Volumetric Lighting
&
Environment Mapping

GRADUATE THESIS

Author: Ioannis Moschos
Supervisor: Georgios Papaioannou
September 2017
Department of Informatics
Athens University of Economics & Business
Greece

Abstract

The accurate simulation of the various effects of light is considered one of the most important aspects of real-time photo-realistic rendering. This is due to the fact that, in 3D graphics, the generation of physically accurate results depends, to a large extent, on the correct approximation of light's transport and its interaction with matter. In this thesis, we investigate the influence of the application of popular scientific and mathematical concepts to the implementation of the well-known lighting techniques of volumetric lighting and environment mapping. The evaluation of this integration is based on the achieved equilibrium between performance and quality, which corresponds to the ability of performing the above-mentioned techniques in real time while generating results to a satisfactory level of realism.

Περίληψη στα Ελληνικά

Η ακριβής προσομοίωση των διάφορων εφέ του φωτός θεωρείται μία από τις πιο σημαντικές πτυχές της φωτορεαλιστικής απεικόνισης σε πραγματικό χρόνο. Αυτό οφείλεται στο γεγονός ότι, στο κλάδο των γραφικών, η δημιουργία αποτελεσμάτων με φυσική ακρίβεια εξαρτάται σε μεγάλο βαθμό από τη σωστή προσέγγιση της κίνησης του φωτός αλλά και της αλληλεπίδρασής του με την ύλη. Σε αυτή τη διατριβή, διερευνάμε την επίδραση που μπορεί να έχει η εφαρμογή δημοφιλών επιστημονικών και μαθηματικών εννοιών στην υλοποίηση των γνωστών τεχνικών ογκομετρικού φωτισμού (volumetric lighting) και χαρτογράφησης περιβάλλοντος (environment mapping). Η αξιολόγηση αυτής της ενσωμάτωσης βασίζεται στην επιτευχθείσα ισορροπία μεταξύ απόδοσης και ποιότητας, η οποία αντιστοιχεί στη δυνατότητα εκτέλεσης των προαναφερθείσων τεχνικών σε πραγματικό χρόνο με την παράλληλη παραγωγή αποτελεμάτων σε ένα ικανοποιητικό επίπεδο ρεαλισμού.

Acknowledgements

First and foremost, I wish to express my sincere gratitude to my supervisor, Assistant Professor Georgios Papaioannou from Athens University of Economics and Business, for his continuous guidance and support throughout the completion of this thesis.

I am also very grateful to PhD candidate Konstantinos Vardis for the crucial technical advice and knowledge he provided me.

Furthermore, I would like to thank all the members of the AUEB Computer Graphics Group, as their research activities pave the way for newcomers such as myself, to study and experiment in the amazing field of computer graphics.

Last but not least, I would like to thank my family: my parents and my brother, for their unconditional support throughout my studies and my life in general.

Contents

Chapter 1

Introduction

1.1 XEngine

The XEngine is a C++ scene graph-based deferred rendering API that was developed as a research platform for testing various graphics techniques and effects using OpenGL. This engine contains a plethora of real-time graphic algorithms that were designed by the AUEB Graphics Group for the purpose of scientific experimentation. It has also been utilized for the provision of precise graphical material for research papers and theses that are published on the group's website. For an earlier version of XEngine's API, including a range of additional software released by the group, readers are advised to visit the web-page's download section, where a free copy of the engine is provided under the terms of MIT license (MIT). Finally, for more information and a thorough user guide we recommend reading the engine's documentation and API reference [PV13].

In this thesis, our goal is to replicate two real world physical phenomena in a 3-dimensional virtual environment. There are multiple available computer applications able to accommodate the development of such effects. Nevertheless, without access to their software's source code, we would be forced to compromise with various workarounds, that would probably lead to an inadequate solution of the problem. Otherwise, we could create a custom graphics engine for the sole purpose of hosting those effects. However, the complexity of the production of such a program would ultimately overshadow this thesis' main objective.

To avoid the aforementioned pitfalls, we have selected the XEngine as the development platform for our implementation. First of all, by given permission to tweak the engine's internal components without disrupting its initial work-flow, we manage to perform the desirable execution of both effects without any trade-offs. Even more, as mentioned

above, this engine already contains various techniques and algorithms (discussed in Sections 3.2.1 and 3.3.1) that could be utilized for the facilitation of the effects' construction. A final advantage that this engine provides, is that since it is regularly extended and optimized due to being used for various tests and scientific publications, it ensures the continuity, utilization and future improvement of this implementation.

1.2 Volumetric Lighting post-process Effect

The volumetric lighting post-process effect is an approximation of the real-world light scattering physical phenomenon. This phenomenon is responsible for various spectacles, one of which is the appearance of light shafts in a medium filled with dispersed particles. Due to their magnificence, those light beams are sought out (or generated by using different tricks) for the purpose of producing more aesthetically pleasing results. In the world of 3D graphics, the utilization of this effect greatly enhances the elegance and realism of the virtual scene. This technique is used in many fields including 3D modeling, 3D gaming and animated films (for some examples of light scattering, see Figure 1.1).

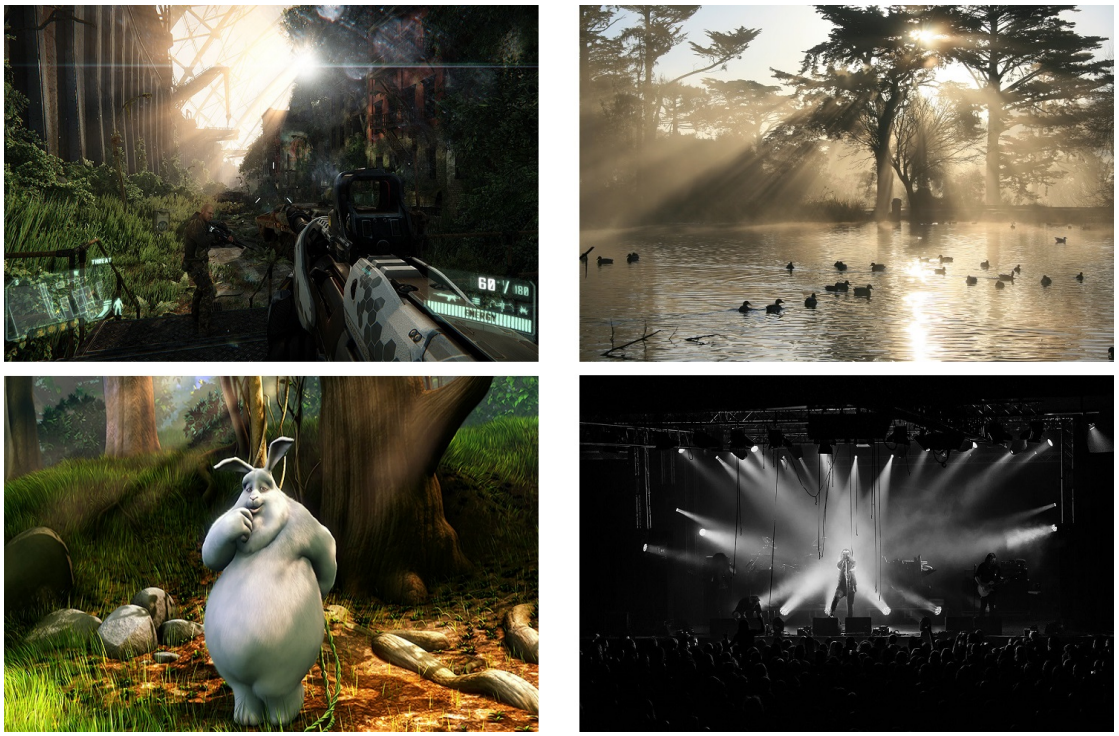

FIGURE 1.1: Left side: Light scattering in 3D graphics. Right side: Light scattering in the real world. (Image sources: Top-Left: [Liv13], Top-Right: [Ina08], Bottom-Left: [Fou08], Bottom-Right: [Fel15])

The volumetric lighting effect has different requirements than those of the real-world application of the light scattering effect for the purposes of enriching a work of art. In online, i.e. real-time, rendering, a balance between quality and performance is essential

for the overall outcome. As we already stated, this technique's goal is to simulate light's behavior as it passes through a space where interaction with participating media takes place. For current hardware, an exact, physically correct simulation would be computationally impossible for online rendering, due to reasons such as the peculiar nature of light, the variation of physical attributes of each and every intersecting particle and the complexity of the collisions that occur. Therefore, the use of several mathematical approximations and various optimization techniques in the implementation, is required to achieve the replication of the phenomenon in real time with lesser computational complexity.

As an important side note, volumetric lighting is typically implemented as a post-process effect. Those effects operate differently than regular graphical effects. Rather than rendering the visible part of the scene to the view-port, a post-process effect generates a texture of the camera's view, which is a 2-dimensional off-screen image of the 3-dimensional scene (see Figure 1.2). Positive aspects of this loss of dimensionality include the simplification and accessibility of editing of the generated texture, for example, by application of various popular performance and optimization image processing algorithms.

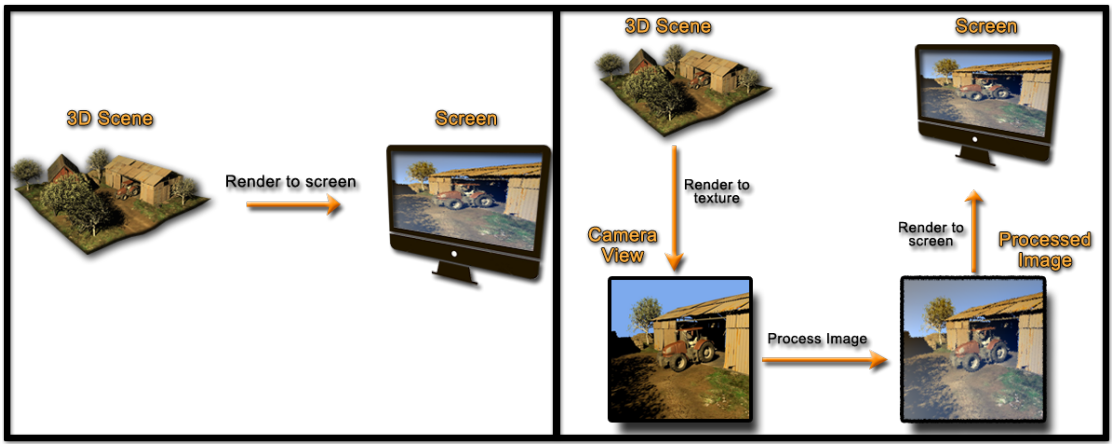

FIGURE 1.2: Difference between regular and post-process effects. Left: Regular effect, the camera view is directly rendered to the screen with the applied effect. Right: post-process effect, the camera view is stored as an off-screen image that is later processed and finally rendered to the screen.

1.3 Environment Mapping Effect

The majority of real-world surfaces reflect a portion of light creating a particular mirroring of their surrounding environment. The way light bounces off them varies and predominantly depends on the material by which they are made of. There are many types of reflections, such as metallic, blurry and polished. Those categories arise from the combination of the following two phenomena. When light hits a smooth surface, we notice a mirror-like reflection of the ambient, whereas, when the impact is on a rougher

surface, we observe a diffused image of the environment. Those events are classified as highly directional, i.e. specular, and scattered, i.e. glossy, reflections respectively, further analyzed in Section 2.2.1 (to visualize the difference between specular and glossy reflections see Figure 1.3).

FIGURE 1.3: Left: Specular reflection, the surface of the water is calm therefore we can see a clear reflection of the surrounding environment. Right: Glossy reflection, the surface of the water is "wavy" and helps to visualize how a rough, bumpy surface appears in the microscopic level. In this image we notice a diffused reflection of the ambient.

To replicate this physical phenomenon in our virtual world, we perform the environment mapping image-based lighting technique, that utilizes texture images to display the reflection of an object's surrounding environment. This effect is applied in many fields, especially in 3D gaming as the ray-tracing alternative is too computationally complex for the demands of a video game's world (see [Var16, sect. 1.2.3] for more information on ray-tracing). Figure 1.4 depicts how the environment information, light and reflections, affect the overall aesthetics and realism of a video game world.

FIGURE 1.4: Ambient lighting and reflections of the environment. Notice that in each image those reflections on the scene's objects vary and depends on their surface material properties. Left: Black Desert Online. Middle: The Elder Scrolls IV: Oblivion (modded). Left: Dark Souls III. (Image sources: Left: [Aby14], Middle: [Stu06], Right: [Nam16])

The most widely used method to represent the scene's ambient, is to assemble a skybox, which is a 3-dimensional box of textures, also known as a cube map, that encases the camera and appears to be at the same distance from any location that it is observed. This creates the illusion that the viewer is positioned inside an enormous sphere, similarly

to how a person perceives the real world external surrounding environment (see Figure 1.5).

FIGURE 1.5: Left: Skybox in Minecraft (modded). Right: Surrounding environment in the real world. (Image sources: Left: [PB09]

Next, when the focus is directed at any object inside the scene, the suitable part of the aforementioned skybox is drawn onto its surface to depict this environment reflection phenomenon (Figure 1.6).

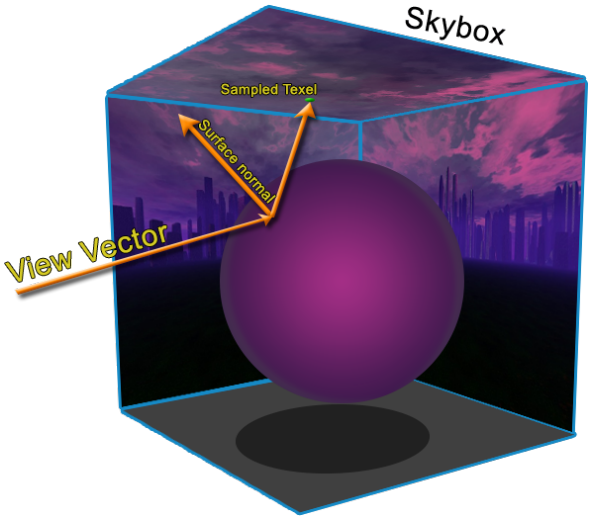

FIGURE 1.6: When the viewer focuses at an object inside the scene, the skybox that contains the environment information is sampled in order to obtain the information necessary to create its reflection on the object's surface

The resulting effect and processing time may vary according to the way it is implemented. For example, the different properties of each surface, such as roughness and metallicity, can be utilized in the reflection's calculations to improve the effect's quality. This way, we can realistically replicate both specular and glossy reflections. Even more, one can pre-compute cube maps around different locations inside the scene to include most or even all of the objects' information in the reflections. This is known

as the dynamic environment mapping technique (Figure 1.7). Although this sophisticated approach creates more convincing results in situations where an object's visible surroundings are heavily populated by other (dynamic) geometry, the pre-computation time and scene complexity may have an overall negative influence on the performance. For our implementation of the effect, our goal is to approximately replicate this phenomenon in real-time rendering. Therefore, even though we include both specular and glossy reflections to enhance the accuracy of the effect, we do not involve other objects' geometrical information in the reflection calculations, to avoid any detrimental effects on the implementation's performance.

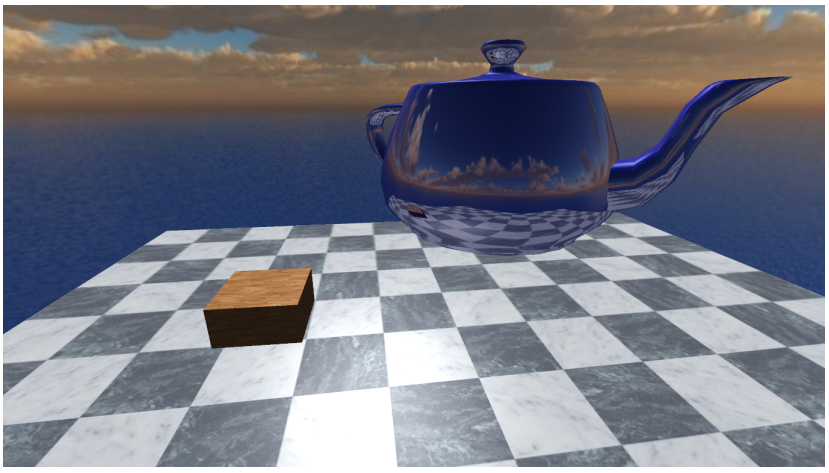

FIGURE 1.7: Dynamic environment mapping. Notice how both the skybox and the surrounding object information is apparent in the teapot's surface reflections. (Image source: [Geo13])

Chapter 2

Theoretical Background

In this chapter, we focus on the theoretical background required to make our implementation of both effects more physically correct. We divide the following segment into two subsections. We begin with the scientific concepts and mathematical approximations necessary for a more realistic and optimized construction of the volumetric lighting poss-process effect (see Section 2.1). Then, we provide the necessary methods essential for an accurate and computationally inexpensive execution of the environment mapping technique (see Section 2.2).

