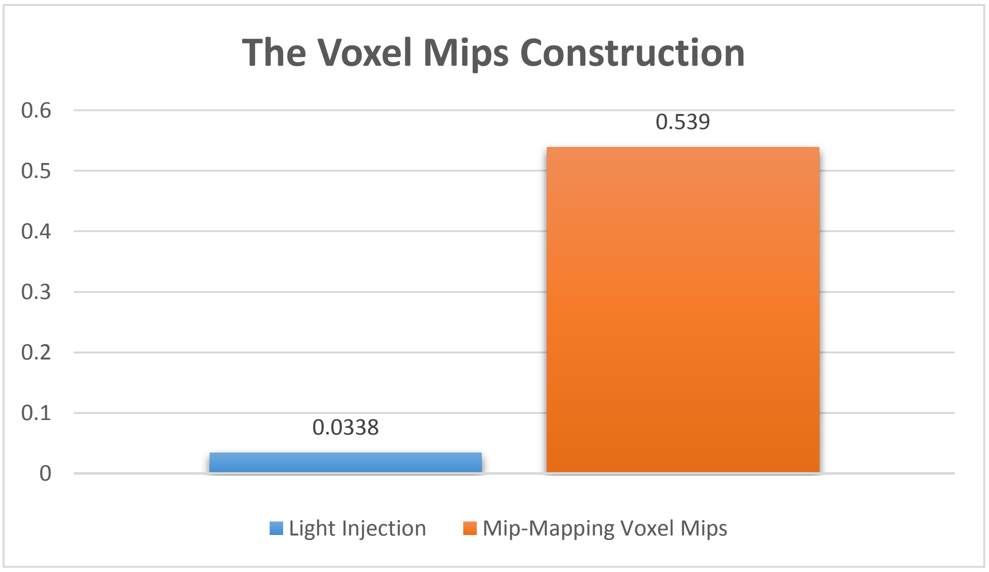
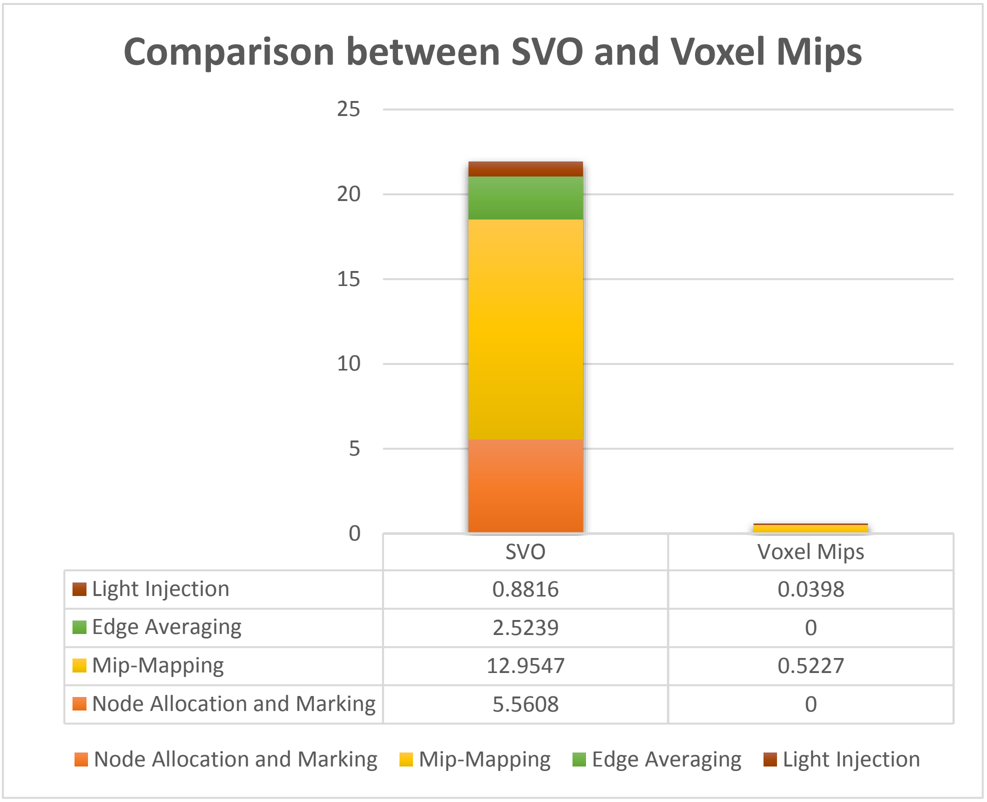


Figure 54. Time Complexity for the Voxel Mips Construction

Basically, the representative Graphics Device Interfaces (GDI), DirectX11 and OpenGL support the functionality to mip-map the voxel mips. The voxel mips is a collection of multiple 3D textures; each has half the resolution of the next. When we trace cones for final rendering to the G-Buffer, we make use of the quad-linear hardware filtering to sample the voxels in world space. Without the voxel mips, we should sample the SVO twice to make use of the custom quad-linear filtering. This incurs twice as many texel sampling operations. The voxel mips makes it possible to sample directly at mip-levels, which reduces overhead caused from fetching texels twice in the SVO (Rauwendaal, 2015).



	SVO	Voxel Mips
🟫 Light Injection	0.8816	0.0398
🟩 Edge Averaging	2.5239	0
🟨 Mip-Mapping	12.9547	0.5227
🟧 Node Allocation and Marking	5.5608	0

Figure 55. The comparison for construction time complexity between the SVO and the Voxel Mips

It is time to analyze the graphs performed in the environment of the Sponza scene that was tested on an NVIDIA Geforce GTX 660 2GB, at 1280×720 and 256^3 resolution. The first graph in Figure 53 profiles all steps in the Sparse Voxel Octree (SVO) construction. The second graph in Figure 54 times the Voxel Mips construction. From these two graphs, we find that the construction of the SVO has more complicated steps and more heavy computational operations than that of the Voxel Mips. The result of constructing SVOs in same environment are similar (Drınovsky, 2013). From here, we can assure that the result is reasonable. The more evident difference in the voxel data construction complexity is observed in Figure 55. With the Voxel Mips, we could remove two steps that costs a total of 8.0847ms: the Edge Averaging and Node Allocation and Marking. For light injection, in the SVO, we need to splat photons from the RSMs into the octree, which requires an octree traversal from the root for each photon that represents each pixel in the RSMs. This is why the light injection stage in the SVO is heavier than in the Voxel Mips. In the mip-mapping, we get further benefits from the Voxel Mips as illustrated in the graph.

Converting from the SVO to the Voxel Mips results in better performance in the voxel data construction, which allows us to voxelize the scene every frame in real-time. However, it requires more memory storage than the SVO. If we can get this amount of performance benefit, it is a good trade−off for the increase in the memory usage. Considering the hardware architecture of the new console generation, we need to adopt a paradigm to increase performance by trading off memory space.

4.2 Interleaved Sampling for the Voxel Cone Tracing

Interleaved sampling is a widely used technique for reducing the computation time of real-time ray tracing algorithms. This scheme involves smoothly blending between regular and irregular sampling, yielding the most efficient compromise (Keller & Heidrich, 2001). The interleaved sampling used in global illumination is practically applied to the Instant Radiosity with Virtual Point Lights (VPLs) (Wald, et al., 2002). The interleaved sampling can be also applied to the VCT for the diffuse indirect illumination as well. Like with the path-tracing, the indirect diffuse method requires the span of multiple cones to approximate the ground-truth reference. Spanning a greater number of cones, guarantees a more realistic rendered image. However, we should find a compromise between the time complexity and the render quality.

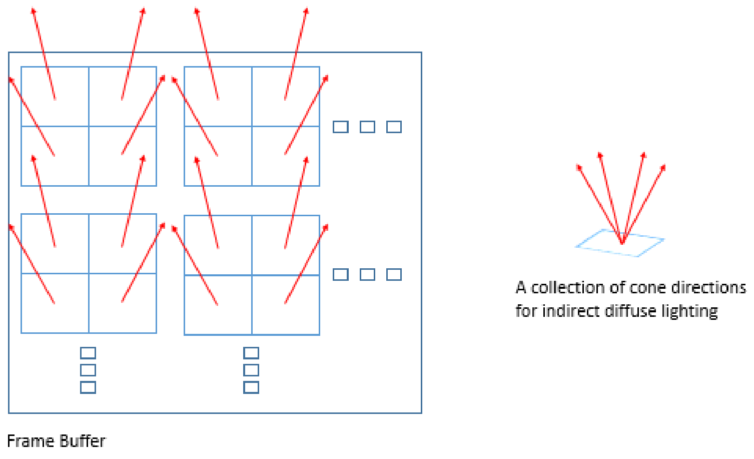

A collection of cone directions for indirect diffuse lighting
Frame Buffer

Figure 56. The description for the interleaved pattern in the voxel cone tracing for indirect diffuse illumination

For the diffuse indirect illumination in the VCT algorithm, we should perform several cone tracings for each pixel to approximate the reference point. However, this is computationally heavy. If we assign only one cone tracing per pixel, the speed increases greatly. Interleaved sampling makes this possible by interleaving different cone directions into the pixels (Figure 56)

In the VCT, interleaved sampling is done in a similar manner to (Wald, et al., 2002) with the VPLs. We interleave sets of $n \times m$ diffuse cone directions in regular patterns. All elements in a set will be rotated randomly to make variation in irregular patterns. Each pixel has an irregular pattern, but each set has a regular pattern, which is exact interleaved sampling. Different from classic interleaved sampling in VPLs, this requires one additional step (the gathering phase), which will be covered in more depth later.

Interleaved sampling is cache inefficient. Each pixel performs a cone tracing in different direction. If we rearrange pixels into collections according to their cone direction, we could trace cones in the GPU in a more cache-friendly manner; it becomes more efficient to access the memory in the GPU. The two−pass rearrangement method is suggested in (Segovia, et al., 2006). It originated from the fact that the naïve approach for the one-pass algorithm is also cache-inefficient when accessing texture memory. To optimize this, it suggests a two-pass algorithm, splitting the original buffer into $p \times q$ blocks to generate the $p \times q$ sub-buffers. The principle is to enhance the data locality for cache coherency. It consists of two step, one for Block Splitting and the other for Block Translation.