2.1 Theoretical Background in Volumetric Lighting

Regarding volumetric lighting, we begin by providing an important rendering equation necessary for the production of an accurate representation of the real world light scattering phenomenon (see Section 2.1.1). Subsequently, we present a range of available formulae that simulate how light scatters when interacts with participating media (Section 2.1.2). We continue with one more approximation of how light attenuates as it travels to any point in space (Section 2.1.3). Next, we examine a blurring algorithm that could be applied on our final result to hide imperfections with little to no encumbrance on our implementation's performance (Section 2.1.4). Last but not least, we explain how to achieve better visual results by utilizing a specific anti-aliasing technique, typically used on the scene's shadows construction (Section 2.1.5).

2.1.1 Volume Rendering Equation

The rendering equation proposed by Kajiya in 1986 [Kaj86] is responsible for describing the light and matter interactions inside a virtual scene. It accounts for the conservation of energy during different events that occur from those interactions, in order to create more

physically correct and photo-realistic results. Because its exact calculation involves the computation of high dimensional integrals it is generally approximated via mathematical techniques such as the Monte Carlo Integration [Sob94]. In volumetric lighting, we aim to simulate the physical properties of light and the medium in which it travels through and interacts. Therefore, we need a variation of the rendering equation which is the volume rendering equation (see Equation 2.1 for a general form of this equation) which accounts for different events occurring inside this space/volume such as those depicted in Figure 2.1.

$$L(x,\omega_i) = T_r(x, x+t\omega_i)L(x+t\omega_i,\omega_i) + \int_0^t T_r(x, x+s\omega_i)\sigma_s(x+s\omega_i)L_s(x+s\omega_i,\omega_i)ds$$

$$L_s(x,\omega_i) \equiv L_e(x,\omega_i) + \int_S p(\omega_i,\omega)L(x,\omega)d\omega \tag{2.1}$$

where:

x	$=$ *point inside volume (on ray)*
ω	$=$ *direction of ray inside volume*
T_r	$= e^{-\int_0^t \sigma_t(x+s)ds}$ *(Transmittance)*
$\sigma_t(x)$	$= \sigma_\alpha(x) + \sigma_s(x)$ *(Attenuation coefficient)*
$\sigma_\alpha(x)$	$=$ *absorption coefficient*
$\sigma_s(x)$	$=$ *scattering coefficient*
$T_r(x, x+t\omega_i)L(x+t\omega_i,\omega_i)$	$=$ *absorption + out-scattering*
$L_e(x,\omega_i)$	$=$ *emission*
$\int_S p(\omega_i,\omega)L(x,\omega)d\omega$	$=$ *in-scattering*
$p(\omega_i,\omega)$	$=$ *phase function*

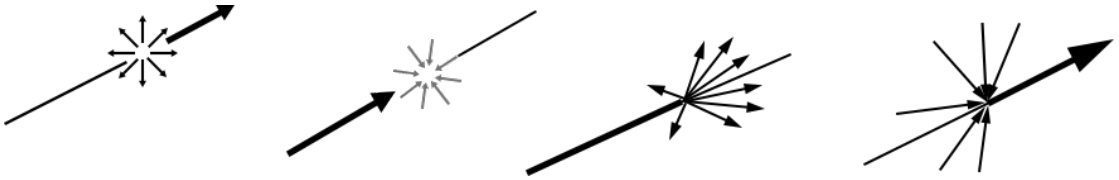

FIGURE 2.1: Interactions with medium. Left: Emission, light emitted from point of interest. Middle-Left: Absorption, light absorbed in point of interest. Middle-Right: Out-scattering, light is scattered away from the point of interest. Right: In-scattering, light is scattered into the point of interest.

2.1.2 Light Scattering Phase Functions

Phase functions are mathematical formulae that provide means to approximate light's scattering behavior inside a medium filled with different particles such as dust, water, air molecules, skin or even blood. For now, we will not delve into the wave-particle nature of light, however we will treat it as an abstract structure that is able to collide with particles and scatter. As light travels through an area with dispersed participating media, collision with some of those particles causes a range of different scattering events. In general, those occurrences are classified as back, forward or isotropic scattering: back-scattering, where redirected light tends to return towards its source, forward scattering, where the light is deflected further away and isotropic scattering, where the scattered light spreads homogeneously (Figure 2.2).

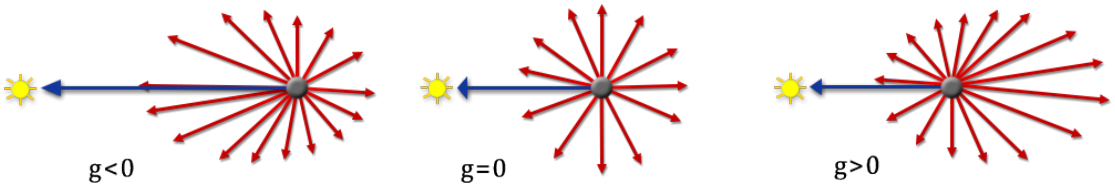

FIGURE 2.2: The yellow dot represents the light's source, whereas the black dot represents the particle that causes light's scattering. The length of each red arrow represents the likelihood that a scattered ray follows that specific direction after its collision. Right: Back scattering. Middle: Isotropic scattering. Left: Forward scattering.

The way phase functions simulate light's diffusion from particles varies in each function and depends from coefficients that define whether those collisions lead to any of the aforementioned cases. Some phase functions even take into consideration the light's characteristics and the properties of the participating media. For more information about scattering fundamentals and more details on the following phase functions the interested reader is directed to [Jar08, chap. 4].

Henyey-Greenstein Phase Function

The Henyey-Greenstein [HG41] scattering phase function (Equation 2.2) was used in a variety of different fields, including astrophysics and oceanography. Even though, nowadays this particular phase function has been replaced by highly complicated formulae that simulate scattering events more accurately, its mathematical simplicity and accessibility along with its overall approximative performance constitutes a satisfactory candidate for graphical simulation.

$$P_{HG}(x,\theta) = \frac{1-g^2}{4\pi(1+g^2-2gcos\theta)^{1.5}} \tag{2.2}$$

where:

x = *scattering point, i.e. point of light's collision with a particle*
θ = *scattering angle, i.e. the angle of light's deflection caused by collision with a particle*
g = *average of the cosine of the scattering angle*

The variable g varies in the range of [-1,1] and by adjusting its value, we control the amount of back-scattering and forward scattering of the light in the medium as depicted in Figure 2.2.

Schlick Phase Function

Although Henyey-Greenstein's formula is a useful tool to replicate light scattering in computer graphics, the fact that there is a fractional exponent in the denominator makes it costly to calculate. In actuality, the formula needs to be calculated for each frame, for every pixel of the view-port and for any sample we accumulate to produce the final result (process described in Section 3.2.3). Therefore, an accurate approximation of the shape of Henyey-Greenstein's function with less mathematical complexity is required to remove this computational cost. The Schlick [BSS93] Scattering phase function (Equation 2.3) was developed to combat those complications by approximating the Henyey-Greenstein function as an ellipsoid.

$$P_S(x,\theta) = \frac{1-k^2}{4\pi(1+kcos\theta)^2} \tag{2.3}$$

where:

x = *scattering point, i.e. point of light's collision with a particle*
θ = *scattering angle, i.e. the angle of light's deflection caused by collision with a particle*
k = $1.55g - 0.55g^3 (approximately)$.

The variable k also varies in the range of [-1,1] and acts similarly to g in Henyey-Greenstein Scattering (Equation 2.2).

Rayleigh Scattering

Another popular phase function we use in our implementation is the Rayleigh [Str71] Scattering phase function (Equation 2.4). It is used to estimate the way light scatters inside a medium composed of particles up to about a tenth of the wavelength of light. This formula takes into account the light's wavelength and the particles' characteristics. It simulates the participating media as randomly distributed micro spheres with properties such as diameter and refractive index.

$$P_R(x,\theta) = \frac{3}{16\pi}(1 + cos^2\theta)$$
$$\sigma_s = \frac{2\pi^5}{3}\frac{d^6}{\lambda^4}\left(\frac{n^2-1}{n^2+2}\right)^2 \tag{2.4}$$

where:

x = *scattering point, i.e. point of light's collision with a particle*
θ = *scattering angle, i.e. the angle of light's deflection caused by collision with a particle*
σ_s = *scattering coefficient.*
d = *particles' diameter*
n = *particles' refractive index*
λ = *light's wavelength*

Lorenz-Mie Theory

In situations where particles are comparable to the size of light's wavelength, for example droplets of water in fog, the Rayleigh Scattering approximation is not accurate. Therefore, we apply Lorenz's and Mie's Theory [Lor90, Mie08] of scattering for better simulations of such events. In our implementation, we use the two following empirically derived approximations provided by [NMN87]. The Lorenz-Mie Hazy phase function (Equation 2.5) for "hazy" atmospheric conditions

$$P_{LMH}(x,\theta) = \frac{1}{4\pi}\left(\frac{1}{2} + \frac{9}{2}\left(\frac{1+cos\theta}{2}\right)^8\right) \tag{2.5}$$

where:

x = *scattering point, i.e. point of light's collision with a particle*
θ = *scattering angle, i.e. the angle of light's deflection caused by collision with a particle*

and the Lorenz-Mie Murky phase function (Equation 2.6) for "murky" atmosphere.

$$P_{HG}(x,\theta) = \frac{1}{4\pi}\left(\frac{1}{2} + \frac{33}{2}\left(\frac{1+cos\theta}{2}\right)^{32}\right) \tag{2.6}$$

where:

x = *scattering point, i.e. point of light's collision with a particle*
θ = *scattering angle, i.e. the angle of light's deflection caused by collision with a particle*

2.1.3 Light Attenuation

As mentioned in the previous section, there are multiple phase functions to simulate light's scattering behavior as it diffuses from particles in the medium. This wave-particle redirection occurs when light collides with participating media. For simplification purposes, we will consider light to be made up of particles known as photons, and totally ignore its dual-wave nature. Photons are carriers of photon energy and momentum. When a photon collides with a particle, apart from being deflected from its current path (scattering), it also losses both energy and momentum depending on the impact angle and the particle's characteristics (see Figure 2.3). Based on its energy package, the photon may be completely absorbed during the impact. This leads to the transfer of energy from the light's system to the participating media thus causing its attenuation (for more details on this subject the interested reader is referred to [Wei09]).

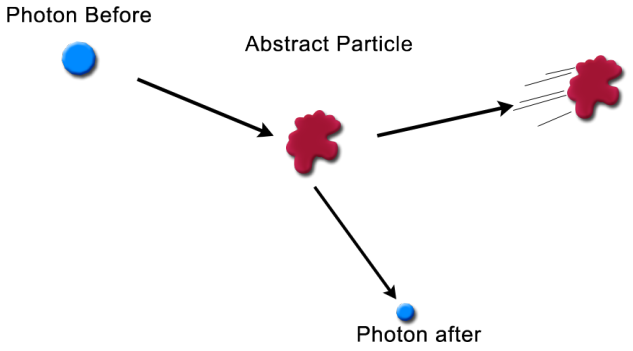

FIGURE 2.3: Photon collision with a particle. After their interaction, the particle gains kinetic energy whereas the photon loses energy and momentum.

In most cases, the way light attenuates as it moves through the medium is exponentially proportional to its traveling distance divided by a factor dependent on the particles' physical attributes. In our approximation formula (see Equation 2.7), the influence on the light's attenuation by dispersed participating media is adjusted by the coefficient known as optical thickness, which is experimentally derived for a range of materials and is dependent on their physical constitution and form.

$$L_A(x, l) = e^{-\frac{x}{l}} \tag{2.7}$$

where:

x = *length of light's path*
l = *medium's optical thickness*

2.1.4 Gaussian Blur

In photo-realistic rendering, we seek to achieve the most realistic outcome with the limitations of hardware's performance. Our implementation's goal is to approximate a specific physical phenomenon in real time. The cost of duplicating such an effect could be detrimental to the ability of online rendering. So, by reducing our result's quality and generating noise to hide visible artifacts produced by that downgrading, we could minimize the computational power required to construct an authentic representation of the volumetric lighting visual effect.

As stated in the introduction to volumetric lighting (see Section 1.2), this specific technique is performed in post process. Therefore, by application of image processing algorithms, such as the Gaussian blur, we can produce better looking results. With Gaussian blur, we achieve reduction of potential noise in our outcome by cutting down on detail, creating the illusion of smoothness to the final image (see Figure 2.4). The way this algorithm operates, is by constructing a new image where each pixel's color value is equal to the weighted average value of the inputted image's corresponding pixel's neighborhood. Every nearby pixel's contribution to the total weights differently and depends on its relative position to the main pixel's location. For more technical information on Gaussian blur, readers are referred to [Rá10].

FIGURE 2.4: Example of Gaussian blur application for noise reduction.

2.1.5 Shadow Mapping and Percentage Closer Filtering

Shadow mapping [Wil78] is a rendering technique used to add shadows to a virtual scene. By comparing a pixel's location to a depth buffer called shadow map, which operates as the light source's view depth buffer and is stored in a form of a texture, the process distinguishes whether each fragment is illuminated or not (process illustrated in Figure 2.5). Volumetric lighting effect's visual results are heavily dependent on the shadow map's quality, which in turn considerably influences the overall performance of

the scene. If the shadow map's resolution is low, the binary relation that separates the illuminated from the dark pixels can produce noticeable aliasing near the borders of light's and shadow's territories, and as a result, negatively affect the quality of the effect.

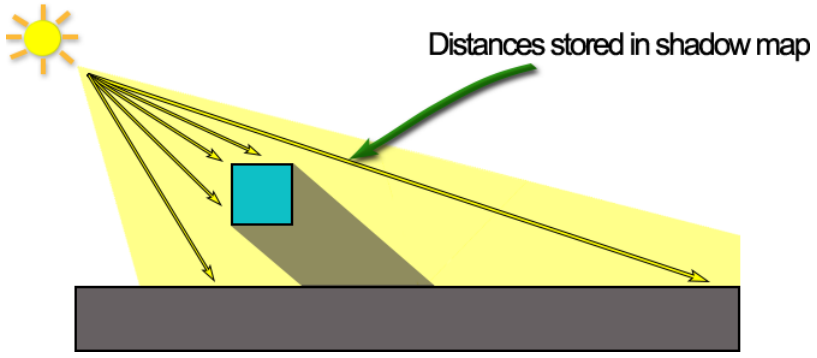

FIGURE 2.5: If a pixel's location is further away from the light source than the corresponding shadow map value then the pixel is dark. Otherwise, it is illuminated.

To intelligently combat such implications, instead of simply increasing shadow map's resolution, we use a technique similar to Gaussian blur, which is shadow blur with percentage closer filtering (also known as PCF). With this algorithm, we smoothen the boundaries between light and shadow to hide any visible aliasing. In percentage closer filtering, we average the neighborhoods of the shadow map's texels to differentiate points in the scene as partially illuminated. This method creates a gradient around the borders as seen in Figure 2.6 and as consequence, achieves the anti-aliasing of shadow along its edges.

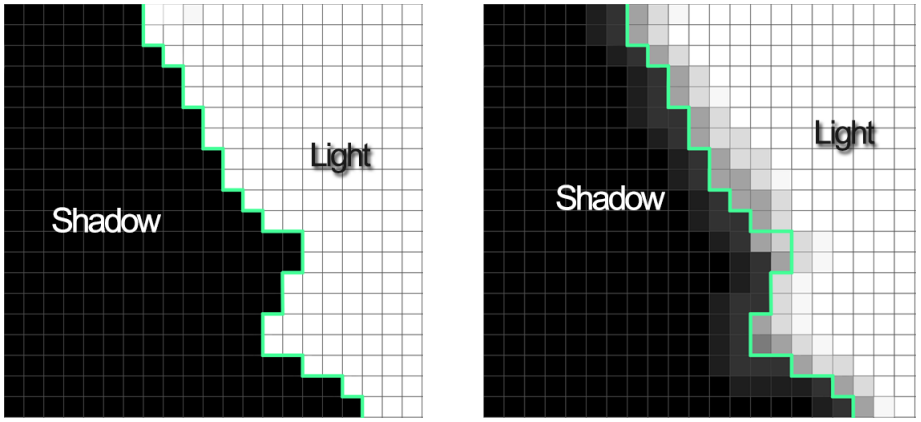

FIGURE 2.6: Left: The pixels are either dark or illuminated, this can create aliasing due to many reasons such as the light's and objects' position and the resolution of the shadow map. Right: By application of percentage closer filtering, the pixels can now be partially illuminated which smoothens the boarders of light and shadow.