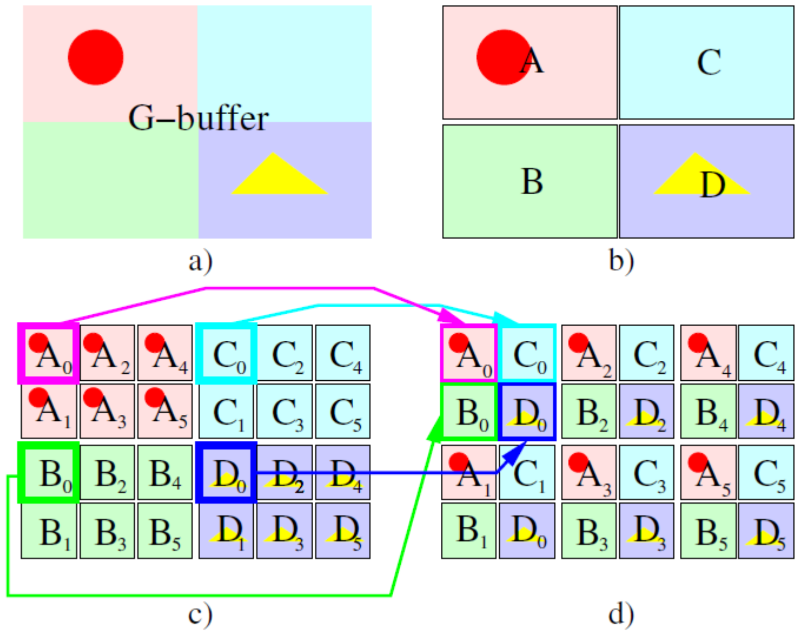


Figure 57. The desired interleaved pattern is 3x2. (a) presents the initial G-buffer. (b) The G-buffer is subdivided in 4 (2x2) blocks. The interleaved pattern size is 3x2. (c) shows the G-buffer after the block splitting. As the interleaved pattern is 3x2, each block has been split in 6 (3x2) sub-blocks. The wanted sub-buffers ($A_iB_iC_iD_i$) are therefore spread across the whole buffer: (d) shows how the sub-buffers ($A_iB_iC_iD_i$) are retrieved by the block translation. (Segovia, et al., 2006)

The Block Splitting step is introduced to improve cache-friendly access in textures. It limits the manipulation of data into small sub-blocks. The initial G-Buffer is divided by $p \times q$ blocks, and each block has $n \times m$ sub-blocks. The Block Translation step is performed to build each of the sub-buffers by the translation of sub-blocks. The Figure 57 describes the algorithm visually in detail. Each step will be look like this (Figure 58).

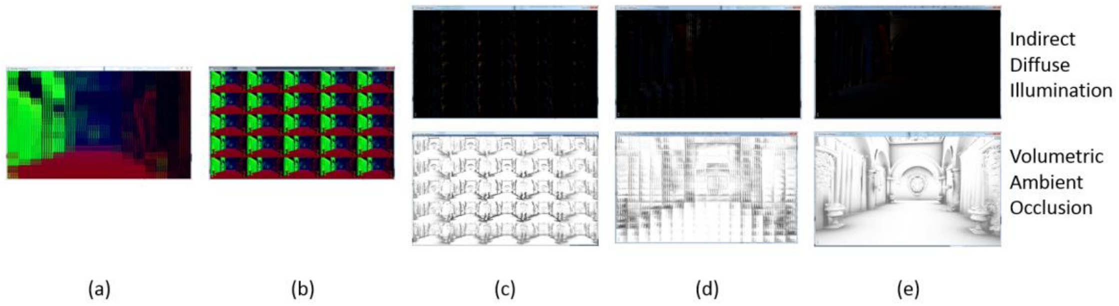


Figure 58. The illustration for splitting G-buffer to sub-buffers and gathering sub-buffers into original buffer. (a) Block Splitting (b) Block Translation (c) Voxel Cone Tracing for each sub-buffer in individual cone direction (d) Inverse Block Translation (e) Inverse Block Splitting (or Block Gathering)

We need to note that the memory access of Block Translation also remains coherent. Like the CPU multi-threading, the GPU operations are basically done by multi-threading, so the same scheme is applied since memory access and cache coherency take a great role in the performance. Compared to the naïve one-pass algorithm, the two-pass method provides a way to rearrange the G-Buffer for more efficient memory access.

Now we are ready to operate the voxel cone tracing on the sub-buffers categorized into the same cone directions. With this rearrangement, we can execute cone tracings without updating

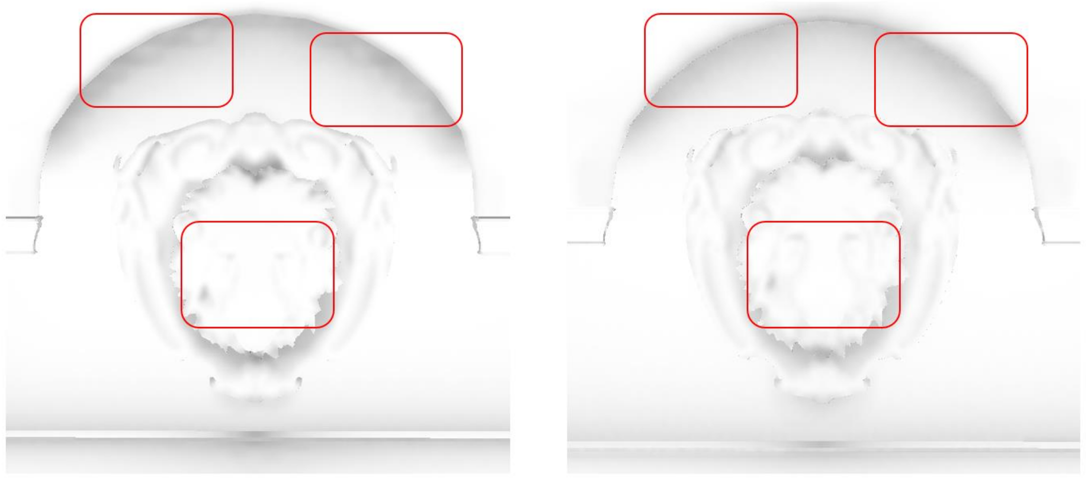


Figure 59. The Comparison between 3x3 interleaved sampling (Left) and 5x5 interleaved sampling (Right)

the constant buffer that stores the parameters for shaders, which increases performance. Most of all, reducing multiple cone tracings to a single cone tracing by interleaved sampling is the biggest contributor to the performance enhancements. As we have said earlier, using more cones, provides a more realistic scene (Figure 59).

The demo is determined to span total 25 cones, so it splits the G-Buffer into a 5x5 interleaved pattern.

After the VCT with the sub-buffers, we need to restore the result to its initial arrangement in the G-buffer. We call this Buffer Gathering. Buffer Gathering shares the same scheme as Buffer Splitting. We just do it in the reverse order. First, we do the reverse Block Translation and as Block Gathering, the reverse order of the Block Splitting.

Due to interleaved sampling, we again face the structured noise artifacts (Figure 60).

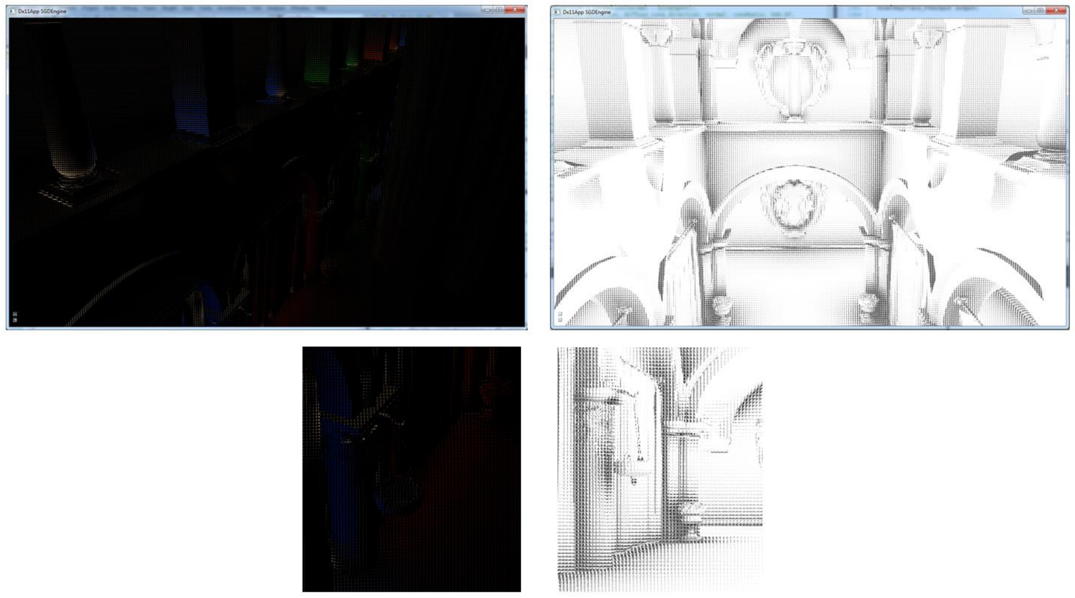


Figure 60. The structured artifacts (Left) Indirect Diffuse Illumination (Right) Volumetric Ambient Occlusion

To reduce these artifacts and provide a smoother transition, the proper blur algorithms should be applied. Between various blur algorithms, the bilateral blur with adaptive sampling is used. We know that the bilateral blur is used to blur the image, preserving the edges with the discontinuity buffer. The adaptive filtering sounds unfamiliar. The adaptive blur is suggested in (Bavoil & Andersson, 2012). It is a hybrid process that samples the inner half of the kernel with a step size of 1 with point sampling, and the outer half of the kernel with a step size of 2 with bilinear filtering (Figure 61).

Before the blurring the result to remove the artifacts, we need the discontinuity buffer to preserve the edges. Two discontinuity thresholds for normals and position are fixed, then a pixel shader reads the G-buffer to decide if the current pixel is discontinuous or not by evaluating an arbitrary measure between the current pixel (x_0, y_0) and its neighbors $(x_0 + 1, y_0)$, $(x_0, y_0 + 1)$ and $(x_0 + 1, y_0 + 1)$. If thresholds for position and normal are exceed, the discontinuity is determined (Figure 63).