2.2 Theoretical Background in Environment Mapping

Regarding environment mapping, we start with an overview of reflection fundamentals and categorizations (on Section 2.2.1). We continue with a popular, empirically derived method, commonly utilized for the approximation of light's reflection at an opaque surface (Section 2.2.2). Finally, we explore an important numerical integration technique that could be applied on our implementation to simulate the conservation of energy originating from ambient lighting that is received from the scene objects' surfaces (Section 2.2.3).

2.2.1 Reflections

In general, when a light ray collides with an opaque surface, it may either be absorbed or reflected. The final outcome mainly depends on different factors, such as the surface's material properties, light's characteristics (e.g. wavelength) and the angle of impact. There is an extremely small amount of artificially made materials that completely reflect or absorb light and are used in fields such as camera lens production and military stealth technology. Most of the real-world's surfaces comprise a combination of those two effects.

As mentioned in 1.3, reflected light is divided into two sub-categories, specular and glossy. Before we explain them further, we should first provide common terminology that appears in all cases. Incident light consists of incoming rays of light that hit the surface, whereas, the reflected light refers to the outgoing rays that are bounced away. The surface normal in this case, is the vector which is perpendicular to the surface at the point of light's collision. The angle of incidence and reflection, are the angles between light's incoming and reflected rays to the surface normal respectively. The following Figure 2.7 depicts the aforementioned concepts in an illustrative way.

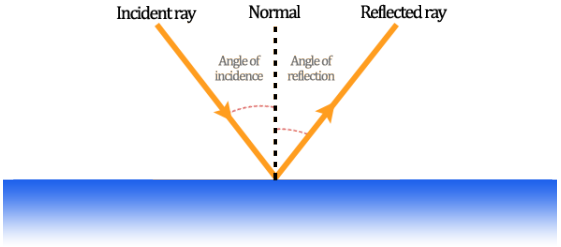

FIGURE 2.7: This figure illustrates the basic concepts of a reflection.

Specular Reflections

Specular reflections are the mirror-like reflections of light created from its collision to a smooth surface (see Figure 2.8). During this phenomenon, the amount of incident and reflected light rays is the same. Also their angle of incidence is equal to the angle of

reflection. This creates a clear image of the surface's surrounding environment similarly to how a mirror does.

FIGURE 2.8: Specular Reflection.

Diffuse Reflections

In diffuse reflections the light rays are scattered in multiple angles rather than at just one angle as in the case of specular reflection (see Figure 2.8). Lambertian reflection is considered an ideal diffuse reflection, meaning that there is equal luminance from any point of view.

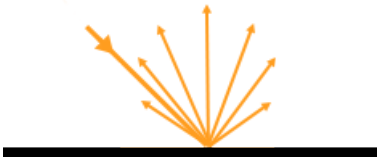

FIGURE 2.9: Diffuse Reflection.

Glossy Reflections

Glossy reflections are more common in nature, as surfaces tend to be imperfect, therefore rough. When light intersects with those surfaces (see Figure 2.10), its incidence angle differs from the reflected angle. Hence, the ambient reflection on the surface is distorted. The distortion of the reflection is dependant to its amount of specular reflection in comparison with diffuse reflection.

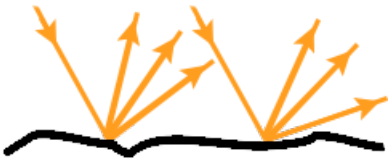

FIGURE 2.10: Glossy Reflection.

2.2.2 Bidirectional Reflectance Distribution Function

In the previous section we classified the primary categories of reflections. As we mentioned in 1.3, the majority of real-world reflections on surfaces are a combination of those types. Also, as noted, the material properties of those surfaces are mainly responsible for the visual outcome of the phenomenon as they determine the amount of

glossiness (specular/diffuse comparison) of each of them. Therefore, for this implementation, an adequate approximation of the effect that material properties have on the visual outcome, is required to correctly simulate the greatest part of those phenomena.

A commonly used function for those estimations is the bidirectional reflectance distribution function or BRDF [NW09]. In general, the BRDF is defined as the ratio of the differential outgoing radiance to the differential irradiance (see Figure 2.11) and for each specific point x of the surface is dependant on the incident and reflected light directions (Equation 2.8). For more information on BRDF fundamentals, the interested reader is referred to [PJH17, chap. 8-9].

$$BRDF(x, \omega_o, \omega_i) = \frac{dL(x, \omega_o)}{dE(x, \omega_i)} \tag{2.8}$$

where:

x = *point on the surface*
ω_o = *reflected light direction*
ω_i = *incident light direction*

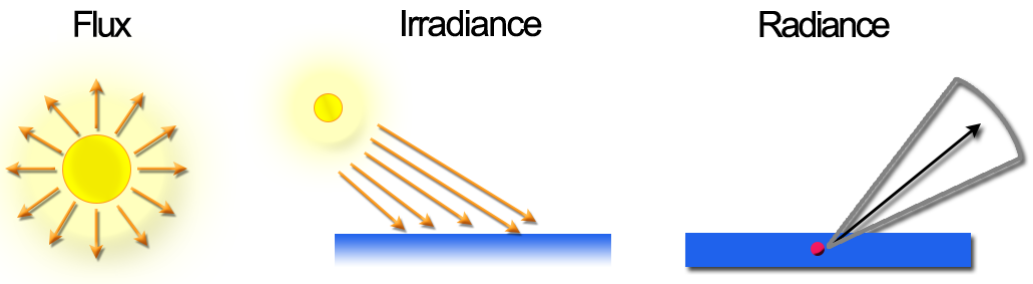

FIGURE 2.11: Flux: the total radiant energy emitted, reflected, transmitted or received, per unit time. Irradiance: the radiant energy received by a surface per unit area. Radiance: the radiant energy emitted, reflected, transmitted or received by a given surface, per unit projected solid angle.

There is a wide range of BRDF functions that operate differently and are typically based on empirical observations. In general, they are divided on surface reflectance, i.e. specular, BRDFs and body reflectance, i.e. diffuse, BRDFs. For the purposes of this thesis, we provide a brief description of the BRDF models utilized for the integration of both of those categories in the implementation.

Specular BRDF

Concerning surface reflectance, we follow the principles of the Microfacet model [TS67, CT82, WMLT07] that simulates the surface as several microscopic, planar and ideal reflectors, meaning that each of them reflect incoming rays of light in only one particular outgoing direction. This model is mathematically represented from the following

Equation 2.9.

$$BRDF_{specular}(l,v) = \frac{D(h,n)F(u,h)G(u,n)}{4(n\cdot l)(n\cdot v)} \tag{2.9}$$

where:

l = *light direction vector*
v = *view direction vector*
n = *surface normal vector*
$h = \frac{v+l}{||v+l||}$, *halfway vector, equals to microfacet normal*

In Equation 2.9, we notice the following three important terms of the Cook-Torrance model:

- **D**istribution Function

  This term represents the micro-facet density oriented along a specific direction. For this implementation we use the Beckmann distribution function [Kat64] to describe the dispersal of micro-facets on the surface (Equation 2.10, Figure 2.12).

$$D_{Beckmann}(h,n) = \frac{1}{\pi a^2 (h\cdot n)^4} \cdot e^{\left(\frac{(h\cdot n)^2 - 1}{a^2 (h\cdot n)^2}\right)} \tag{2.10}$$

  where:

  a = *roughness*
  n = *surface normal vector*
  $h = \frac{v+l}{||v+l||}$, *halfway vector, equals to microfacet normal*

FIGURE 2.12: Beckmann distribution function's directionality and size of the lobe signifies the direction and width of the specular highlight. (Images rendered using Disney's BRDF Explorer [DS13])

- **F**resnel

  The Fresnel function describes the amount of light that reflects from a mirror surface given its index of refraction. For this term we use the Schlick Fresnel [Sch94] formula (Equation 2.11).

$$F_{Schlick}(v,h) = F_0 + (1 - F_0)(1 - v \cdot h)^5 \tag{2.11}$$

  where:

  F_0 = *reflectance at normal incidence which depends on the surface's material properties*
  v = *view direction vector*
  $h = \frac{v+l}{||v+l||}$, *halfway vector, equals to microfacet normal*

- **G**eometric Shadowing

  The geometric shadowing term describes the shadowing originating from the microfacets. As this is dependent on their distribution, we employ Beckmann geometric shadowing [Kat64] as well (Equation 2.12).

$$c = \frac{n \cdot v}{a\sqrt{1 - (n \cdot v)^2}}$$
$$G_{Beckmann}(v,n) = \begin{cases} \frac{3.535c + 2.181c^2}{1 + 2.276c + 2.577c^2} & c < 1.6 \\ 1 & c \geq 1.6 \end{cases} \tag{2.12}$$

  where:

  a = *roughness*
  v = *view direction vector*
  n = *surface normal vector*

Diffuse BRDF

Although the Lambert BRDF [Lam60] is the simplest body reflectance model, it is still widely used as its approximative results are sufficient for photo-realistic rendering. According to this diffuse BRDF, an illuminated surface is equally bright from any point of view (Equation 2.13, Figure 2.13).

$$BRDF_{diffuse}(x) = \frac{p(x)}{\pi} \tag{2.13}$$

where:

$p(x)$ = *albedo, i.e. energy reflected, at point x on surface*

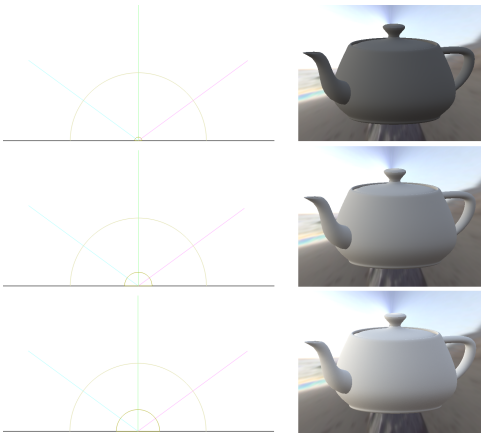

FIGURE 2.13: Lambert BRDF has no directional dependence and the size of its "lobe" (inner green circle), which is dependant on the surface's point albedo, determines the average amount of energy from all of the surface's diffusion paths. (Images rendered using Disney's BRDF Explorer [DS13])

2.2.3 Irradiance Mapping and Monte Carlo Integration

In radiometry, irradiance accounts for the radiant flux, i.e. energy of electromagnetic radiation, that is received by a surface per unit area (see Figure 2.11). For this effect the irradiance that surfaces collect, is the radiant flux originating from the ambient lighting. This light is derived from the environment information, which is comprised of several textures assembled in a cube map, the sky box (mentioned in 1.3).

For this implementation, we take the surface material properties into consideration to improve the quality and accuracy of the effect. As it will be explained further in 3.3, to reduce the computational load of this technique, we pre-filter different levels of irradiance for the sky box textures, based on hard-coded roughness values. As information "diffuses" in higher roughness levels, those irradiance maps should account for the conservation of energy of the environment textures. For the exact computation of each irradiance level, we would have to use all of the texel information stored inside the primary cube map structure. This would be extremely time consuming even if we generated and stored those levels only once for each appointed environment map.

Monte Carlo Integration [Sob94] is a numerical technique that could be utilized for the estimation of each level of irradiance. Its concept is fairly simple and can be easily applied in our case. According to theory, to approximate a definite, n-dimensional integral, random samples are chosen over the integration domain, and the integral is calculated based on the contribution of each of those samples weighted by their probability of selection (for more information see [PJH17, chap. 13]). Hence, to apply this method to our implementation, instead of using every texel's information of the aforementioned cube map, we perform randomized, uniform cone sampling in different parts of it to estimate the contribution of its entire data set (detailed methodology in Section 3.3.3).

Chapter 3

Implementation

3.1 Experiments Setup

All of the following tests are performed on an Intel Core i7 6700K @ 4.00GHz CPU and a Radeon R9 290 GPU rendered in a resolution of 1024x1024. However, note that throughout the duration of this thesis, there have also been several tests on systems with NVIDIA graphics cards (GTX980Ti).

3.2 Volumetric Lighting Implementation

In this subsection, we start by exploring XEngine's assets used for the facilitation of the effect's production (Section 3.2.1). Next, we provide a detailed work-flow of our implementation containing the important parts, in which it is divided (Sections 3.2.3, 3.2.4 and 3.2.5). For each specific stage, we present multiple images illustrating concepts and processes applied, comparative screen-shots of the virtual effect rendered by XEngine and, finally, an algorithmic representation of its gradual development in pseudo-code.

3.2.1 XEngine's Dependencies

In this thesis, one of our major goals is to construct an approximate replication of the real-world light scattering phenomenon into a virtual world. As we mentioned in the engine's introductory segment (Section 1.1), the XEngine is can be a host to the implementation of our graphical effects. Nevertheless, for the sake of completeness, in the following paragraphs, we comment on pre-existing key engine components that this engine possesses essential for the execution of the volumetric lighting effect.

Post Process Effects

The XEngine contains multiple different techniques that were created over the years by the AUEB Graphics Group. Programmatically, each one of those techniques is inserted as an inheritor of the main DRTechnique class and shares a range of common properties and functions with the rest. As previously indicated, the volumetric Lighting effect is typically implemented as a post process technique. The engine is prepared for such occasions by including another general-purpose technique class named DRTechniquePostProcess that inherits the typical aforementioned attributes from its parent, DRTechnique. Post process effects such as fog, tone mapping and volumetric lighting are applied as sub-classes of this class. The generated objects are inserted into a hash map, located in the DeferredRenderer class, that holds all activated effects, which are used to render the final image. Every effect's object contains one or more numerical ids that associate it with its buffers. Those data structures contain texture information (e.g. the 2-dimensional off-screen image mentioned in 1.2) that is essential to their holders operation. For every individual frame, those buffer's internal values are mostly updated and alternately blended to produce the final result that is ultimately displayed on the screen.

Lighting System

The volumetric lighting technique could not operate without the use of a lighting system. This engine provides a light manager and a variety of light source types to choose from. For the purposes of this thesis, we use spot lights, as they are sufficient for a realistic approximation of the phenomenon. Spot lights are a subcategory of point lights that can closely approximate their actual real world counterparts. They have properties such as their source's position and target, their luminous flux and the geometric shape by which their light is emitted, which can be either conical or pyramidal. Finally, by granting access to any light in the scene, at any point, the light manager not only facilitates the overall development of the effect, but also provides us with the opportunity to expand the implementation for multiple concurrent active spot lights.

The Scene Graph

The SceneGraph library is utilized for the purposes of element declaration, such as geometry, transformations and effects, either by parsing an XML file containing Scene Description Language or via the usage of a C API, named XEAPI, that communicates directly to the SceneGraph API through a wide variety of function calls. The information that is gathered for each of those elements is stored in internal asset manager objects and used by other parts of the engine such as the associated techniques' objects for their shader parameterization. The VolumetricLightingEffect is one of those manager classes responsible for the preservation and accessibility of the currently active volumetric

lighting effect's attributes. An example of this effect's declaration in Scene Description Language is shown in Figure 3.1.

```
<?xml version="1.0" encoding="utf-8"?>
<world background="0.2, 0.4, 0.85" ambient="0.0 0.0 0.0" show_fps="true">
    <font name="consola" filename="consolab" size="12" color="1,0.9,0.7,1.0" has_shadow="true"
        shadow_size = "1.0" shadow_color="0.0 0.0 0.0 1.0" shadow_dir="1.0 -1.0" active = "false"/>
    <directory path="barn"/>
    <camera name="first_person" aperture="55.0" near="1" far="50" follow="default_user" primary
        ="true"></camera>
    <eventmessage event="init" recipient="user_spin" message="stop"/>
    <light name="sun" shadows="on" flux="300000" color="0.8 0.57 0.21" active="true" type
        ="spotlight" rsm="true"
     far_range="100" near_range="5" position="30, 40, 65" target="0,0,0"
     conical="true" penumbra="2"  resolution="4096" aperture="60" soft_shadow_size="1" constant_bias
        ="0.00004"></light>
    <object file="barn_houses_simple.obj"/>
    <transformation name="env" rotation="6,1,0,0" scale="1,1,1" translation="0,-0.05,4">
        <object file="tracktor2.obj"/>
    </transformation>
    <user name="default_user" control="roaming" linear_speed="3" angular_speed="1"
    position = "4.174667 2.102174 -4.759601"
    target   = "3.671521 2.104612 -2.908863"
    input="default_input"></user>
    <input name="default_input" devicename="device0">
        <eventmessage event="button2pressed" recipient="user_spin" message="start"/>
        <eventmessage event="button2released" recipient="user_spin" message="stop" />
    </input>
    <volumetric_lighting name="vol" active="1" samples="500" resolution_scale="0.5" blur="1"
        phase_function="rayleigh"  particle_animation="0" particle_animation_type="dusty"
        optical_thickness="18.76"/>
</world>
```

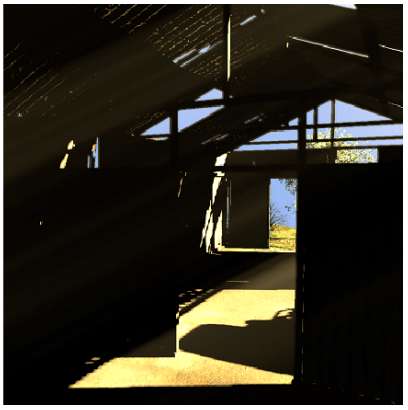

FIGURE 3.1: Example of basic scene description language and volumetric lighting parameterization. (Image rendered using XEngine [PV13])

Previewing Functionality

Another powerful tool included in XEngine is its previewing functionality. This mechanism provides means for easier debugging and quality evaluation of elements such as effects and lights. When enabled, the previewer operates by displaying particular buffer texture data onto the screen. By cycling through loaded buffers, we are able to visualize the internal values of the scene depth buffer, shadow maps, occlusion maps and many more items and to determine whether they are implemented correctly or not. For volumetric lighting as we mentioned above, the data of the effect is stored in a volumetric lighting intermediate buffer. Therefore, this structure, containing the light volume information separated from the rest of the scene, can be displayed via the previewer (example Figure in 3.2). This is beneficial for the technique's debugging and quality assessment.

FIGURE 3.2: Example of previewing functionality. Left: Final result displayed on the screen. Right: Isolated Volumetric lighting information visualized via the previewer. (Images rendered using XEngine [PV13])

3.2.2 Approximating the Volumetric Rendering Equation

In volumetric lighting, we seek to identify and distinguish the volume covered by the light's cone of influence in the 3-dimensional space. This light volume is defined as the space in which light constantly travels through and occupies a 3-dimensional integral of a potentially ever-changing and complicated geometrical arrangement. As discussed in Section 2.1.1, to achieve a physically correct approximation of the energy flow and interactions between light and participating media inside this space/volume we would have to use the volume rendering equation. However its exact calculation can prove to be rather difficult or even computationally impossible for real time rendering. Therefore, to achieve the effect of volumetric lighting we use sampling by tracing the scene to define the light volume bounds (Section 3.2.3) and then for each of those samples we utilize several mathematical concepts (discussed in Section 2.1 and applied in Section 3.2.4) to simulate the various physical properties that the volume rendering equation normally accounts for (e.g. attenuation, phase function, e.t.c).

3.2.3 Light Volume Bounds

As a first approach to create the volumetric lighting effect we use the sampling method proposed in [Cor08]. This procedure manages to approximate a light volume, i.e. a 3-dimensional integral, by a form of randomized sampling. During the construction of each frame, we begin by tracing each pixel of the view-port from the camera, up to the first intersection with either a scene object or the furthest point in its path, currently rendered in the virtual world (Figure 3.3).

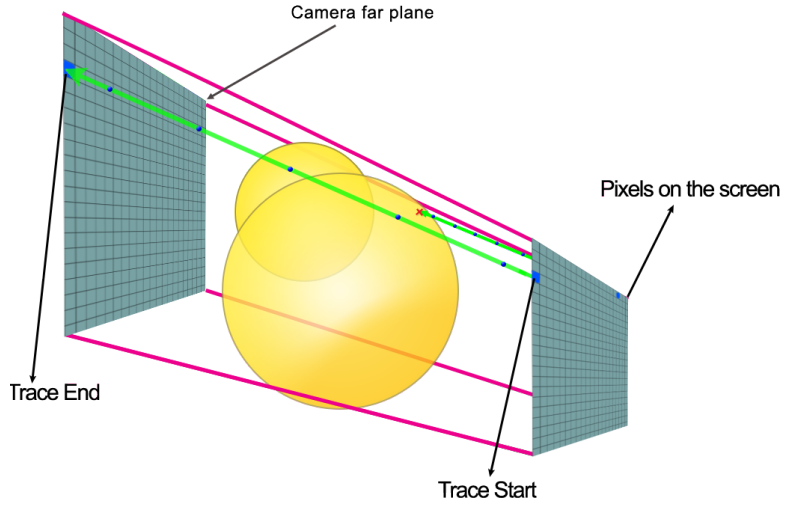

FIGURE 3.3: Green arrows represent the sampling rays. Traces begin from each pixel of the view-port and end up to either the camera's far plane or an intersecting object. Therefore, even though they have an equal amount of samples, their length is not the same.

As already mentioned in our introduction to volumetric lighting (see Section 1.2), this specific effect is done in post process which means that to reconstruct each trace's intersection point we are required to use the scene depth buffer (see following Figure 3.4).

FIGURE 3.4: Scene depth buffer visualized via the previewer. Left: Final result displayed on the screen. Right: Depth Buffer values (0 to 1), determine the depth of the pixel. Darker pixels are closer than brighter ones. (Images rendered using XEngine [PV13])

Next, we accumulate illuminated samples based on the shadow map comparison as shown in Figure 3.5 (and similarly to the way we discussed in the beginning of Section 2.1.5).

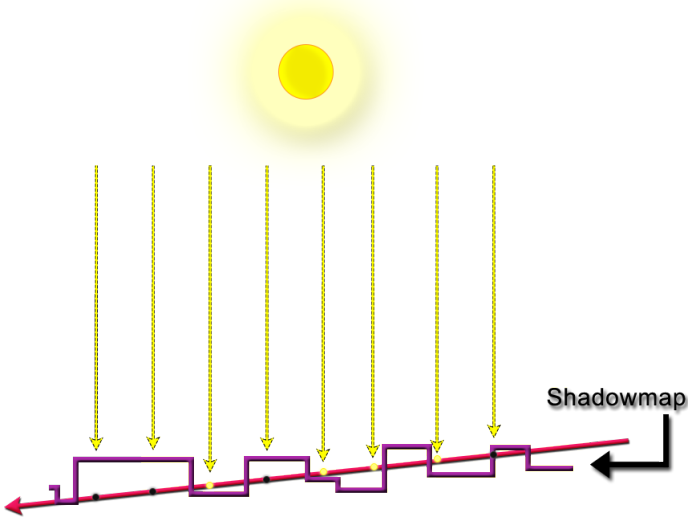

FIGURE 3.5: Red arrow represents the ray traced through each camera pixel. If a sample's distance from light is smaller than the corresponding shadow map texel value, the sample contributes to the volumetric equation integral, otherwise it does not.

Each lit sample adds to the final intensity of the pixel making the light shafts visible. Even though we use the same number of samples for all integration intervals, not all of

them have equal length (as shown in Figure 3.3). Therefore, to equalize the result for every pixel, we need to take the distance between adjacent samples into consideration. Also, note that since the effect works with spot lights (see Lighting System in Section 3.2.1), we also have to examine the fact that some of them may have a conical form of emission instead of pyramidal. On that account, we need to adjust every sample's contribution to the final intensity of the pixel based on the spot effect (all results are sequentially depicted in Figure 3.6).

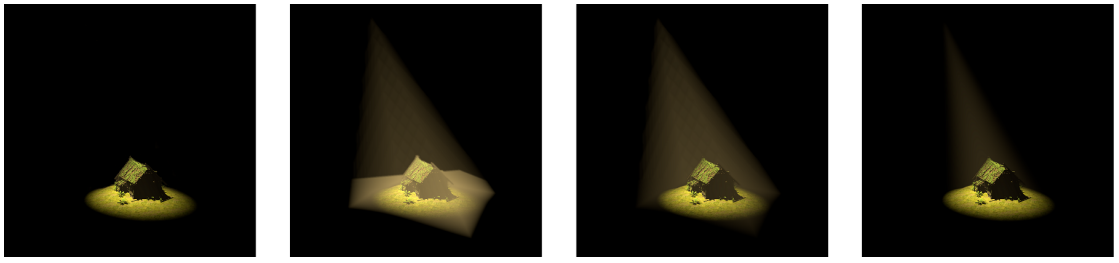

FIGURE 3.6: Left: Scene without volumetric lighting. Middle-Left: Simple emergence of light shafts. Middle-Right: Distance between adjacent samples contribution in the calculations. Right: Consideration of the spot effect. (Images rendered using XEngine [PV13], Model acquired from: [Tec15])

Finally, in Algorithm 1, the pseudo-code describes the aformentioned basic operations performed on the effect's fragment shader.

Algorithm 1 Volumetric Lighting Fragment Shader

1: $trace\_start \leftarrow camera\_position$
2: Reconstruct $trace\_end$ using the scene's depth buffer
3: $light\_factor \leftarrow 0$
4: **for all** samples on trace **do**
5: Estimate $shadow\_factor$ using the shadow map (nearest shadows $\rightarrow$ 0 or 1)
6: $light\_factor \leftarrow light\_factor + step\_distance \cdot spot\_effect \cdot shadow\_factor$
7: **end for**
8: $out\_color \leftarrow light\_factor \cdot light\_color$

3.2.4 Approximation of Physical Properties

Despite having a satisfactory approximation for the light volume bounds from the rest of the scene, the simple emergence of those light shafts is not an accurate representation of the real world physical phenomenon, since it does not account for light's physical properties. The following sections, focus on how the application of those concepts can greatly improve the realism of our implementation.

Light Scattering Phase Functions in Volumetric Lighting

As stated in the Section 2.1.2, light's collision with dispersed particles causes different scattering events such as, back, forward and isotropic scattering. In our implementation, to approximate

such occurrences, we use the phase functions to adjust each sample's contribution based on the factors that influence them.

The Henyey-Greenstein scattering phase function (Equation 2.2) is shown in Figure 3.7.

FIGURE 3.7: Left: Back scattering example, g = -0.7. Middle: Isotropic scattering example, g = 0. Right: Forward scattering example g=0.7. (Images rendered using XEngine [PV13])

The Schlick scattering phase function (Equation 2.3) is shown in the following Figure 3.8.

FIGURE 3.8: Left: Back scattering example, k = -0.7. Middle: Isotropic scattering example, k = 0. Right: Forward scattering example k=0.7. (Images rendered using XEngine [PV13])

The Rayleigh scattering phase function (Equation 2.4) is shown in Figure 3.9.

FIGURE 3.9: Intensity scaling examples of Rayleigh Scattering. (Images rendered using XEngine [PV13])

The Lorenz-Mie phase functions (Equations 2.5 and 2.6) are shown in the following Figure 3.10.

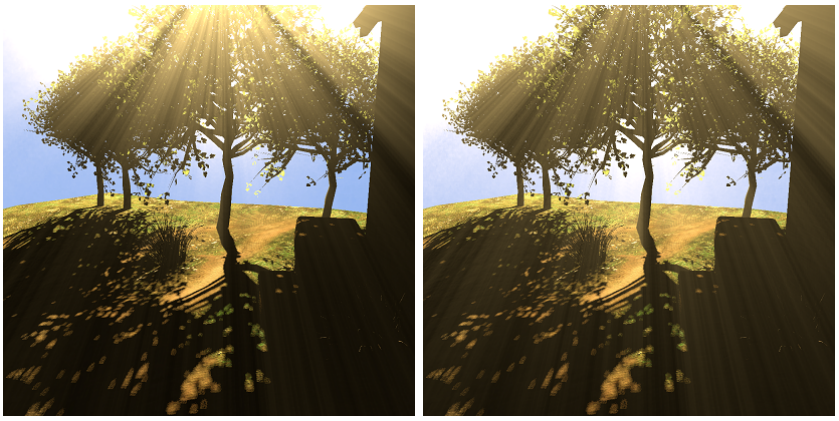

FIGURE 3.10: Left: Lorenz-Mie murky. Right: Lorenz-Mie murky. (Images rendered using XEngine [PV13])

Phase Function Animation

As shown in the previous section, there are coefficients in phase functions, responsible for light's ultimate behavior during a scattering event. In the real world, those operations can be a lot more complicated and dynamic, creating visible real-time changes in different parts of the light beams. To simulate those alterations in our virtual world, we can dynamically adjust those coefficients over time and location, to create an observable animation inside our light volumes. That way we can achieve a more aesthetically pleasing replication of the physical phenomenon (Figure 3.11).

FIGURE 3.11: Left: Real world movement of particles inside a light beam. Right: Phase function animation by adjusting scattering direction coefficients. (Left Image source: [E.m07], Right Image rendered using XEngine [PV13])

Volumetric Lighting Attenuation

In 2.1.3, we discussed light's attenuation caused by its loss of energy, originated in collisions between photons and participating media. To mathematically apply this to our approximation, we are required to consider the two following situations. First of all, we need to calculate light's

attenuation from its source up to its scattering point with regards to light's luminous power. In our implementation, we consider that every lit sample, i.e. every accumulated point inside the light's volume, is a scattering point. Secondly, the space outside light's volume contains participating media too, therefore, we should account for the fact that scattered photons may interact with it and lose even more energy. Hence, we extend light's attenuation equation to include everything mentioned above. The Equation we use is 3.1 which produces even more physically accurate results (Figure 3.12).

$$\begin{aligned} L_A(x,l) &= e^{-\frac{x}{l}} \\ L_{flux}(f,\phi,x) &= \frac{f}{2\pi(1-cos\phi)x^2} \\ Total\_Attenuation(x_1,x_2,l,f,\phi) &= L_A(x_1,l)L_{flux}(f,\phi,x_1)L_A(x_2,l) \end{aligned} \tag{3.1}$$

where:

L_A = *light's attenuation*
L_{flux} = *light's luminous power*
x_1 = *distance between light's source and scattering point*
x_2 = *distance between scattering point and viewer*
l = *medium's optical thickness*
f = *light's flux*
ϕ = *half angle of light's cone*

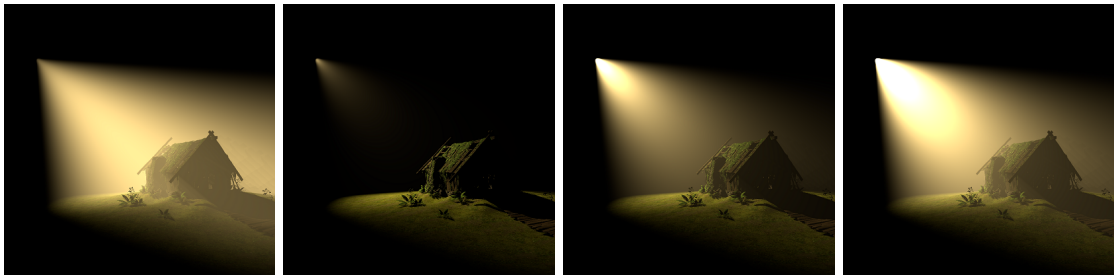

FIGURE 3.12: Left: No light attenuation. Middle-Left: Light attenuation, optical thickness = 7.5. Middle-Right: Light attenuation, optical thickness = 15. Right: Light attenuation, optical thickness = 22.5. (Images rendered using XEngine [PV13])

The changes in pseudo-code are shown in Algorithm 2.