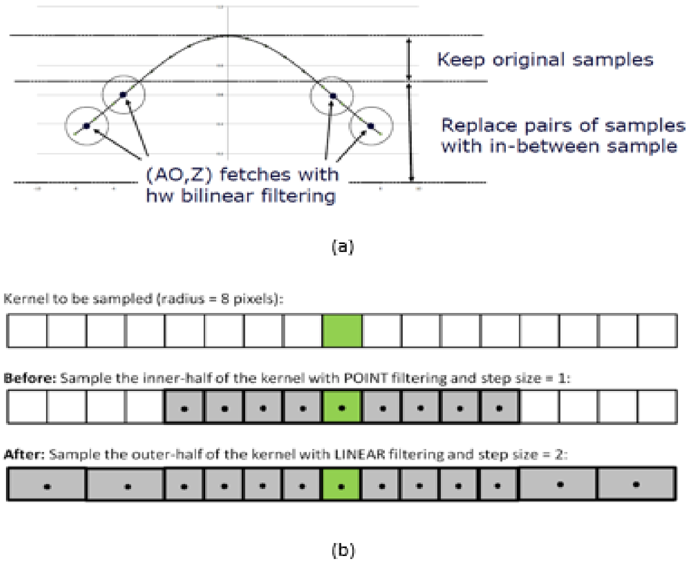

Keep original samples
Replace pairs of samples with in-between sample
(AO,Z) fetches with hw bilinear filtering
(a)
Kernel to be sampled (radius = 8 pixels):
Before: Sample the inner-half of the kernel with POINT filtering and step size = 1:
After: Sample the outer-half of the kernel with LINEAR filtering and step size = 2:
(b)

Figure 61. The illustration of Adaptive Sampling. (a) Gaussian distribution description for adaptive blurring. (b) 1D cross bilateral filtering; inner half for point-sampling with step size 1 and the outer half for linear-sampling with step size 2 (Bavoil & Andersson, 2012)

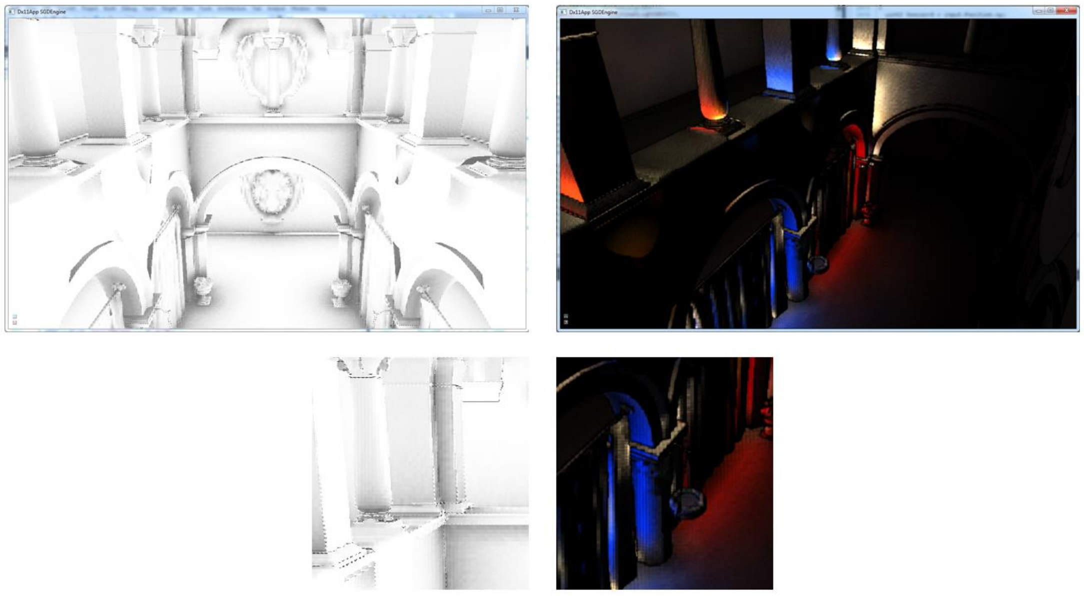


Figure 62. The 5x5 gathering phase for the volumetric AO (Left) and the indirect diffuse illumination (Right)

With the discontinuity buffer, we need one extra step, which differs from the method described in (Segovia, et al., 2006). This stage is decided to be called the 'Gathering Stage'. In this stage, we gather the diffuse indirect illumination for a $n \times m$ block, preserving edges with the

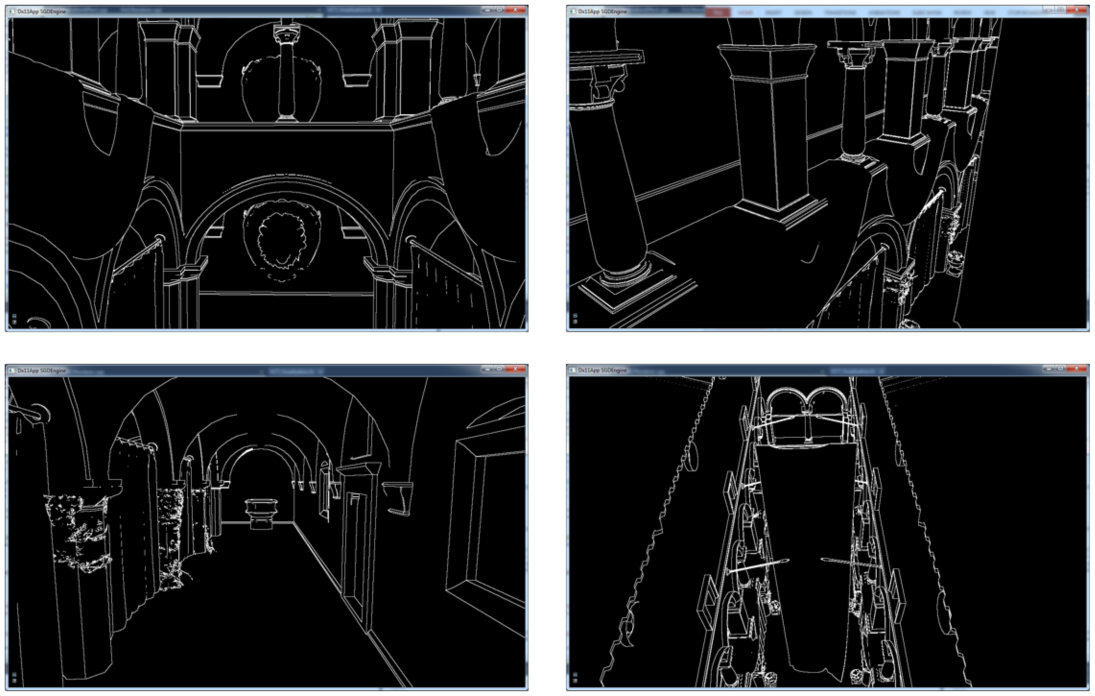


Figure 63. The discontinuity buffer in different camera views

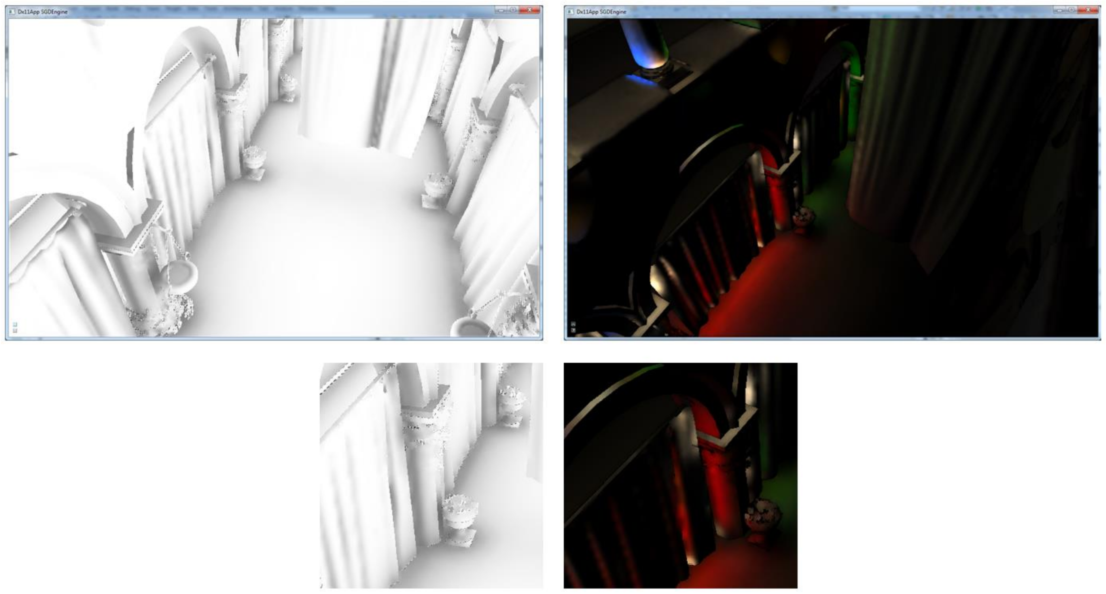


Figure 64. The bilateral Blur with the discontinuity buffer. (Left) Ambient Occlusion, (Right) indirect diffuse illumination

discontinuity buffer (Figure 62). It is now time for a bilateral blur with adaptive sampling. The bilateral blur is basically edge-preserved with the discontinuity buffer (Figure 64). By using interleaved sampling, we could hugely boost the performance of the diffuse indirect illumination, which takes up a majority of the time complexity of the VCT. But it suffers critical artifacts like sinuous waves, which make the scene look really unstable (Figure 65). We could remove these noisy artifacts by combining the techniques known as the reverse reprojection and temporal filtering.

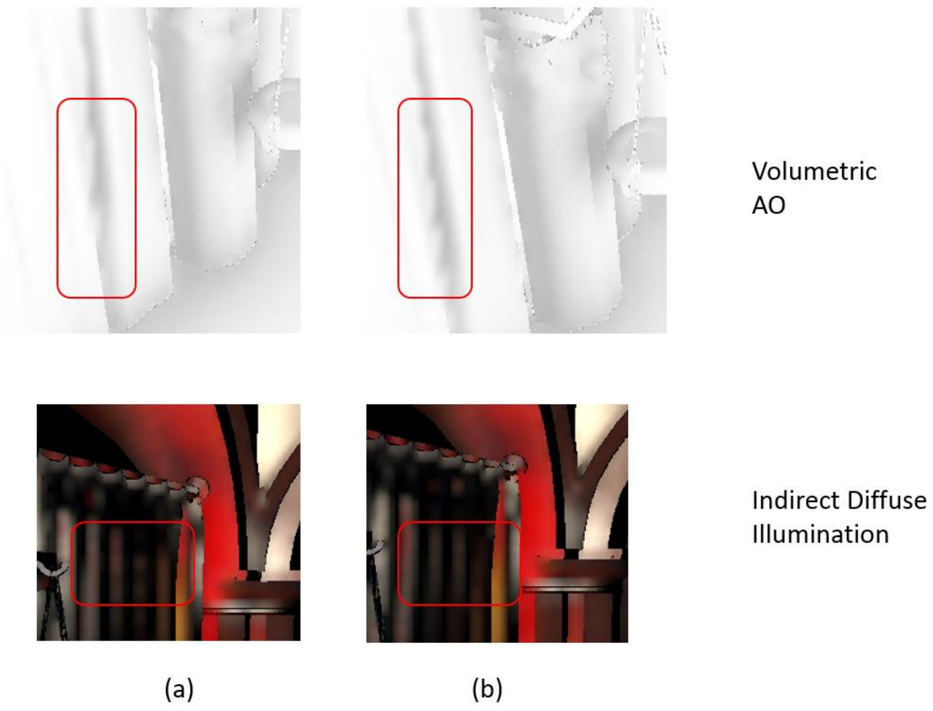


Figure 65. The sinuous wave artifacts whenever camera is moving or rotated; (a) before camera movement (b) after camera movement

4.3 Reverse Reprojection and Temporal Filtering

Reverse reprojection and temporal filtering have been used in the Screen-Space Ambient Occlusion (SSAO) for removing trailing artifacts and ghost artifacts.