Algorithm 2 Volumetric Lighting Fragment Shader

1: $trace\_start \leftarrow camera\_position$
2: Reconstruct $trace\_end$ using the scene's depth buffer
3: $light\_factor \leftarrow 0$
4: **for all** samples on trace **do**
5: Estimate $shadow\_factor$ using the shadow map (nearest shadows $\rightarrow$ 0 or 1)
6: Dynamically update phase function coefficients for phase function animation
7: Use appointed function to calculate phase_function (Equations 2.2, 2.3, 2.4, 2.5)
8: Calculate $total\_light\_attenuation$ using Equation 3.1
9: $light\_factor \leftarrow light\_factor + step\_distance \cdot spot\_effect \cdot shadow\_factor \cdot phase\_function \cdot total\_light\_attenuation$
10: **end for**
11: $out\_color \leftarrow light\_factor \cdot light\_color$

3.2.5 Quality and Performance Optimization

So far, we have managed to construct an adequate approximation of the real-world light scattering phenomenon. XEngine provides a functionality to measure the performance of any active effect in our virtual world. As shown in Figure 3.13, to achieve a satisfactory result, the effect has a substantial impact on our hardware's real-time rendering capability. The following subsections describe the way to accomplish the optimal balance between quality and performance, by the optimization of specific items and procedures utilized in the implementation.

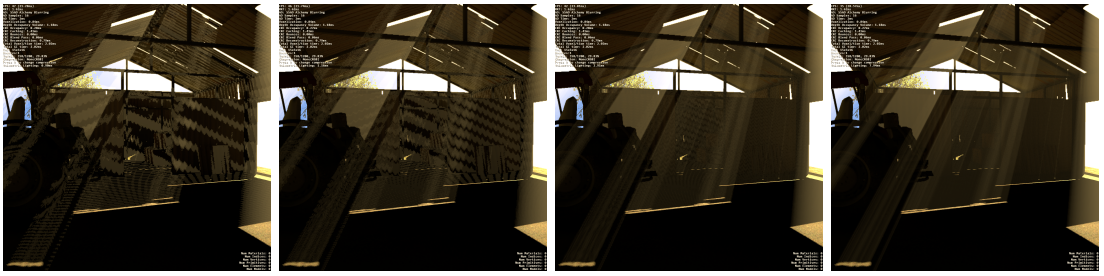

FIGURE 3.13: Left: 100 samples per sampling ray [FPS:47 Effect time:0.9ms] (substandard). Middle-Left: 200 samples per sampling ray [FPS:46 Effect time:1.16ms] (substandard). Middle-Right: 500 samples per sampling ray [FPS:42 Effect time:2.91ms] (adequate - there is still visible banding). Right: 1000 samples per sampling ray [FPS:35 Effect time:7.94ms] (perfect result). (Images rendered using XEngine [PV13])

Buffer Optimization

As noted in XEngine's utilized assets, in the post process effects segment (3.2.1), the texture containing the information of the volumetric lighting effect, is stored individually inside a particular buffer, before entering a stage where it is blended to produce the final result. We can make full use of this feature by adjusting the volumetric lighting buffer to a lower resolution, thus reducing the computations per frame required to create the effect. Figure 3.14 shows the buffer size influence on the implementation's performance. Our desired objective is the balance

between quality and performance, therefore, a buffer reduction of 50% is probably the ideal choice.

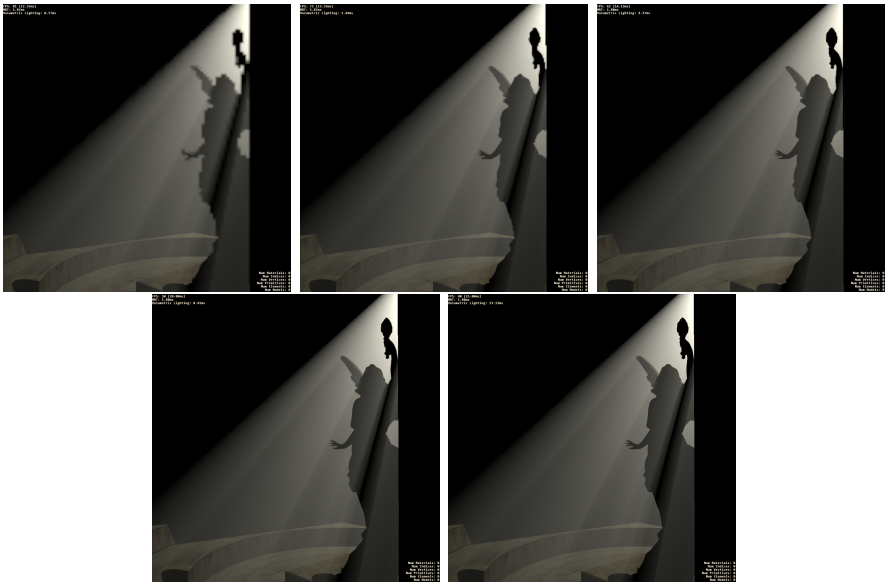

FIGURE 3.14: Buffer reductions: Top-Left: 90% [FPS:81 Effect time:0.57ms]. Top-Middle: 75% [FPS:75 Effect time:1.44ms]. Top-Right: 50% [FPS:62 Effect time:4.37ms]. Bottom-Left: 25% [FPS:50 Effect time:8.41ms]. Bottom-Right: 0% [FPS:40 Effect time:13.13ms]. (Images rendered using XEngine [PV13])

In the two previous examples, we saw that keeping the number of traced rays low (image-space resolution) and the per-ray sample count low, both incur a substantial degradation of image quality, with most notable the banding artifact of the latter case (Figure 3.14). Banding is also evident in locations where the volumes are quite thin (see Figure 3.15), when image-space sub-sampling is utilized. To combat such implications, we create jittering on the sampling rays. This process generates noise on the volumes, by introducing variability to the position on the trace that each individual sample has, based on their UV-coordinates and the engine's run time. Consequently, flaws originated by both the buffer's resolution reduction and small amount of per-ray samples are masked, however note that the noise may be distinguishable.

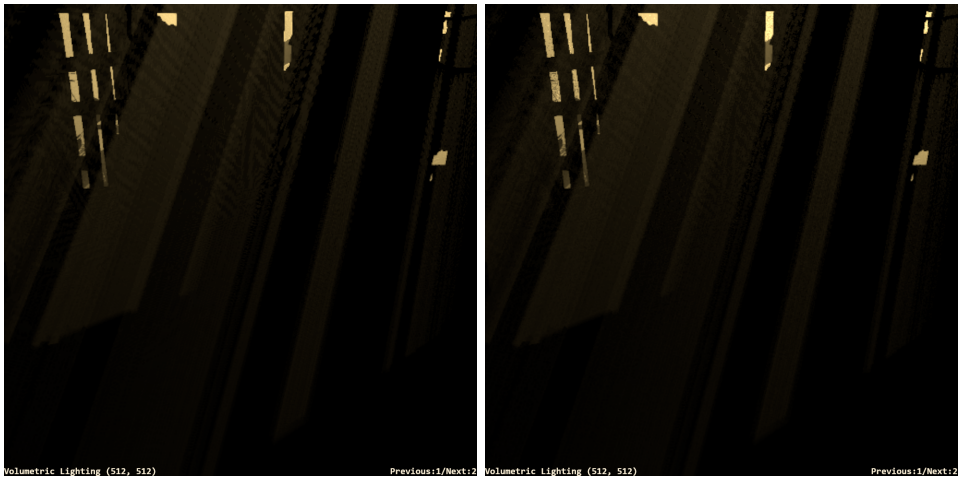

FIGURE 3.15: Left: Before jittering. Right: After jittering. (Images rendered using XEngine [PV13])

At the beginning of this section, we mentioned that the effect is initially stored at the volumetric lighting buffer and then enters the blending phase. The way the buffers combine is by additive blending, where the final texture appearing on the screen is the result of the addition of each of the buffers' corresponding UV-coordinated texel values. In 2.1.4, we discussed that by utilizing a certain blurring algorithm, we can conceal visible noise in our implementation. So, for a final optimization on the buffer, we apply the Gaussian blur image processing technique (Algorithm 3) on the volumetric lighting effect's texture before the additive blending is performed. The results are shown in Figure 3.16.

Algorithm 3 Volumetric Lighting Blend Fragment Shader - Gaussian blur (single pass, 5x5, s = 1)

1: $weights \leftarrow$ (0.003765, 0.015019, 0.023792, 0.015019, 0.003765, 0.015019, 0.059912, 0.094907, 0.059912, 0.015019, 0.023792, 0.094907, 0.150342, 0.094907, 0.023792, 0.015019, 0.059912, 0.094907, 0.059912, 0.015019, 0.003765, 0.015019, 0.023792, 0.015019, 0.003765)
2: Get *scene_color* from the other buffer using current_texture_coordinate
3: $weight\_counter \leftarrow 0$
4: $light\_volume\_color \leftarrow 0$
5: **for** $i \leftarrow -2$ to 2 **do**
6: **for** $j \leftarrow -2$ to 2 **do**
7: $light\_volume\_color \leftarrow light\_volume\_color + get\_vl\_color(current\_texture\_coordinate + (i, j)) \cdot weights[weight\_counter]$
8: $weight\_counter \leftarrow weight\_counter + 1$
9: **end for**
10: **end for**
11: $out\_color \leftarrow scene\_color + light\_volume\_color$

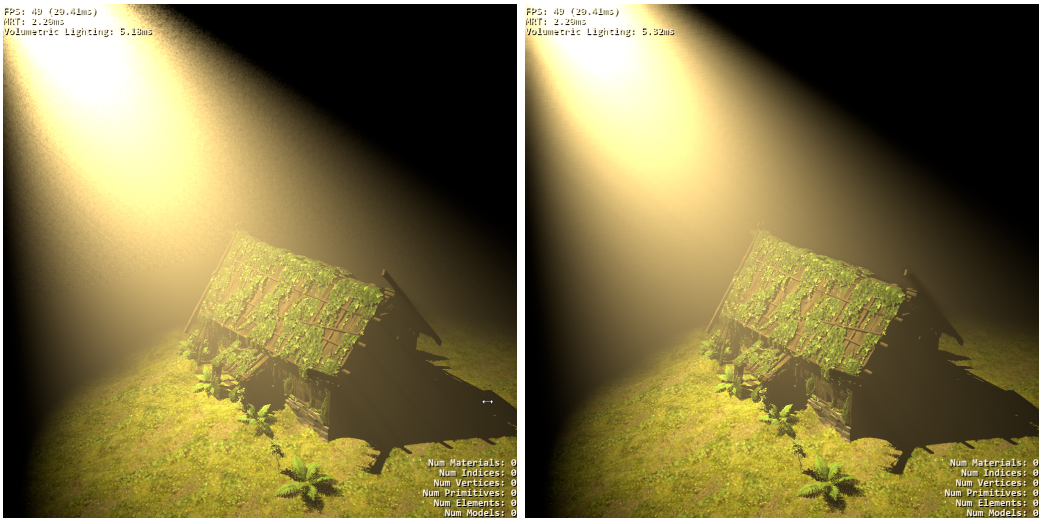

FIGURE 3.16: Left: Visible noise before Gaussian Blur. Right: Most of noise is hidden after Gaussian Blur. (Images rendered using XEngine [PV13])

Sampling Optimization

So far, for sampling we have followed the "naive" approach of tracing every pixel up to its first intersection point (as described in Section 3.2.3). This method may be adequate for many simple examples, however, its simplistic nature has many disadvantages. First of all, it occasionally creates lengthy traces that extend all the way up to the furthest point in their direction, currently rendered in the virtual world. That affects the quality of the light volumes as samples are so

widely dispersed into space that their distribution does not reflect the importance of the sampled medium portion, in turn causing severe banding in extreme cases. Moreover, traces cover area that is useless and can be easily skipped, consuming samples that could be utilized for the light volumes. While every one of the above problems could be solved by dramatically increasing the number of samples, the detrimental effect this action would have in the implementation's performance makes it an unacceptable solution.

One way to tackle those issues with inconsiderable computational cost, is to limit the sampling interval only inside the light volume. As mentioned before, the spot-light emits a pyramid of light in the scene. Our goal is to isolate those traces inside the pyramid and perform sampling exclusively within this space. Obviously, this pyramid structure can be divided into six triangles. Therefore, this becomes a ray-triangle intersection problem. Each sampling ray can intersect with zero, one or two triangles of this pyramid structure. Note that to define the amount of intersections a sampling ray has with the pyramid, we use its starting location and direction vector. In other words, for those geometric calculations, collisions with the pyramid can happen even if there are physical obstacles obstructing them. We should account for such occasions, by comparing the ray's initial starting and ending points to those intersection points, to avoid any impossible scenarios. In case of zero intersections, we simply skip the current ray. One intersection means that the tracing begins from inside the light pyramid. Finally, two intersections indicate that the sampling ray begins outside the pyramid and probably (as mentioned before, it can also be obstructed) passes through it. The way we clip the tracing interval according to the number of intersections is described in more detail in Algorithm 4. The results are shown in Figure 3.17.

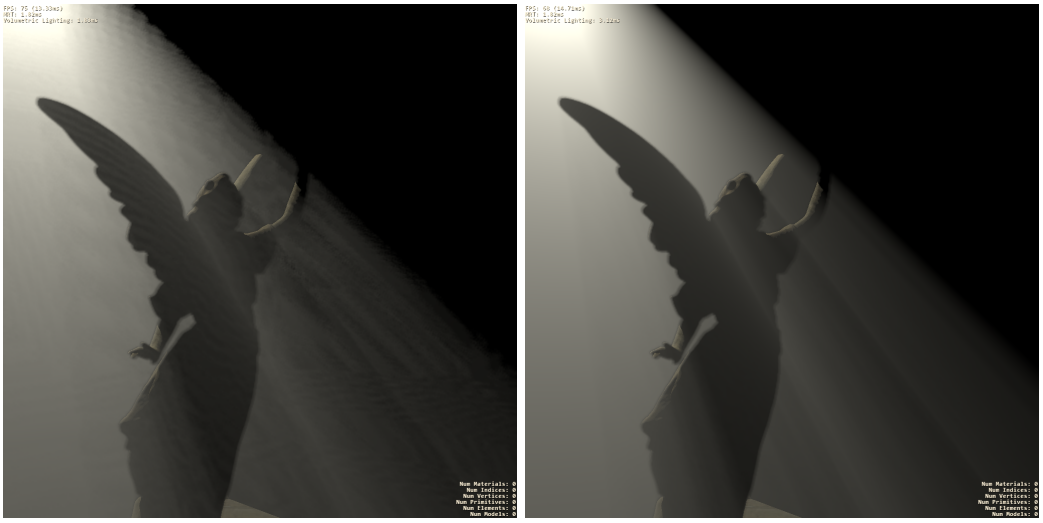

FIGURE 3.17: Left: Before clipping, 50 samples per sampling ray [FPS:75 Effect time:1.88ms]. Right: After clipping, 50 samples per sampling ray [FPS:68 Effect time:3.12ms]. (Images rendered using XEngine [PV13])

Algorithm 4 Clip Sampling Ray inside Light Volume

Calculate *number_of_intersections* and return the *first_intersection_point* and *last_intersection_point* // see Figure 3.18 for each case
if *number_of_intersections* = 0 **then**
  return FALSE // case A
else if *number_of_intersections* = 1 **then**
  if *first_intersection_point* is closer than *trace_end* **then**
    trace_end ← *first_intersection_point*
    return TRUE // case B
  else
    return TRUE // case C
  end if
else if *number_of_intersections* = 2 **then**
  if *last_intersection_point* is closer than *trace_end* **then**
    trace_start ← *first_intersection_point*
    trace_end ← *last_intersection_point*
    return TRUE // case D
  else if *first_intersection_point* is closer than *trace_end* AND *last_intersection_point* is further away than *trace_end* **then**
    trace_end ← *last_intersection_point*
    return TRUE // case E
  else if *first_intersection_point* AND *last_intersection_point* are both further away than *trace_end* **then**
    return FALSE // case F
  end if
end if

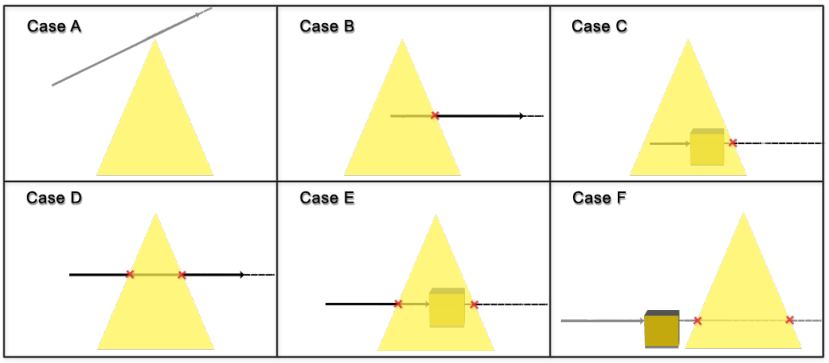

FIGURE 3.18: All cases of sampling ray clipping. Case A: Zero intersections - no clipping and no sampling. Case B: One intersection - intersection point is new trace_end. Case C: One intersection - no need for clipping. Case D: Two intersections - first and second intersection points are the new trace_start and trace_end points respectively. Case E: Two intersections - first intersection is new trace_start. Case F: Two intersections - no clipping and no sampling.