The sinuous artifact from the interleaved sampling is caused by the same thing as ghost artifacts − the inconsistent sampling that we have covered in the problem analysis. If we reuse the sampling that was already calculated in the previous frame, we can reduce this artifact in the interleaved sampling of the VCT. The reverse reprojection and temporal filtering are the most effective methods to reduce the artifact.

The reverse reprojection is conceptually simple. We can determine if the current pixel could reuse the value from the history buffer of the previous frame by comparing the current depth and previous depth found in previous frame. We can achieve the following steps (Figure 66);

1) Calculate the world position P_i with (UV_i, Z_i) and $ViewProjectionMatrix^{-1}$
2) Project the world position with Previous$ViewProjectionMatrix$ of previous camera
3) Fetch the previous view depth P_{i-1}
4) If the difference of P_i and P_{i-1} is within the threshold, we reuse the previous one

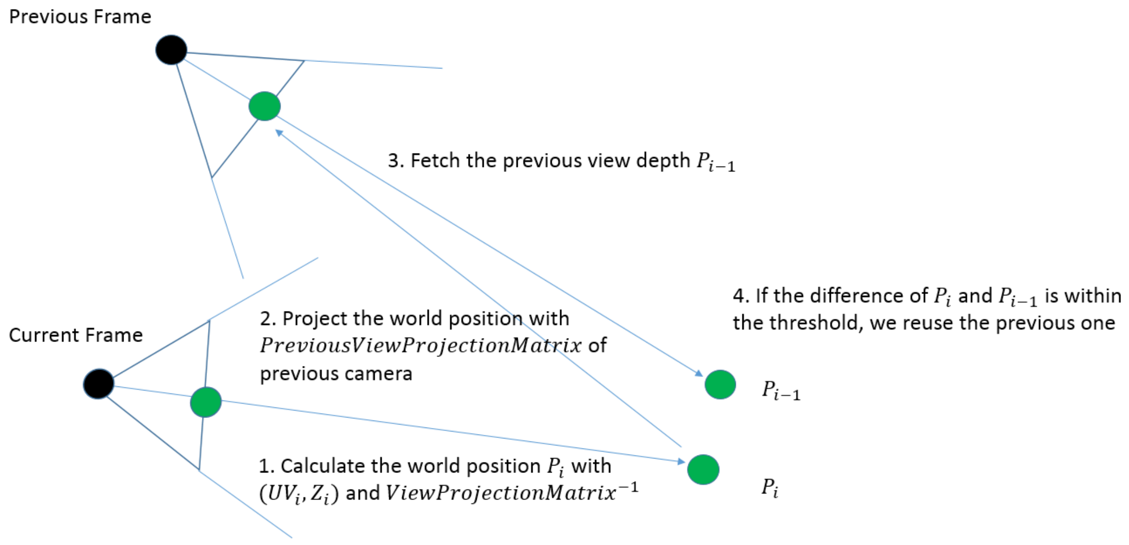

Previous Frame
3. Fetch the previous view depth P_{i-1}
Current Frame
2. Project the world position with $PreviousViewProjectionMatrix$ of previous camera
4. If the difference of P_i and P_{i-1} is within the threshold, we reuse the previous one
P_{i-1}
1. Calculate the world position P_i with (UV_i, Z_i) and $ViewProjectionMatrix^{-1}$
P_i

Figure 66. The illustration of the Reverse Reprojection

However, the reverse reprojection causes same problem in the SSAO from the paper (Bavoil & Andersson, 2012), the trailing artifact and the ghost effect (Figure 67).

To reduce them, temporal filtering needs to be introduced. Temporal filtering is also called 'Temporal Refinement'. These trailing artifacts usually appear when the current changes are not reflected at all. To slowly converge towards the current pixel state, from using the previous stat of the pixel, the pixel will consider applying a small portion of the current state. The formula for the temporal refinement is defined like this:

$$newGI_f(p) = \frac{w_{f-1}(p_{f-1})GI_{f-1}(p_{f-1}) + kGI_f(p)}{w_{f-1}(p) + k}$$

$$w_f(p) = \min(w_{f-1}(p_{f-1}) + k, w_{max})$$

The $newGI_f(p)$ is the result of temporal refinement, after taking the current value $GI_f(p)$ into account. The $w_{f-1}(p)$ is the weight of the previous frame, which − before convergence has been

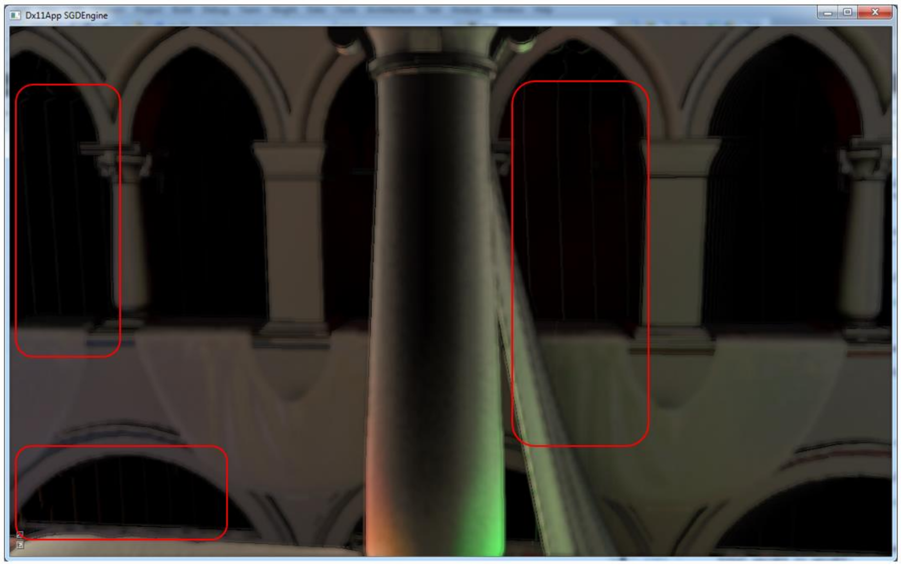


Figure 67. The Trailing artifacts caused by the reverse reprojection

reached and without invalidation − is equal to the number of samples that have already accumulated. The $w_f(p)$ is a weight of the current frame. The value of k is the number of samples contributing to the current solution. The w_{max} is a maximum weight, or maximum number of samples to accumulate for the total frame.

By doing this, we have removed the trailing artifacts caused by the reverse reprojection. Figure 68 describes the stable interleaved sampling of the AO with the reverse repojection and temporal refinement. Finally, we have produced a stable version of the voxel cone tracing algorithm using interleaved sampling to optimize performance.

Before closing this section, compare the results between the raw voxel cone tracing for executing 25 rays per pixel and the interleaved sampling voxel cone tracing in the 5×5 pattern.

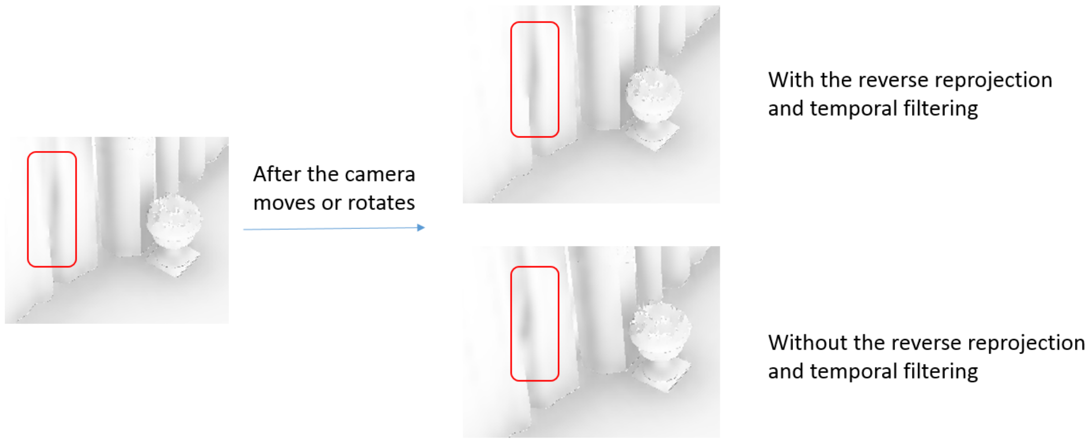

After the camera moves or rotates
With the reverse reprojection and temporal filtering
Without the reverse reprojection and temporal filtering

Figure 68. The comparison between the results, with the reverse reprojection and temporal filtering, and without the reverse reprojection and temporal filtering

Before comparing the performance, we need to know the reason why even the raw voxel cone tracing needs to have the adaptive blur step. Because of the voxelized scene, raw voxel cone tracing can cause the blocky artifacts (Figure 69). To release artifacts, we need the adaptive blur on the raw voxel cone tracing like in interleaved sampling.

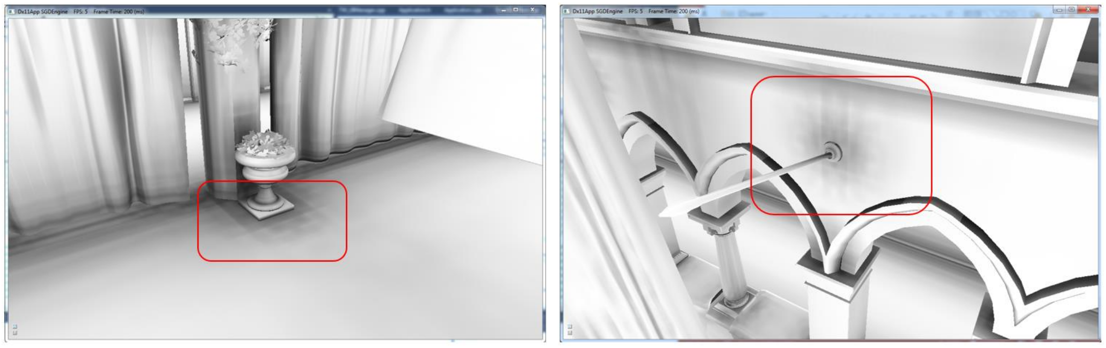


Figure 69. The blocky artifact on the raw voxel cone tracing with 25 cones

The Figure 70 describes the overall Frames per Second (FPS) tested on NVIDIA Geforce 660 GTX 2GB. The interleaved method for the VCT gets closer to real-time FPS compared to the raw voxel cone tracing.

The difference between the raw method and the interleaved method is well-described in Figure 71. The overall intensity for both volumetric AO and diffuse indirect illumination is increased for the comparison. In the difference image (Right), we can observe most of the differences in read. In other words, we could speed up the voxel cone tracing stage by sacrificing

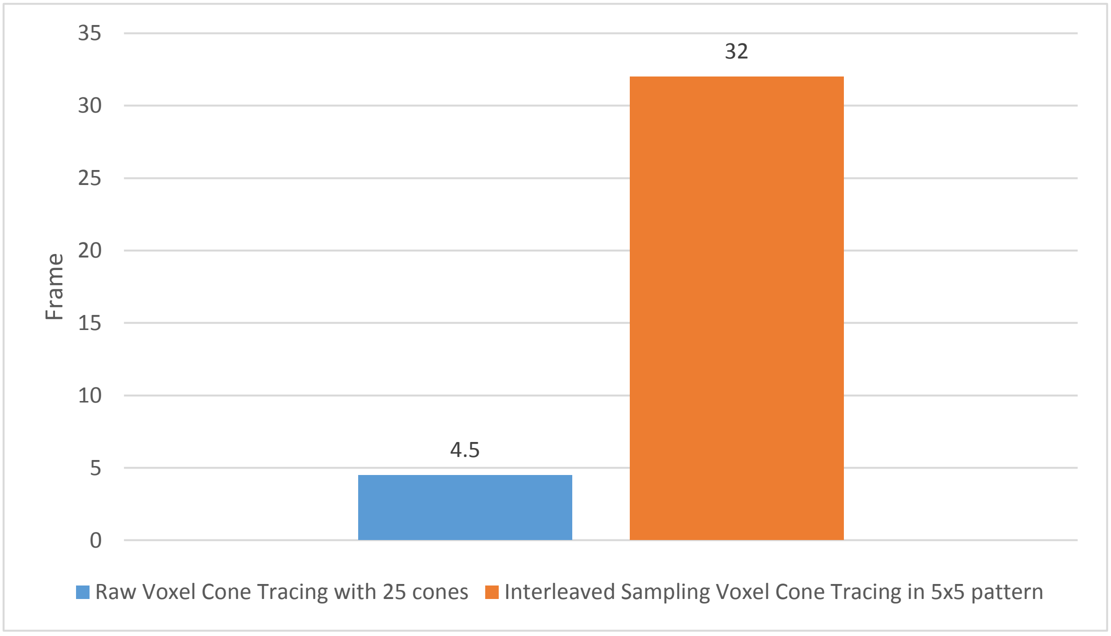


Figure 70. The comparison between the raw voxel cone tracing with 25 cones and the interleaved sampling voxel cone tracing in 5x5 pattern

the precision within 0.1. In the final rendered scene, we can't find any large differences caused by sacrificing 0.1 precision (Figure 72).

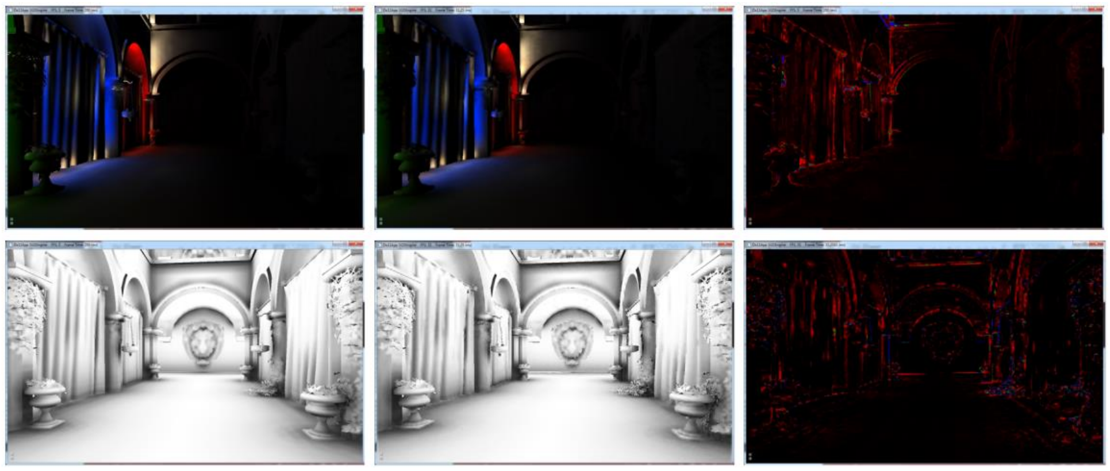


Figure 71. The difference of indirect diffuse illumination and volumetric AO between the raw voxel cone tracing and the interleaved sampling. Raw voxel cone tracing (Left), Interleaved sampling voxel cone tracing (Middle) and the difference (Right). For difference image, the Sum of absolute differences (SAD) is used, and the red is ranging from 0.0 to 0.1 and green is from 0.1 to 0.2 and blue represents over 0.2.

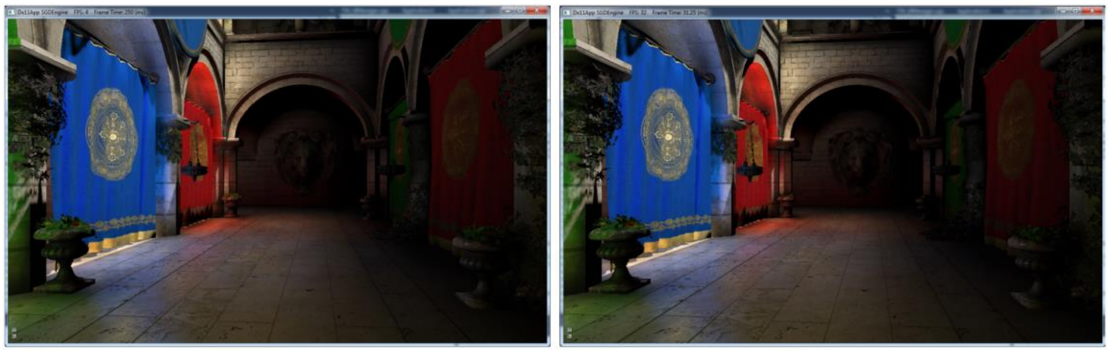


Figure 72. The Final images - the raw voxel cone tracing (Left) and the interleaved sampling voxel cone tracing (Right)

4.4 Results

All timings have been done on a Kepler-based NVIDIA GTX 660. All tests have also done in conditions where the Crytek Sponza is dynamically voxelized for each frame and the voxelization is done in the 256^3 voxel mips.

The Figure 73 describes the voxelization phase containing the light injection stage. Compared to Reflective Shadow Maps (RSMs) generation and Voxelization, the other two stages − Mip-Mapping Voxel Mips and Light Injection − take as really small portion of the timing. After the Voxelization phase, the Voxel Cone Tracing Phase is executed. All relevant timings are well-

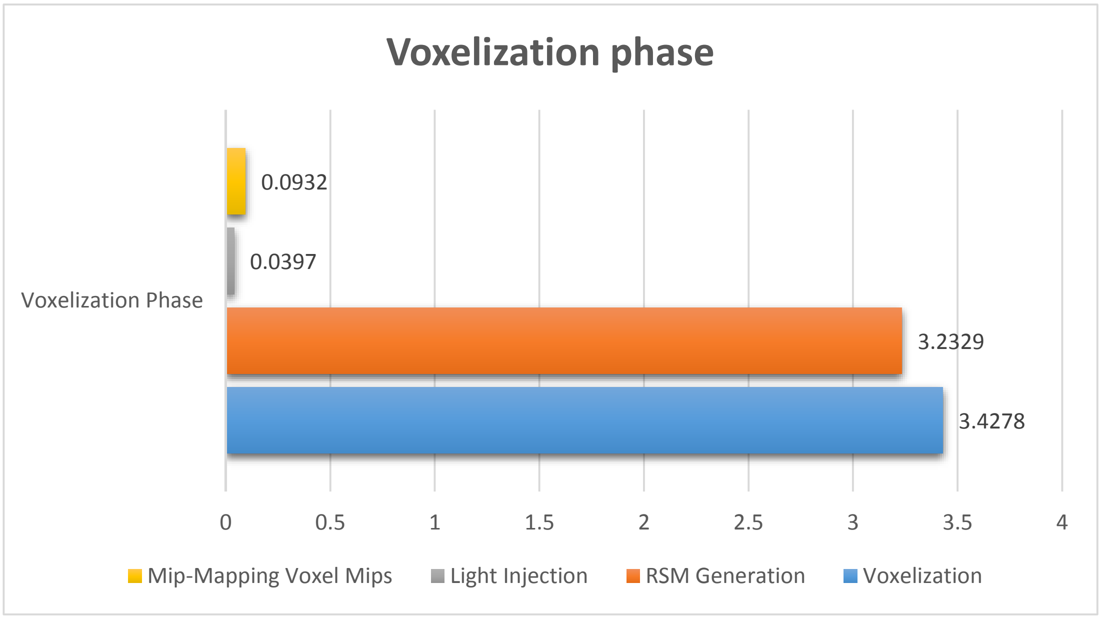


Figure 73. All timings for the Voxelization Phase

illustrated in Figure 74. Including specular cone tracing, all voxel cone tracing calculations are hugely decreased by implementing interleaved sampling.