In the previous paragraph, we implemented a way to clip the sampling rays, to avoid collecting purposeless samples around the space of the light volume. One more optimization that can be applied on the sampling procedure concerns the valid samples in the clipped interval. As we already mentioned, those volumes can be complex and ever-changing. There may be dispersed parts inside those volumes, totally submersed in shadow. If a sample's contribution to the result is zero, i.e. is unlit, we simply skip it. Even though, this may appear as an insignificant step for optimization, the results shown in Figure 3.19 prove otherwise.

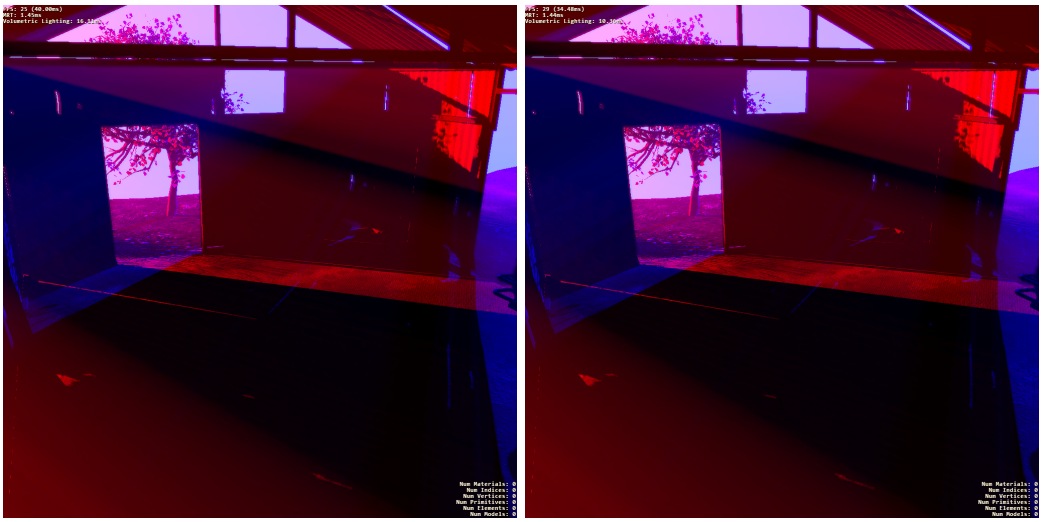

FIGURE 3.19: Sampling shadow skipping (same visual results) Left: Before unlit sample skipping [FPS:25 Effect time:16.11ms]. Right: After unlit sample skipping [FPS:29 Effect time:10.36ms]. (Images rendered using XEngine [PV13])

Shadow Map Optimization

A final quality optimization, involves the process in which samples contribute to the result. In 3.2.3, we discussed that the way to determine whether a sample is illuminated or not, is to compare its location with the shadow map. As specified in Section 2.1.5, the shadow map's low resolution can cause visible aliasing along the shadow's edges. In the volumetric lighting effect, this can translate to aliased light shafts.

Therefore, by applying the shadow blur with percentage closer filtering during the comparison process of the sample's position with the shadow map, we can estimate each sample's percentage contribution to the final intensity of the light shaft. This way, we exploit PCF to anti-alias the volumetric effect transitions. Similar to how this algorithm normally operates, it creates a "gradient" (3-dimensional) of different intensities along the light volume's borders, smoothing out and anti-aliasing the final result (Figure 3.20).

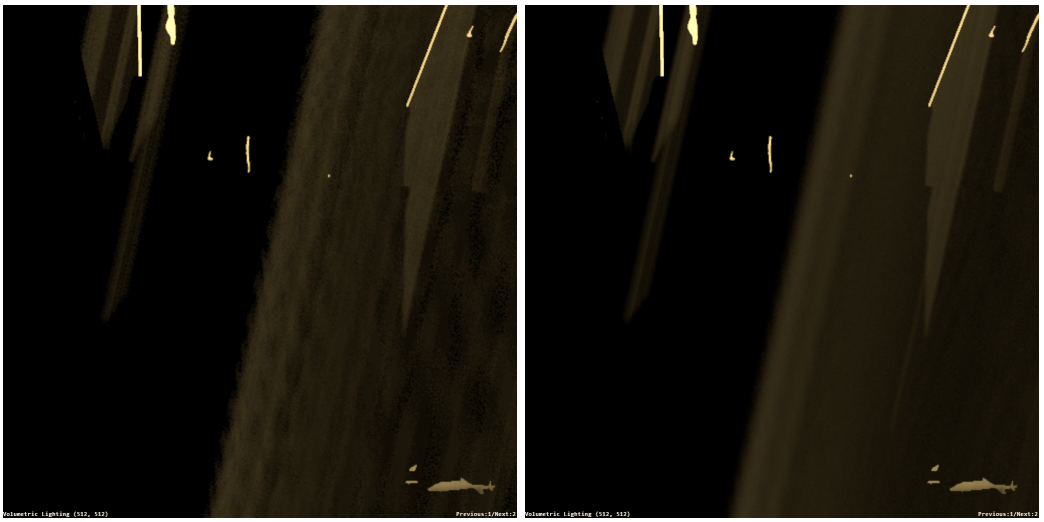

FIGURE 3.20: Left: Before percentage closer filtering [FPS:44 Effect time:2.04ms]. Right: After percentage closer filtering [FPS:41 Effect time:3.72ms]. (Images rendered using XEngine [PV13])

Algorithm 5 Volumetric Lighting Fragment Shader (Final)

1: $trace\_start \leftarrow camera\_position$
2: Reconstruct $trace\_end$ using the scene's depth buffer
3: $light\_factor \leftarrow 0$
4: Use Algorithm 4 to clip the sampling ray and return $sampling\_required$
5: **if** $sampling\_required$ = TRUE **then**
6: **for all** samples on trace **do**
7: Jitter the position of the sample on the trace to produce noise
8: Estimate $shadow\_factor$ using the shadow map (pcf shadows → 0 to 1)
9: **if** $shadow\_factor > 0$ **then**
10: Dynamically update phase function coefficients for phase function animation
11: Use appointed function to calculate phase_function (Equations 2.2, 2.3, 2.4, 2.5)
12: Calculate $total\_light\_attenuation$ using Equation 3.1
13: $light\_factor \leftarrow light\_factor + step\_distance \cdot spot\_effect \cdot shadow\_factor \cdot phase\_function \cdot total\_light\_attenuation$
14: **end if**
15: **end for**
16: **end if**
17: $out\_color \leftarrow light\_factor \cdot light\_color$

3.3 Environment Mapping Implementation

Our approach to environment mapping differs from the one of volumetric lighting. For the latter, the code we presented in its implementation was purely executed on the GPU, i.e. graphics processing unit, side of our hardware. For environment mapping, the use of the computer's

CPU, i.e. central processing unit, is also important. This is due to the following reason. Each environment inside the virtual world is stored in a form of a cube map of textures and to construct a scene object's environment-mapped reflection, we simply draw the correct part of this skybox on it. However, to account for the distortions caused by scattered reflections on rough surfaces, we need to either perform some sort of sophisticated randomized sampling of this cube map on the spot (meaning for every frame and for each pixel that is a part of those surfaces), or we could simply apply the method discussed in 2.2.3 for every aforementioned textures, to generate various new ones corresponding to specified roughness levels. Then, according to the surface's roughness, we interpolate between those textures to produce the desired reflection. The second option is preferable, as this pre-filtering process is performed only once in the CPU side of the hardware and the resulting textures can be reemployed for any times of the effect's execution.

In environment mapping, we begin with an overview of the engine's components used for the construction of the effect, including extensions that we performed on them, to permit the CPU utilization that is required as mentioned in the previous paragraph (Section 3.3.1). Then, we construct the skybox data structure that hosts the scene's environment information (Section 3.3.2). Next we perform the pre-filtering process, to account for the levels of irradiance for the environment map's textures based on several hard-coded roughness values (Section 3.3.3). Finally, on the graphics card side of the implementation, we provide an explanation of how BRDF is finally integrated, to account for the surfaces material properties (Section 3.3.4).

3.3.1 XEngine's Dependent Components and Extensions

Apart from some of the assisting assets described in Section 3.2.1 that facilitate with the construction of environment mapping, similarly to the volumetric lighting implementation, XEngine provides a few additional basic functions that can both be extended and utilized. Subsequently, we specify how we employed already discussed and newly mentioned features of this engine to achieve the final desired result.

Ambient Pass

Environment mapping is responsible for the reflections of the scene's ambient information on surfaces. Contrary to volumetric lighting, which is introduced to the engine as a new post process technique, environment mapping is integrated as a code extension to the ambient lighting technique's class. This technique was initially created to perform the ambient render pass which, based on several of its shader parameters, produced an imitation of the real world environment lighting.

Nodes

XEngine's SceneGraph contains a Node3D class for parameterization, dynamic events, messaging and grouping which acts as a predecessor to the graph's nodes. A few of them, concern essential elements such as the camera, the user, events e.t.c. Some effects that could possibly require real time parameterization are also created as nodes, e.g. lights, tone-mapping and environment mapping. Each node possesses its own exclusive attributes and a range of important shared ones, inherited from its predecessors. Those characteristics can be updated anytime, by

an event caused by a trigger, or by a message manually sent by the user. For example, in environment mapping we could manually alternate between different sky-boxes/environment maps, or we could even change them dynamically, by proximity triggers when the user enters or exits predetermined areas. For this implementation, we have created two node classes. The first is the EnvironmentMap3D class, which is responsible for the skybox's cube map texture specification and loading. The second one is the SkyMap3D class, which acts as the indicator of the current active environment map node, for the purposes of having multiple available environments for the scene.

Texture Manager

The way this engine handles its textures operations, is by the use of the FreeImage library [Ima03]. The Texture class is able to perform functions calls to the FreeImage's API to load, store and delete most of the types of image data. It is also able to upload each texture to the GPU, in order to be sampled by the engine's shaders. The main class responsible for the texture handling is the TextureManager. This class works as an interface to register, generate and unload the specified texture data at any point of the engine's execution. In Section 3.3, we mentioned that for our construction of the effect, we use several cube maps to represent the skybox information and the different irradiance levels. Therefore, in order to achieve that, we extend the capabilities of both of those classes to include the loading, deletion and GPU uploading of cube map textures.

3.3.2 Environment Cube Map Construction

As stated earlier, the way we selected to represent the environment or ambient information of the virtual scene is by a data structure, typically called a skybox. This is a cube map, which is a 3-dimensional box of 6 textures where each of them is faced inward. The textures we use for this implementation are acquired from [Mak11] for free and follow a specific naming convention displayed in Figure 3.21.

FIGURE 3.21: Orientation and naming conventions of the skybox textures.

To upload this data structure to the GPU for rendering, we first load the texture in the engine by employing its texture loader functionality we mentioned in the previous section. Then, we

adjust every face's orientation to match the one shown in 3.21. Finally we utilize OpenGL's cube map texture configuration to create the unique id that points to the skybox data that is later on sampled from the graphics card to display the final result shown in Figure 3.22.

FIGURE 3.22: Skyboxes rendered in XEngine. (Images rendered using XEngine [PV13])

3.3.3 Mip-Mapping Based on Roughness

Briefly, Mip-Mapping is the sequential storing pre-filtered versions of a texture, in order to improve rendering time and reduce visible aliasing artifacts. The dimensions of each level in the mip-map is lower by a power of two from the previous level (Figure 3.23).

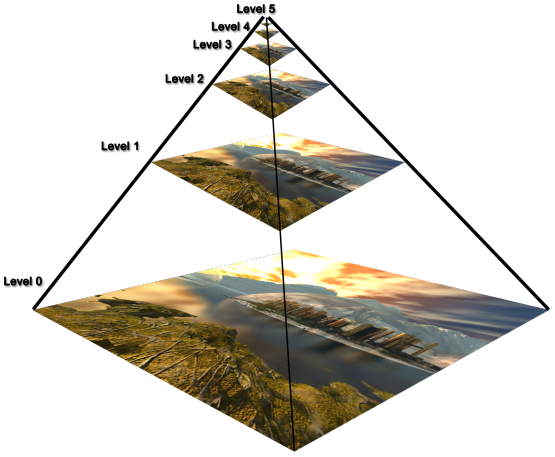

FIGURE 3.23: Mip-Mapping illustrative example.

As mentioned in 3.3, to reduce this implementation's computational cost resulting from calculating a surface's ambient reflection, we generate and store multiple textures according to the information of the environment's cube map data. Those are divided in levels of detail, also known as LOD in mip-mapping, shown in Figure 3.24. However, instead of simply down-scaling the information by a factor of two, the mip levels are here carefully crafted to represent sampled versions of the original environment map at distinct and non-linearly spaced roughness levels. Each mip map is constructed by separately sampling the original environment map using an importance function discussed below and not by averaging the values of the lower-level mip map. The importance function for the sampling directions distribution over the hemisphere is driven by the micro-facet distribution function of the specular BRDF we discussed in 2.2.2.

	LOD Dimensions	Roughness Value	Beckmann BRDF Lobe Aperture	Sampling Disk Radius (tanθ)
Level 0 (Environment Map)	1024 x 1024	0.0	0	0°
Level 1 (Irradiance Map)	512 x 512	0.01	0.01	1°
Level 2 (Irradiance Map)	256 x 256	0.05	0.1	6°
Level 3 (Irradiance Map)	128 x 128	0.1	0.21	12°
Level 4 (Irradiance Map)	64 x 64	0.2	0.42	23°
Level 5 (Irradiance Map)	32 x 32	0.3	0.62	32°
Level 6 (Irradiance Map)	16 x 16	0.4	0.83	40°
Level 7 (Irradiance Map)	8 x 8	0.5	1.07	47°
Level 8 (Irradiance Map)	4 x 4	0.6	1.27	52°
Level 9 (Irradiance Map)	2 x 2	0.7	1.48	56°
Level 10 (Irradiance Map)	1 x 1	1.0	2.14	65°

FIGURE 3.24: First column: The resolution of each level. Third and Second columns: The roughness - Beckmann BRDF lobe aperture correspondence was estimated using the graph depictions from [DS13]. Fourth column: The sampling cone's radius for each level of irradiance (see Figure 3.25 for its calculation).

With OpenGL, we could perform mip-mapping on the cube map to construct a data structure that contains both the primary textures of the skybox and all of the different irradiance maps. With its first level, we could depict the scene's background and perfect reflections on highly specular surfaces and the rest of the levels could be interpolated to achieve most of the possible reflections, based on the surface material properties.

Before Mip-Mapping the skybox's cube map though, we first need to generate the aforementioned levels based on the primary environment textures. Each of those levels contain their own cube map/irradiance map, with the necessary corresponding textures and as stated in the previous paragraph, their resolution is lower by a power of two from the previous level (Equation 3.2).

$$res(l) = \frac{r}{2^l}; \tag{3.2}$$

where:

l = *current level*
r = *primary texture's resolution*

As we discussed in Section 2.2.3, to calculate the different irradiance levels of the environment's textures we use the Monte Carlo Integration technique to estimate the contribution of all the directions, based on a number of random samples and their statistical mean. For each texel of every generated texture (i.e. for each irradiance integral hemisphere orientation) and for all the levels required, we perform cosine weighted and uniformly randomized cone sampling on the main environment map (the skybox) to gather color information. Instead of crafting a complex distribution based on the BRDF response, we opted to find the optimal window over the hemisphere in which the BRDF has a non-negligible contribution, and perform uniform or cosine weighted sampling in it. In other words, we constrain the sampling within a cone centered at the normal hemisphere direction and spanning the part of the hemisphere with significant radiance contribution. This way, we decouple the specific BRDF from the sampling process, while not wasting samples on a naive cosine sampling of the entire hemisphere, when this is not

contributing to the integral. The cone's radius for each level is different and depends on the previously mentioned hard-coded roughness values (see Figure 3.24).