The overall timing for the rendering pipeline in the demo is described in Figure 75. Considering the timing of G-Buffer generation, the voxelization phase (including the RSM generation) is adaptable for real-time rendering. With the interleaved sampling, we achieve compatible timing for diffuse voxel cone tracing. In the rendering engine, it runs on average over 30 fps. Considering the test environment used a NVIDIA GTX 660, and that the most current NVIDIA graphics cards

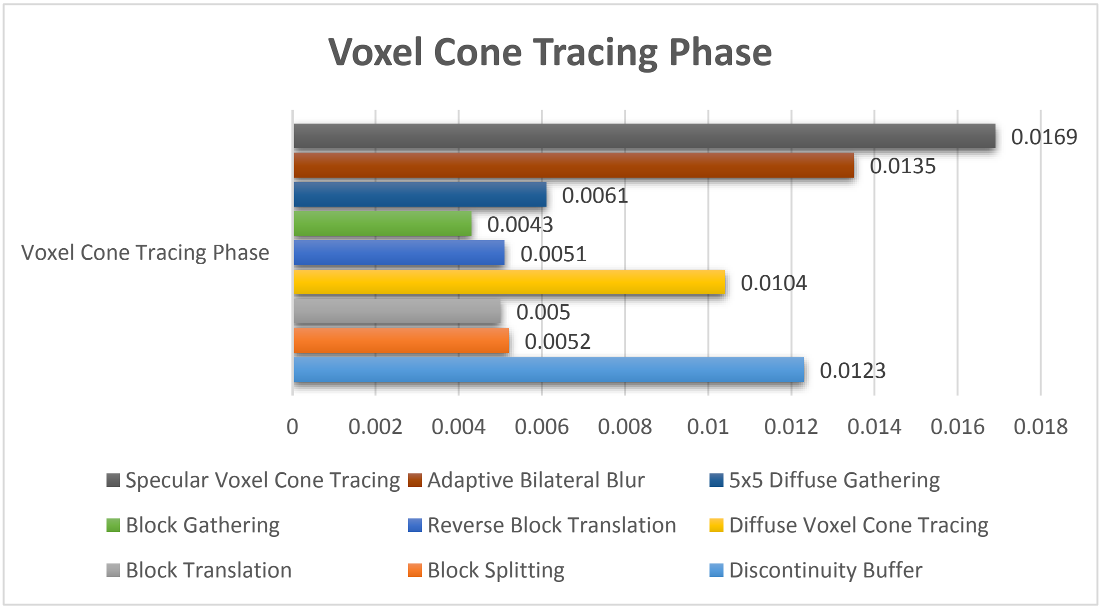


Figure 74. Total timings for the Voxel Cone Tracing Phase

are in the 900 series, it is enough to consider the voxel cone tracing as a solution to global illumination.

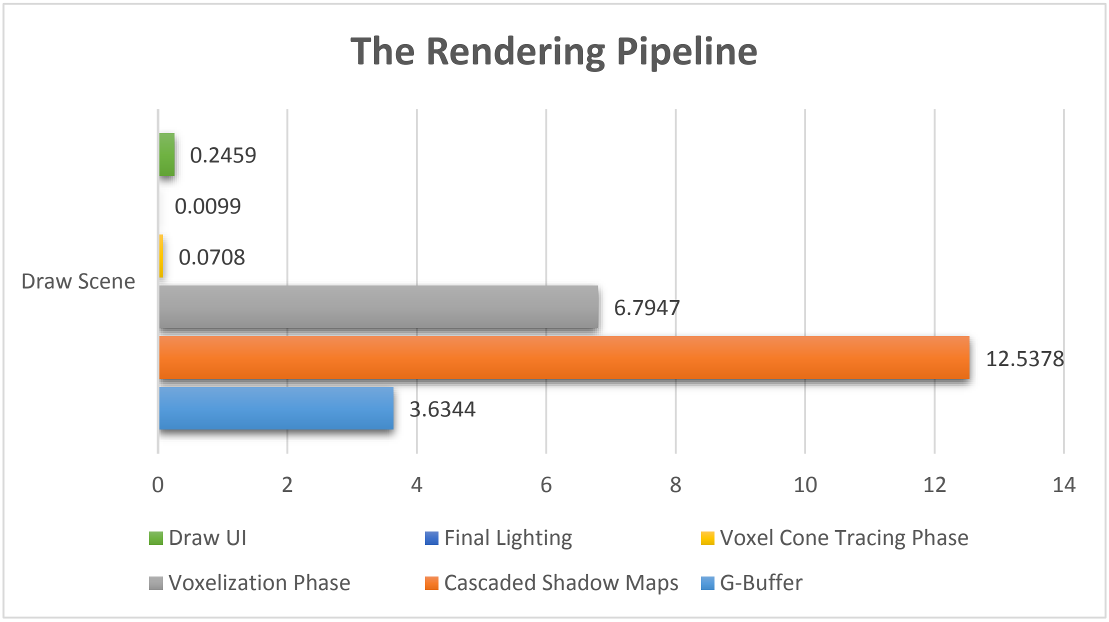


Figure 75. All timings for the Rendering Pipeline

Figure 76 and Figure 77 describes the differences between the voxel global illumination and the direct illumination. The impact of voxel global illumination is big in terms of visual appearance. From these images, we can also find the indirect specular illumination that usually is referenced by the reflectance. In the Figure 78, the light bleeding is well-described. The light bleeding is one of the side-effects of the voxel cone tracing. It is caused by the thin wall. In materials like fabric, it is natural to generate some light bleeding. With the voxel cone tracing, we could even make an effect out of the naturally−occurring light bleeding.

We can understand how the complete global illumination is composited by indirect diffuse and specular illuminations, as well as volumetric ambient occlusion in Figure 79. The voxel cone tracing algorithm can represent physically−based shading by adjusting the specular cone size. By narrowing the angle, we can make shiny, metallic shading. By increasing the angle, a material will appear less shiny. Figure 80 describes the different results produced depending on different specular angle sizes.

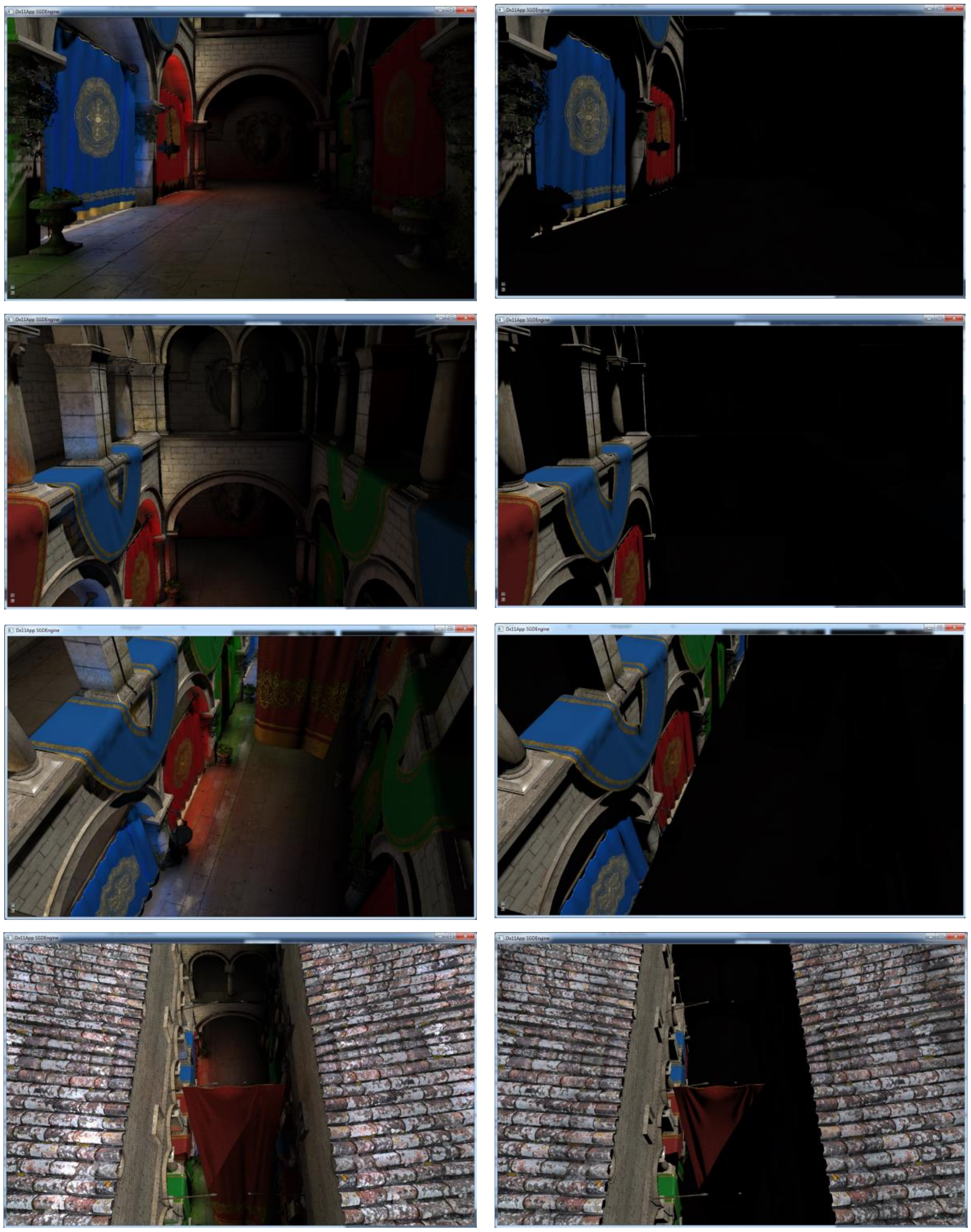


Figure 76. The comparison between the voxel global illumination (Left) and the direct illumination (Right)

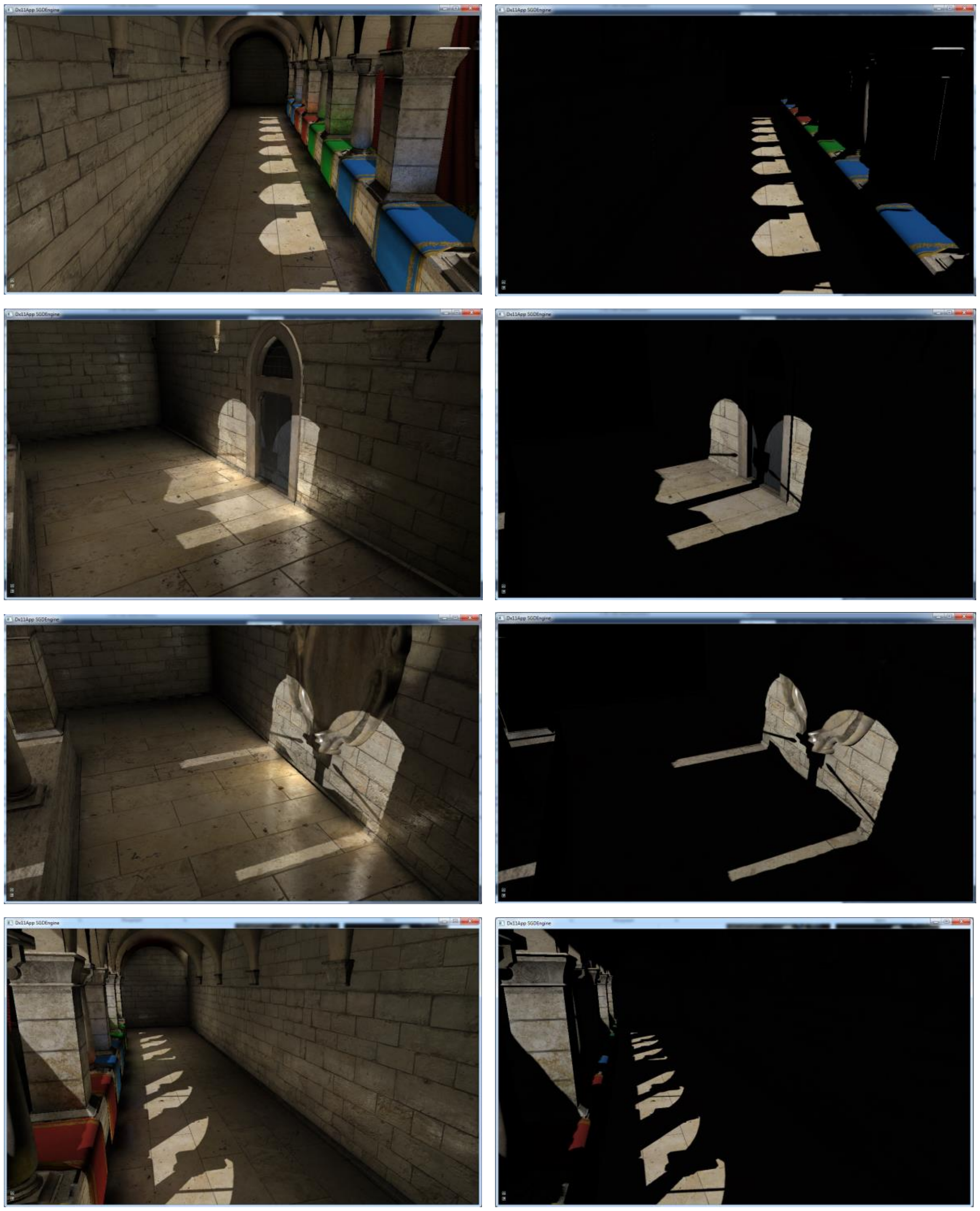


Figure 77. Another comparison between the voxel global illumination (Left) and the direct illumination (Right)

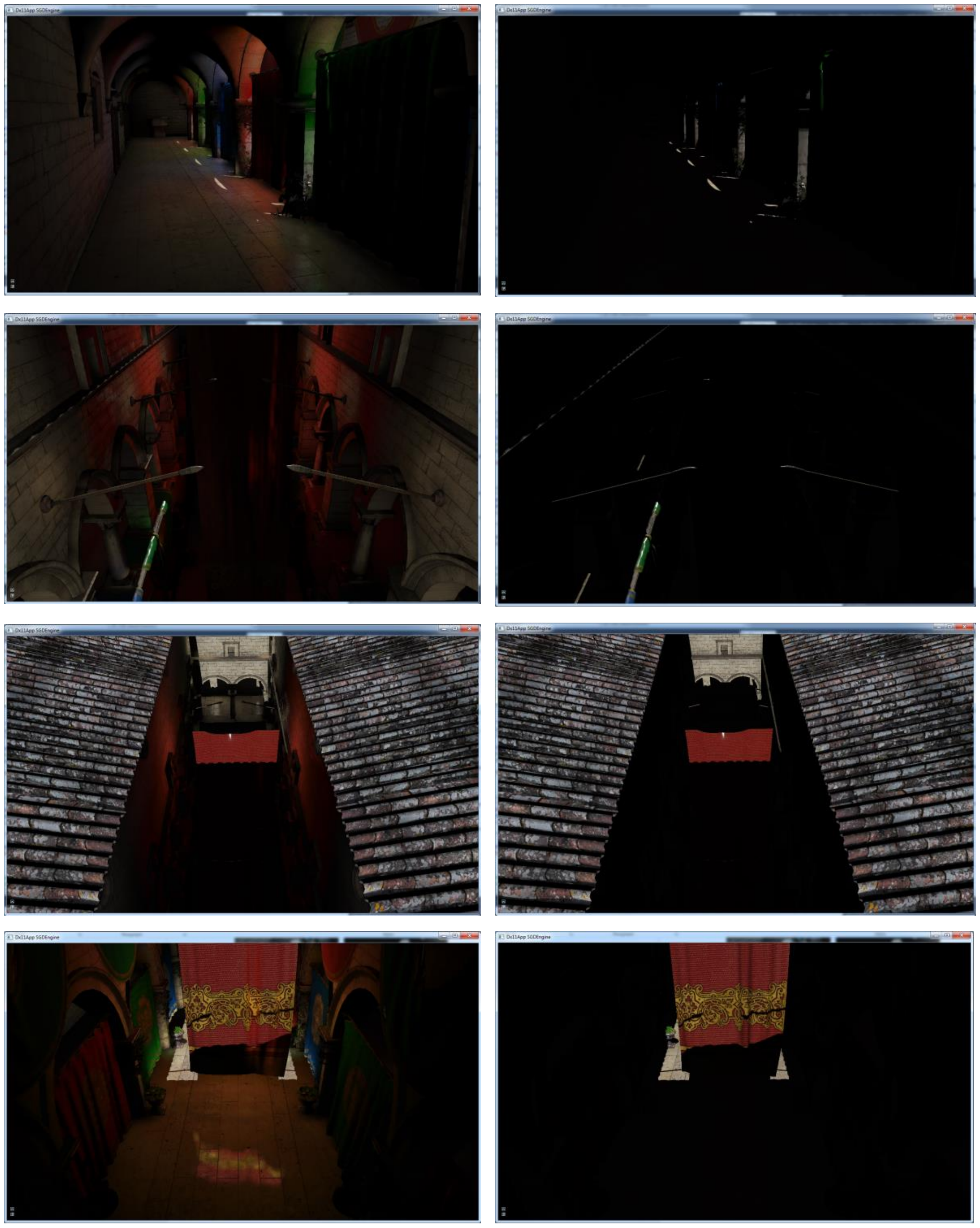


Figure 78. The illustration for the light bleeding, the voxel global illumination (Left) and the direct illumination (Right)

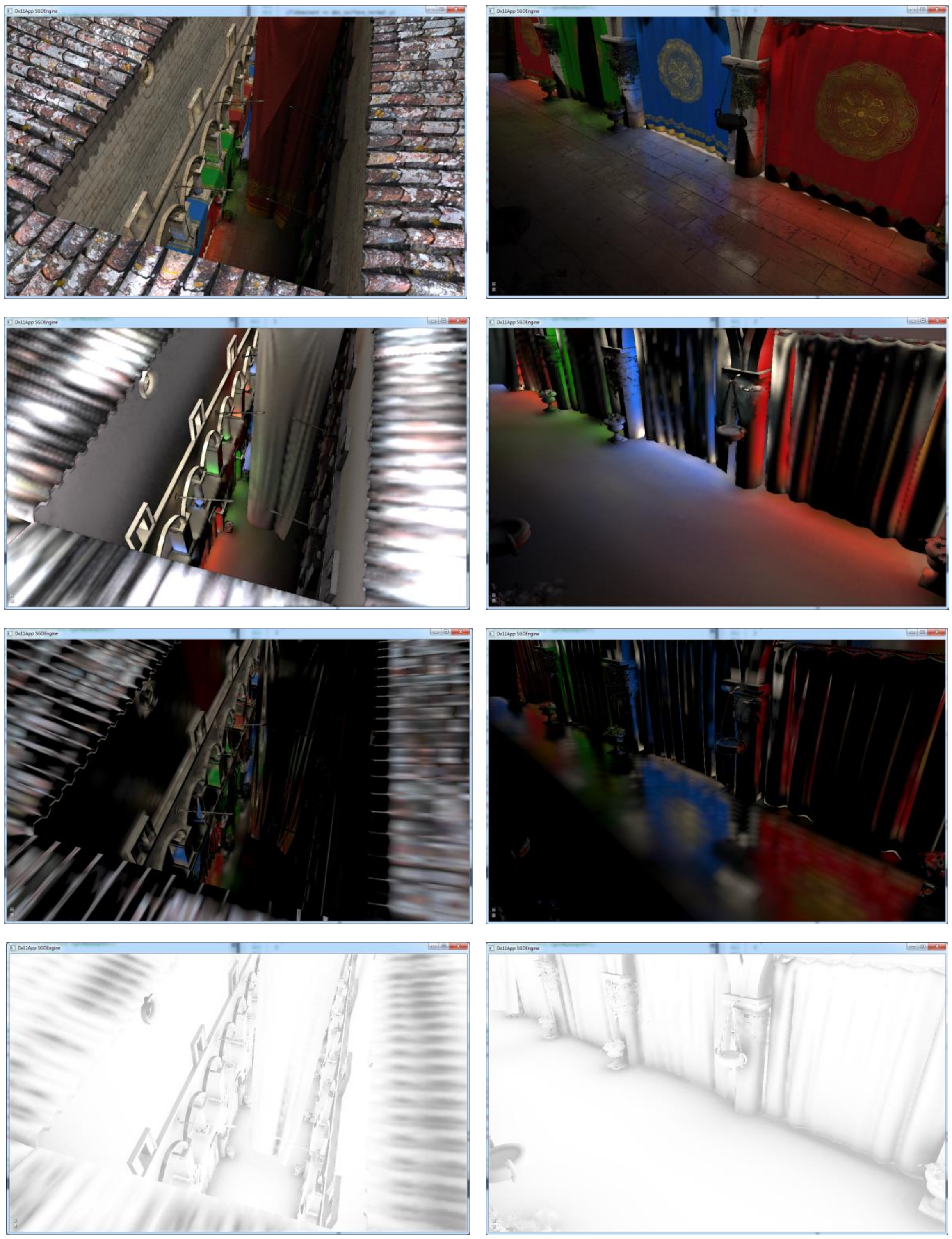


Figure 79. The illustration of the global illumination - the voxel global illumination (Top), the indirect diffuse illumination (Middle Top), indirect specular illumination (Middle Bottom), volumetric AO (Bottom)