To estimate the irradiance of each texel, we need to multiply each sample's contribution to its inverse probability. Uniform cone sampling can be reduced to a problem of uniform disk sampling (see [Wei17a] and [Wei17b] for more technical details) as the total number of random direction vectors that begin from the cone's apex and are confined inside it are equal to its disk base's number of points. Therefore, the probability is the same for every sample and is equal to $1/\pi r^2$, where r is the radius of the cone's disk base. As shown in Figure 3.25, to simulate the effect that the surface material properties have on the level of diffusion of each particular irradiance map, the disk's radius r is equal to $tan\theta$ where θ is the lobe "aperture" (empirically - this aperture defines the angle range of the BRDF micro-facets) of the Beckmann BRDF for each corresponding irradiance map/roughness level (as previously depicted in Figure 3.24). To create all the different environment map levels we use the following Algorithm 6.

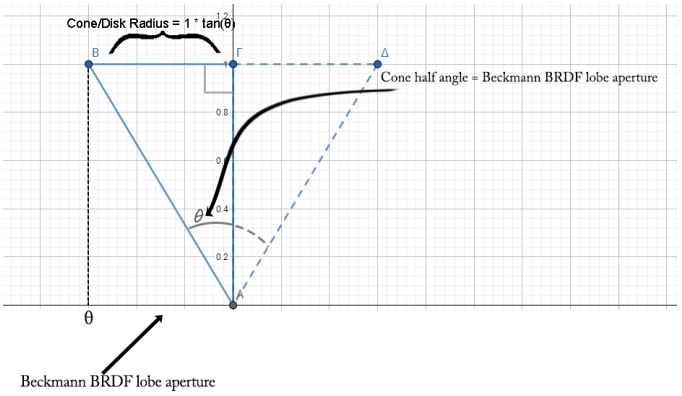

FIGURE 3.25: Cone radius $r = 1 \cdot tan\theta$ where theta is equal to the Beckmann BRDF's lobe aperture for each specific LOD (see Figure 3.24)

Algorithm 6 Irradiance Level Pre-filtering

for all *levels* **do**
  for all *cube_faces* **do**
    Calculate *level_dimensions* based on Equation 3.2
    cone_radius value is based on current LOD (see Figure 3.24)
    $inv\_pdf \leftarrow \pi \cdot cone\_radius^2$
    for all *texels* of the *cube_face* **do**
      for $i \leftarrow 0$ to *number_of_samples* **do**
        Get a new cosine weighted color sample from the environment map using randomized cone and add it to the texel's *color_sum*
      end for
    end for
    $color\_sum \leftarrow color\_sum/number\_of\_samples \cdot inv\_pdf$
    Write *color_sum* to the current *texel* of the generated irradiance texture
  end for
end for

This part of the environment mapping implementation is probably the most computationally expensive and is performed in the CPU side of the hardware. Thus, we could use multi-threading to decrease the total generation time. There are many ways to parallelize this problem, however for this implementation we chose to split each texture's data into subsets that are handled from each of the available cores in the computer's CPU. The application of this optimization on the generation of the irradiance maps resulted in a reduction of 80% of the initial required processing time (example of test case for 200 random samples: 150 seconds → 30 seconds).

So far, we have managed to efficiently produce all the essential textures for the replication of the greatest part of the environment reflections. The number of samples greatly influences the time of generation and quality of the textures, however, as the resolution at each higher levels drops to lesser values, this impact is diminished rapidly. Moreover, as seen in Figure 3.24, the angle of the sampling cone increases accordingly with each next level, demanding more samples for the method to be precise. Therefore, the total number of samples should increase with each next LOD.

3.3.4 Surface Reflections and integration of the BRDF

For the main GPU utilization part of the environment mapping technique, as a code base, we begin with a construction of perfect or mirror reflections. As described in Section 2.2.1, highly specular reflections have equal incident and reflection angle. Results are depicted in Figure 3.26.

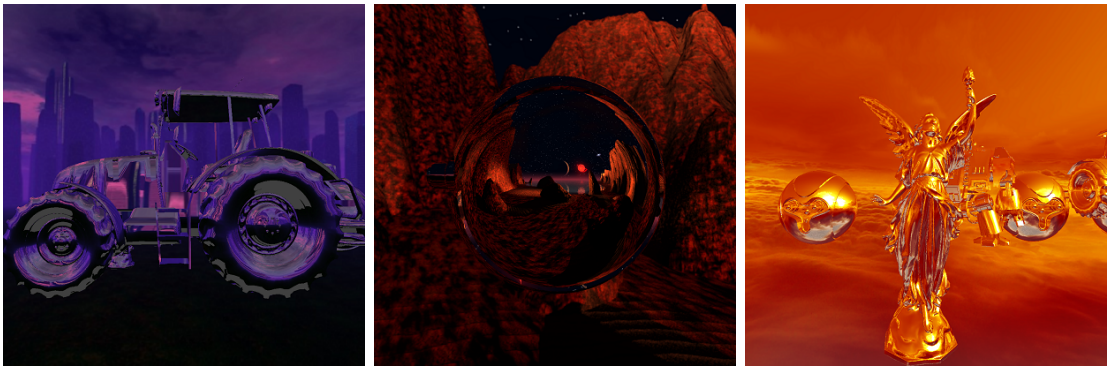

FIGURE 3.26: Mirror reflections on the surfaces of the scene's objects. (Images rendered using XEngine [PV13])

We have already described that real world environment reflections are comprised from both specular and glossy reflections. The level of diffusion of light on rough surfaces is based on their material properties. In the previous section, we managed to generate different mip levels of textures that are stored as irradiance/roughness levels, in the same data structure that contains the primary environment map. Those LOD are divided, depending on the surface's roughness. Therefore, to accurately represent glossy reflections we use the same process that was applied for specular reflections, but instead of sampling the main skybox/environment map we interpolate between the other LOD according to the surface's roughness value.

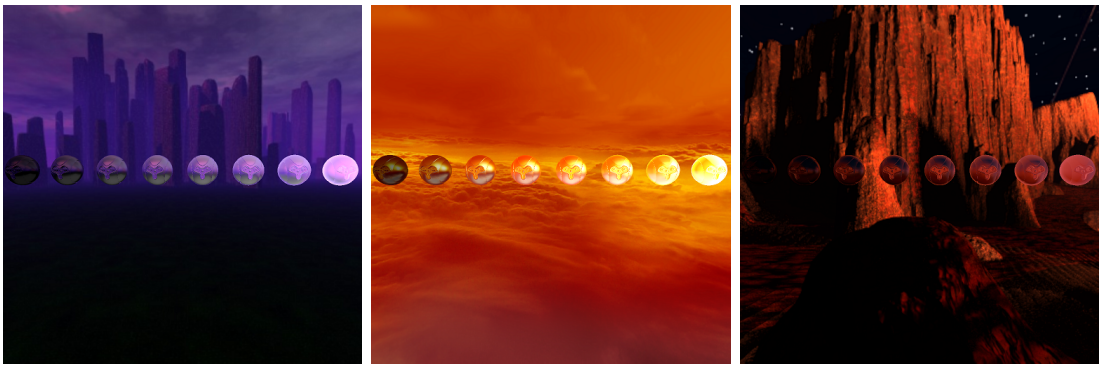

FIGURE 3.27: Rough reflections on the surfaces of the scene's objects. Notice that there is an energy imbalance since the material surfaces have not yet been accounted for. (Images rendered using XEngine [PV13])

In Figure 3.27 we notice that there is no actual conservation of energy on the surfaces of the objects. This is due to the fact that, so far, we have only accounted, through the pre-filtering process we described in 3.3.3, for different levels of irradiance on surfaces originating from environment lighting. We have not yet taken into consideration how material properties at each part of the surface actually affect the energy distribution. To achieve this we integrate the method described in 2.2.2 into this implementation. Algorithm 7 is applied and produces the results depicted in Figures 3.28 and 3.29.

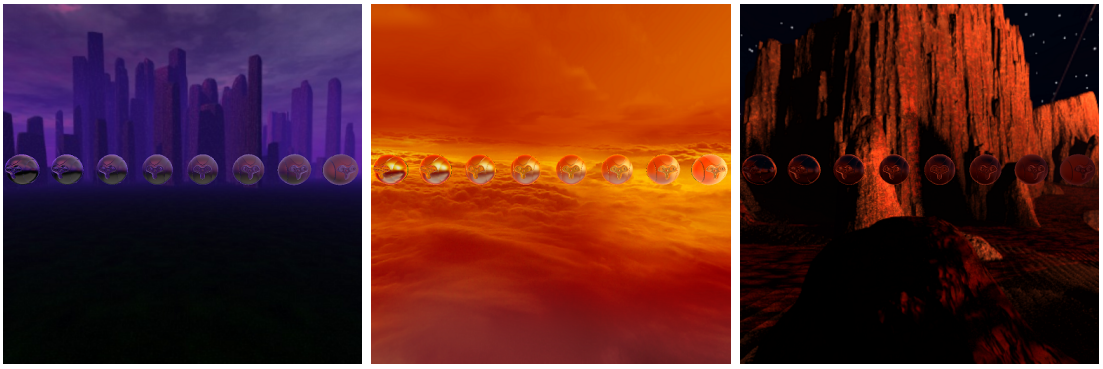

FIGURE 3.28: Energy conservation is correct after the BRDF integration. (Images rendered using XEngine [PV13])

FIGURE 3.29: More complicated geometry and surface roughness distribution. (Images rendered using XEngine [PV13])

Algorithm 7 Environment Mapping Fragment Shader (Final)

1: **if** $current\_dept = 1$ **then**
2: Use the $cube\_map\_color[0]$ values as the out_color to display the skybox
3: **else**
4: Interpolate between the roughness hard-coded values to calculate mip_level
5: Utilize Algorithm 8 to calculate $microfacet\_brdf$
6: $out\_color \leftarrow cube\_map\_color[mip\_level] \cdot microfacet\_brdf$
7: **end if**

Algorithm 8 Micro-facet BRDF

1: Calculate $distribution\_function$ using Equation 2.10
2: Calculate $fresnel$ using Equation 2.11
3: Calculate $geometric\_shadowing$ using Equation 2.12
4: Calculate $specular\_brdf$ using $distribution\_function$, $fresnel$ and $geometric\_shadowing$ in Equation 2.9
5: Calculate $diffuse\_brdf$ using Equation 2.13
6: return $specular\_brdf + diffuse\_brdf$

Chapter 4

Evaluation and Future Improvements

In this work, two lighting techniques were implemented and optimized using various mathematical concepts and procedures. Note that the methods we have followed so far have not been extensively assessed. Consequently, in this chapter, we evaluate each specific component of the overall process in order to determine whether its impact on the balance between performance and quality is positive. Finally, we examine future improvements and extensions that could be applied to further optimize and enhance both of the effects.

Volumetric Lighting Evaluation

Volumetric lighting implementation work-flow evaluation (100 samples per traced ray):

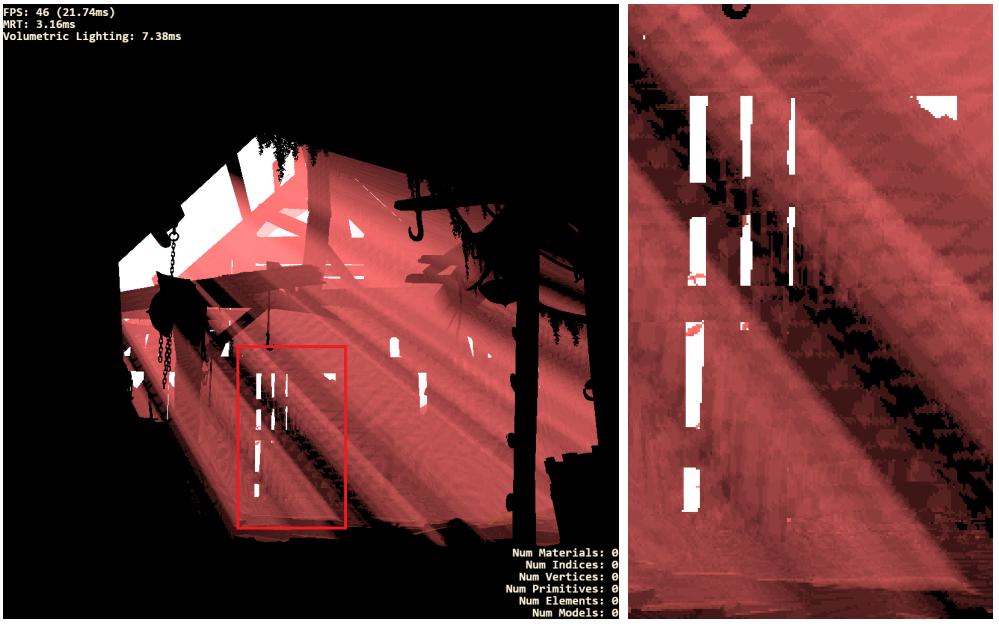

FIGURE 4.1: Geometrical distinction, step distance and spot effect consideration.

Figure 4.1: Even though the results are physically incorrect, this stage is the stepping stone to construct the effect of volumetric lighting. Every method utilized here is essential for the formation of pyramidal or conical light volumes and the normalization of samples' contribution (see Figure 3.3).

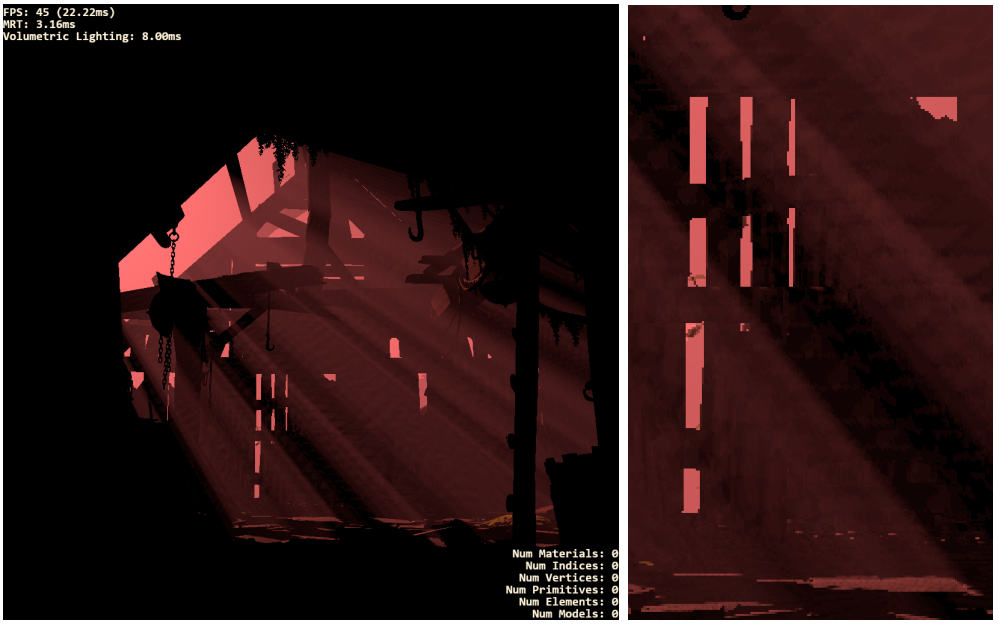

FIGURE 4.2: Phase functions.

Figure 4.2: As discussed in Section 2.1.2, every phase function has its own, specific purpose. Although all of those functions had roughly the same negligible computational cost, their contribution to the effect's quality was extremely positive as they could be utilized to approximate a great part of light scattering phenomena.

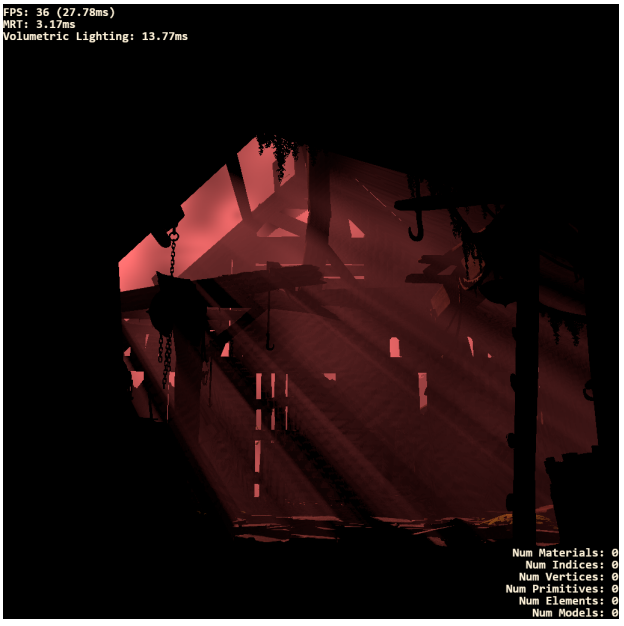

FIGURE 4.3: Phase function animation.