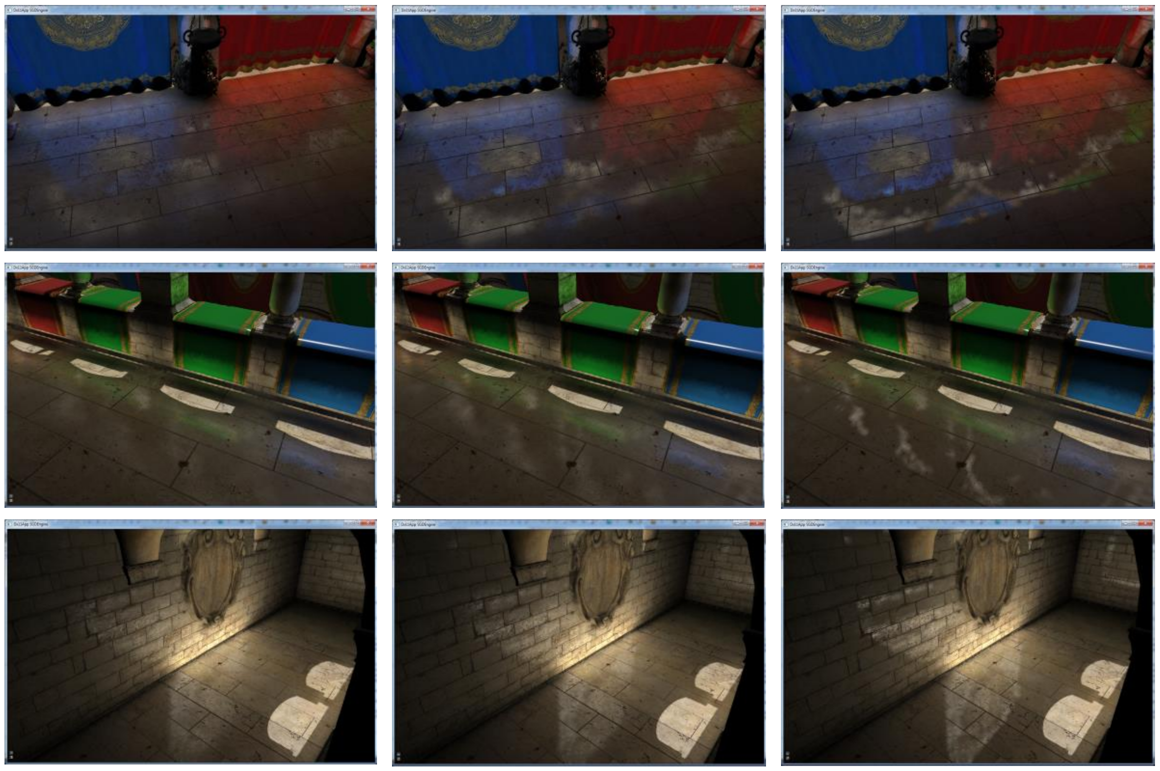


Figure 80. The illustration for various specular cone angles - 9°*(Left),* 5°*(Middle) and* 1°*(Right)*

5. Conclusion and Future Work

In this paper, we present many overall global illumination algorithms, as well as a new contribution to voxel cone tracing for purpose of real-time rendering. In the past few years, the rendering quality for games has been rapidly increasing to satisfy users. With those expectations, the new console came out, but the hardware architecture could not fulfill consumer expectations. The only hope for enhancing graphics quality is in utilizing large amounts of available memory space – about 8GB in modern consoles. This means that for the new generation, we need to focus on memory space for optimization rather than computational time. Voxel cone tracing can be one of the solutions to global illumination which will add a new level of realism to the game world. The original voxel cone tracing algorithm tried to sacrifice the computational time to save on the memory space, which violates the paradigm of the new console generation. By using more memory space, suggest using the new voxel cone tracing method to secure sufficient timing for real-time rendering, which was the main reason it was dropped from Unreal Engine 4. The resultant performance and graphics quality is comparable to more traditional implementations of global illumination like the Precomputed Radiance Transfer (PRT). It is the fact that the real-time global illumination techniques like voxel cone tracing and Light Propagation Volumes (LPV) are unable to reach the quality of the PRT. But, the real-time algorithms have their own advantages in accelerating the production pipeline for artists by allowing previews of their work in real-time. The voxel cone tracing could achieve indirect specular illumination in a high frequency range, which is unable to be done via LPVs.

There are two ways to improve the quality of the voxel cone tracing algorithms: to compress more data into a voxel efficiently and to enlarge the resolution for the voxel mips. To compress efficiently, we can consider using Spherical Harmonics (SH) to store the radiance (McLaren, 2014). From the SH, we can describe the directional radiance, which enhances the overall quality of the voxel cone tracing. To increase the voxel resolution, the clip map suggested in (Panteleev, 2014) is the solution.

Currently, to produce global illumination in real-time, voxel global illumination is being widely researched. As announced by NVIDIA, the future GPU architecture will evolve to process lots of voxels efficiently, like in the NVIDIA GTX 980. We can expect a positive future for voxel global illumination.

References

(Bavoil & Andersson, 2012) Bavoil, L. & Andersson, J., 2012. Stable SSAO in Battlefield 3 with Selective Temporal Filtering.

(Crassin & Green, 2012) Crassin, C. & Green, S., 2012. Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer. In: P. Cozzi & C. Riccio, eds. *OpenGL Insights.* s.l.:CRC Press, pp. 303-319.

(Crassin, et al., 2011) Crassin, C. et al., 2011. Interactive Indirect Illumination Using Voxel Cone Tracing.

(Dimitrov, 2007) Dimitrov, R., 2007. Cascaded Shadow Maps.

(Drınovsky, 2013) Drinovsky, T., 2013. Voxel Cone Tracing for Indirect Illumination.

(Fournier, 1992) Fournier, A., 1992. Normal Distribution Functions and Multiple Surfaces.

(Green, 2003) Green, R., 2003. Spherical Harmonics Lighting: The Gritty Details.

(Greger, et al., 1998) Greger, G., Shirley, P., Hubbard, P. M. & Greenberg, D. P., 1998. The Irradiance Volume.

(Han, et al., 2007) Han, C., Sun, B., Ramamoorthi, R. & Grinspun, E., 2007. Frequency Domain Normal Map Filter.

(Hasselgren, et al., 2005) Hasselgren, J., Akenine-Möller, T. & Ohlsson, L., 2005. Conservative Rasterization. In: *GPU Gems2.* s.l.:NVIDIA.

(Hoffman, 2012) Hoffman, N., 2012. Background: Physics and Math of Shading.

(Hsu & Kakade, 2012) Hsu, D. & Kakade, S. M., 2012. Learning mixtures of spherical Gaussians: moment methods and spectral decompositions.

(JiapingWang, et al., 2009) JiapingWang, et al., 2009. All-Frequency Rendering of Dynamic, Spatially-Varying Reflectance.

(Kaplanyan, 2009) Kaplanyan, A., 2009. Light Propagation Volumes in CryEngine 3.

(Kaplanyan & Dachsbacher, 2010) Kaplanyan, A. & Dachsbacher, C., 2010. Cascaded Light Propagation Volumes for Real-Time Indirect Illumination.

(Keller, 1997) Keller, A., 1997. Instant Radiosity.

(Keller & Heidrich, 2001) Keller, A. & Heidrich, W., 2001. Interleaved Sampling.

(Laine & Karras, 2010) Laine, S. & Karras, T., 2010. Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation.

(Mattausch, et al., 2014) Mattausch, O., Scherzer, D. & Wimmer, M., 2014. Temporal Screen-Space Ambient Occlusion.

(McLaren, 2014) McLaren, J., 2014. Cascaded Voxel Cone Tracing in The Tomorrow Children. In: s.l.:s.n.

(Michael & Shirley, 2000) Michael, A., Premoze, S. & Shirley, P., 2000. An Anisotropic Phong BRDF Model. *Journal of Graphics Tools, vol. 5, no. 2,* p. 22~32.

(Michael, et al., 2000) Michael, A. & Shirley, P., 2000. An Anisotropic Phong Light Reflection Model.

(Mittring, 2012) Mittring, M., 2012. The Technology Behind the "Unreal Engine 4 Elemental demo". In: s.l.:s.n.

(Nehab, et al., 2007) Nehab, D. et al., 2007. Accelerating Real-Time Shading with Reverse Reprojection Caching.

(Nowrouzezahrai, et al., 2012)Nowrouzezahrai, D., Simari, P. & Fiume, E., 2012. Sparse Zonal Harmonic Factorization for Efficient SH Rotation.

(Oat, 2004) Oat, C., 2004. Adding Spherical Harmonic Lighting to the Sushi Engine.

(Panteleev, 2014) Panteleev, A., 2014. Practical Real-Time Voxel-Based Global Illumination for Current GPUs. In: s.l.:s.n.

(Ramamoorthi & Hanrahan, 2001) Ramamoorthi, R. & Hanrahan, P., 2001. An Efficient Representation for Irradiance Environment Maps.

(Rauwendaal, 2014) Rauwendaal, R., 2014. Voxel Based Indirect Illumination using Spherical Harmonics.

(Rauwendaal, 2015) Rauwendaal, R., 2015. *Construction for the Sparse Voxel Octree and Voxel Mips* [Interview with Email] http://rauwendaal.net. (25 Jan 2015).

(Ritschel, et al., 2009) Ritschel, T., Grosch, T. & Seidel, H.-P., 2009. Approximating Dynamic Global Illumination in Image Space.

(Scherzer, et al., 2007) Scherzer, D., Jeschke, S. & Wimmer, M., 2007. Pixel-Correct Shadow Maps with Temporal Reprojection and Shadow Test Confidence.

(Schönefeld, 2005) Schönefeld, V., 2005. Spherical Harmonics.

(Segovia, et al., 2006) Segovia, B., Iehl, J. C., Mitanchey, R. & Péroche, B., 2006. Non-interleaved Deferred Shading of Interleaved Sample Patterns.

(Sloan, 2013) Sloan, P., 2013. Efficient Spherical Harmonic Evaluation.

(Sloan, et al., 2003) Sloan, P.-P., Hall, J., Hart, J. & Snyder, J., 2003. Clustered principal components for precomputed radiance transfer.

(Sloan, et al., 2002) Sloan, P.-P., Kautz, J. & Snyder, J., 2002. Precomputed Radiance Transfer for Real-Time Rendering.

(Sloan, 2008) Sloan, P. P., 2008. Stupid Spherical Harmonics (SH) Tricks.

(Slomp, et al., 2006) Slomp, M. P. B., Oliveira, M. M. & Patrício, D. I., 2006. An Gentle Introduction to Precomputed Radiance Transfer.

(Tatarchuk, 2005) Tatarchuk, N., 2005. Irradiance Volumes for Games.

(Toksvig, 2004) Toksvig, M., 2004. Mipmapping Normal Maps.

(Tsai & Shih, 2006) Tsai, Y.-T. & Shih, Z.-C., 2006. All-Frequency Precomputed Radiance Transfer using Spherical Radial Basis Functions and Clustered Tensor Approximation.

(Valient, 2007) Valient, M., 2007. Stable Rendering of Cascaded Shadow Maps. In: *Shader X5.* s.l.:Charles River Media.

(Wald, et al., 2002) Wald, I. et al., 2002. Interactive Global Illumination using Fast Ray Tracing.

(Ward, 1992) Ward, G. J., 1992. Measuring and Modeling Anisotropic Reflection.

(Xu, et al., 2013) Xu, K. et al., 2013. Anisotropic Spherical Gaussians.

(Zhang, et al., 2007) Zhang, F., Sun, H. & Nyman, O., 2007. Parllel-Split Shadow Maps on Programmable GPUs. In: *GPU Gems 3.* s.l.:NVIDA, p. 203~235.