Figure 4.3: Animation by adjusting the phase function's scattering coefficients did not produce very realistic results. It had an overall negative influence on the balance between performance and quality. Especially considering the fact that it caused almost 50% increase in the time required to compute the effect.

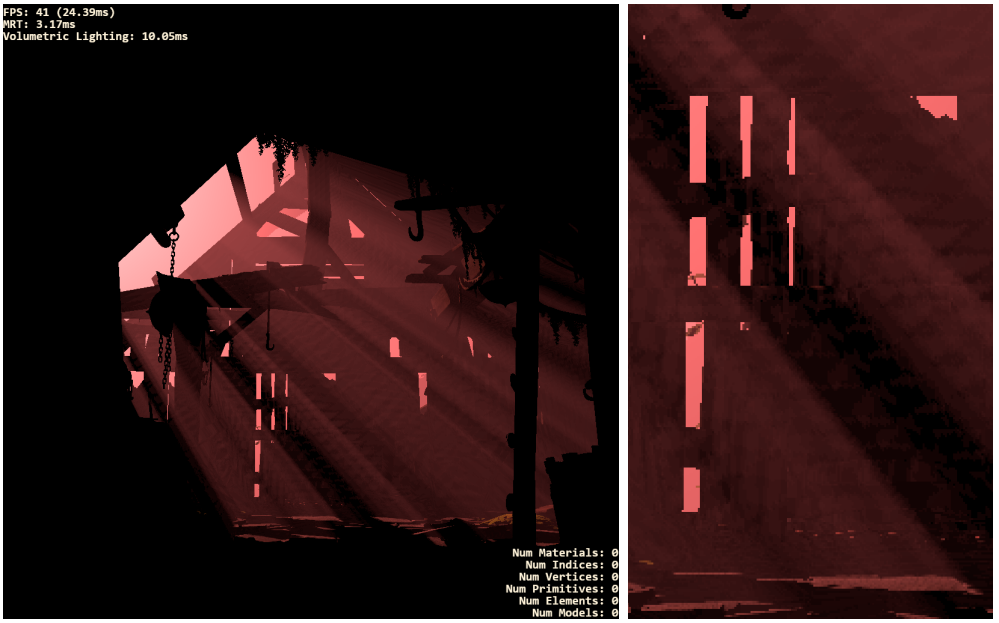

FIGURE 4.4: Light attenuation.

Figure 4.4: Similarly to phase functions, light attenuation is also necessary to make the effect more physically correct. Because of its complicated mathematical equation (Equation 2.7), it causes an increase of 25% on the required time.

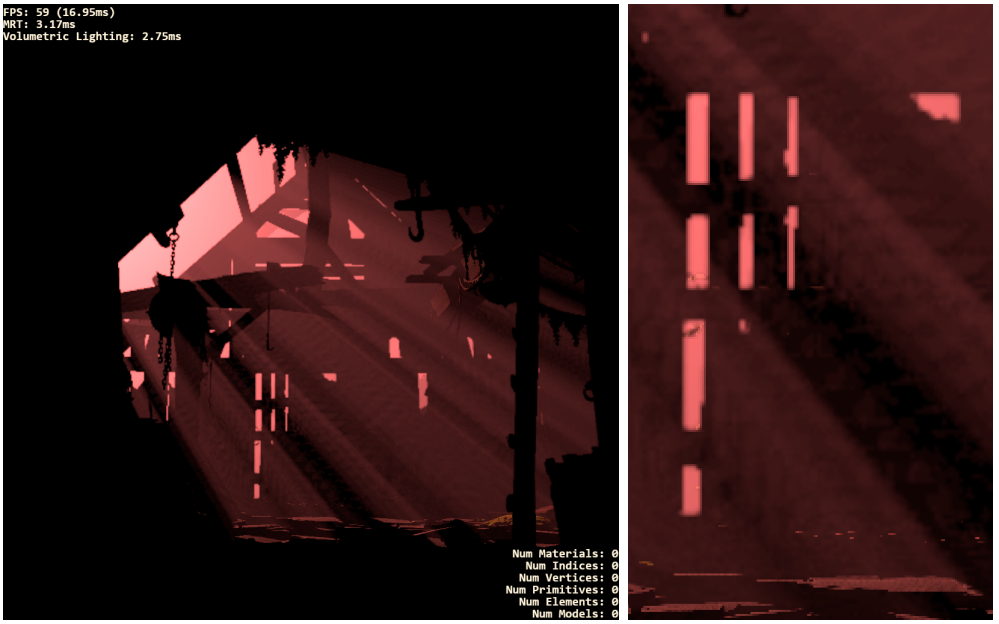

FIGURE 4.5: Buffer reduction.

Figure 4.5: As explained in Section 3.2.5 and as shown in Figure 3.14, a buffer reduction of 50% is the optimal choice. We can verify this in this example too as we notice approximately 75% reduction of the effect's computation time with little impact on the quality.

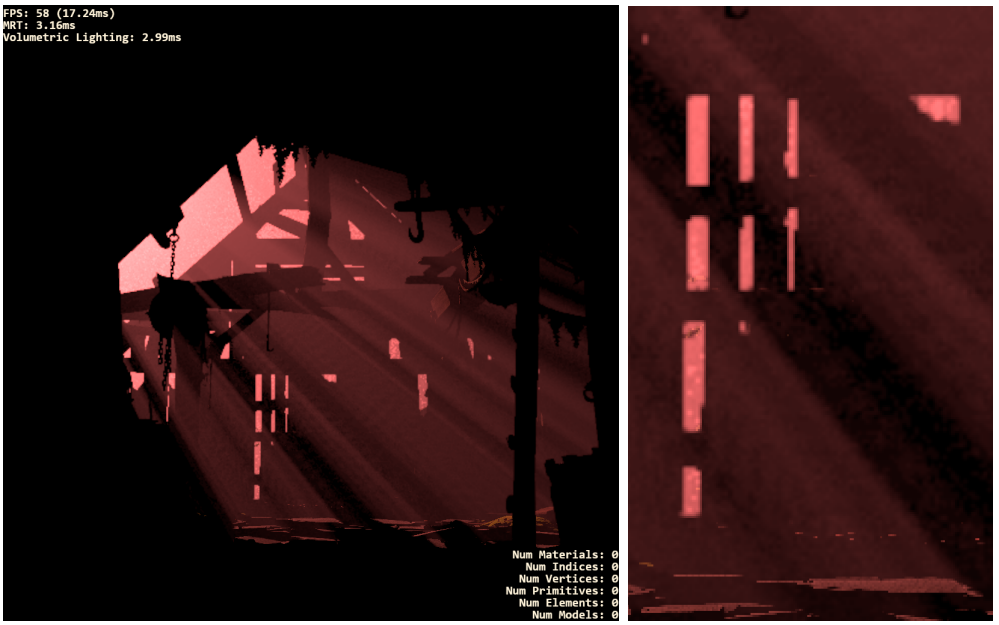

FIGURE 4.6: Sample jittering.

Figure 4.6: The noise generated from jittering manages to mask some of the imperfections caused by low amount of samples or reduction of the volumetric lighting buffer's resolution.

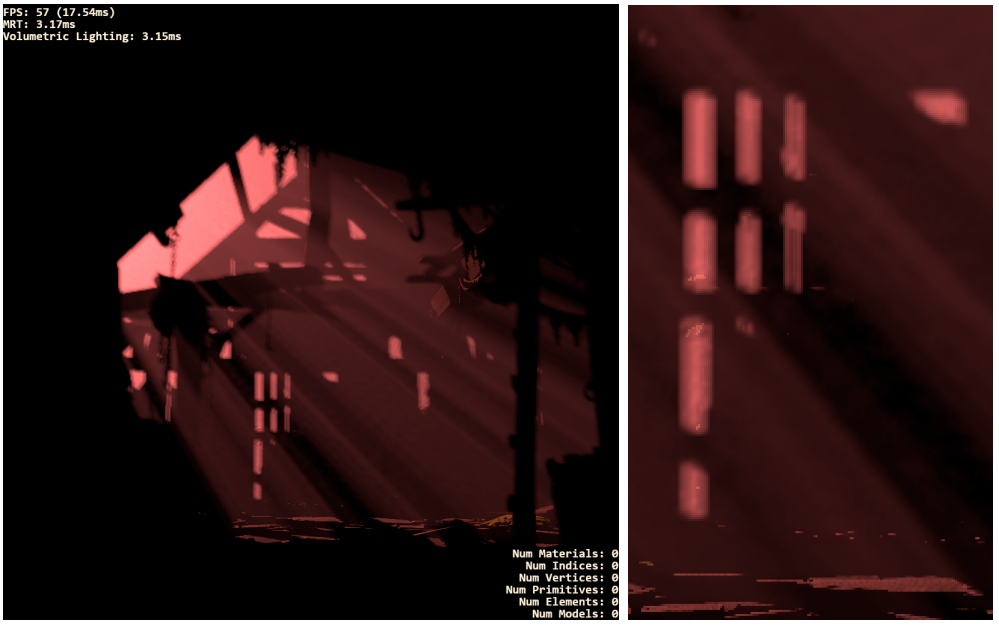

FIGURE 4.7: Gaussian blur.

Figure 4.7: In its turn Gaussian blur smoothens the noise created by the jittering of samples, improving the overall quality of the effect.

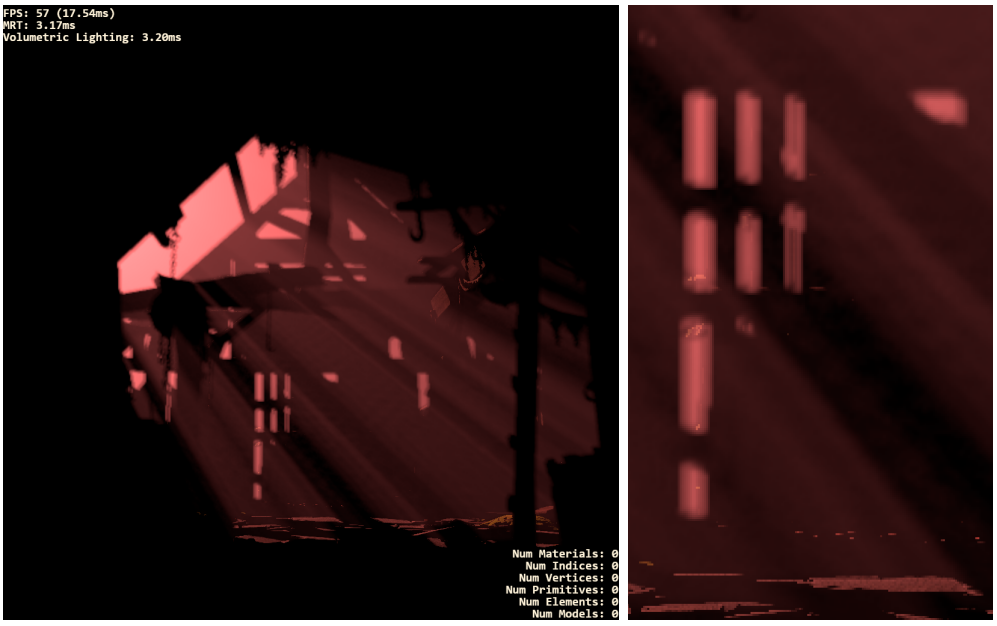

FIGURE 4.8: Sampling ray clipping.

Figure 4.8: Clipping the sampling ray is extremely beneficial to this implementation. However it is not obvious in this example (see Figure 4.12).

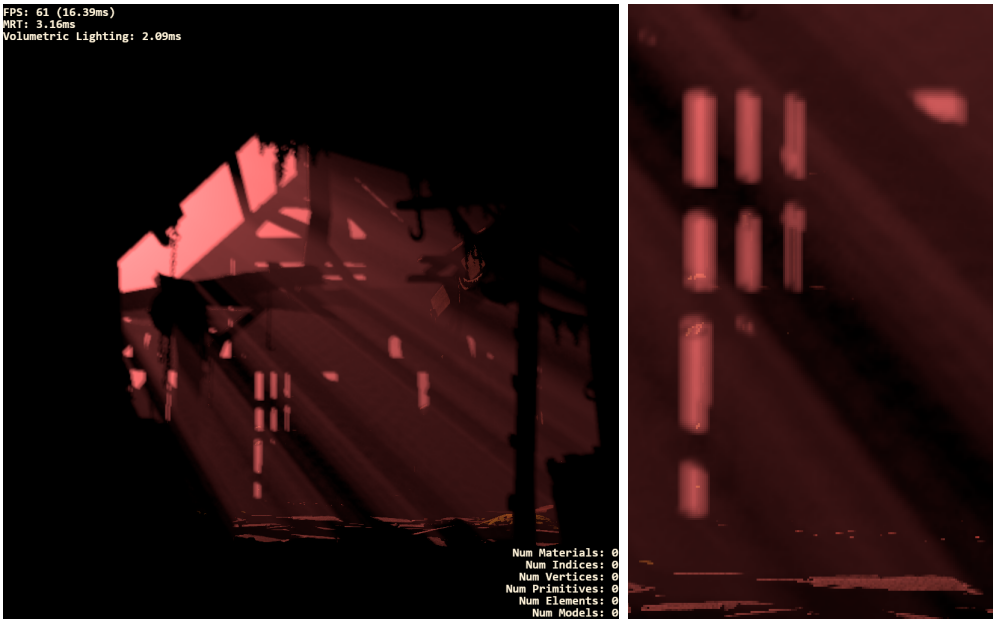

FIGURE 4.9: Dark sample skipping.

Figure 4.9: This example's light volumes are dispersed. Therefore, the benefits of skipping a dark sample's calculations are apparent.

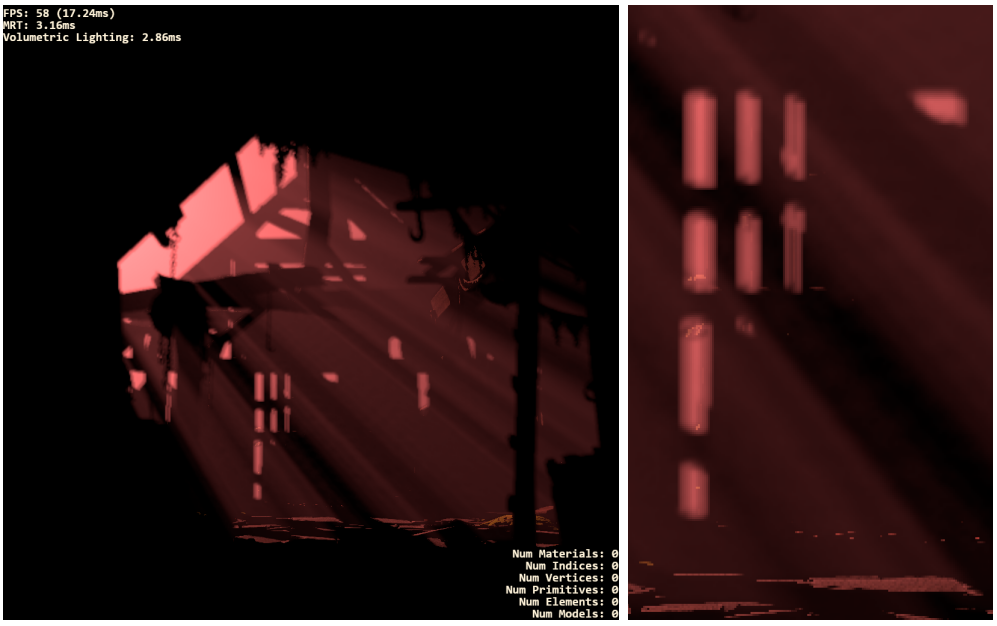

FIGURE 4.10: Shadow blur with percentage closer filtering.

Figure 4.10: Finally, we notice the positive influence of percentage closer filtering as it anti-aliases the light volumes.

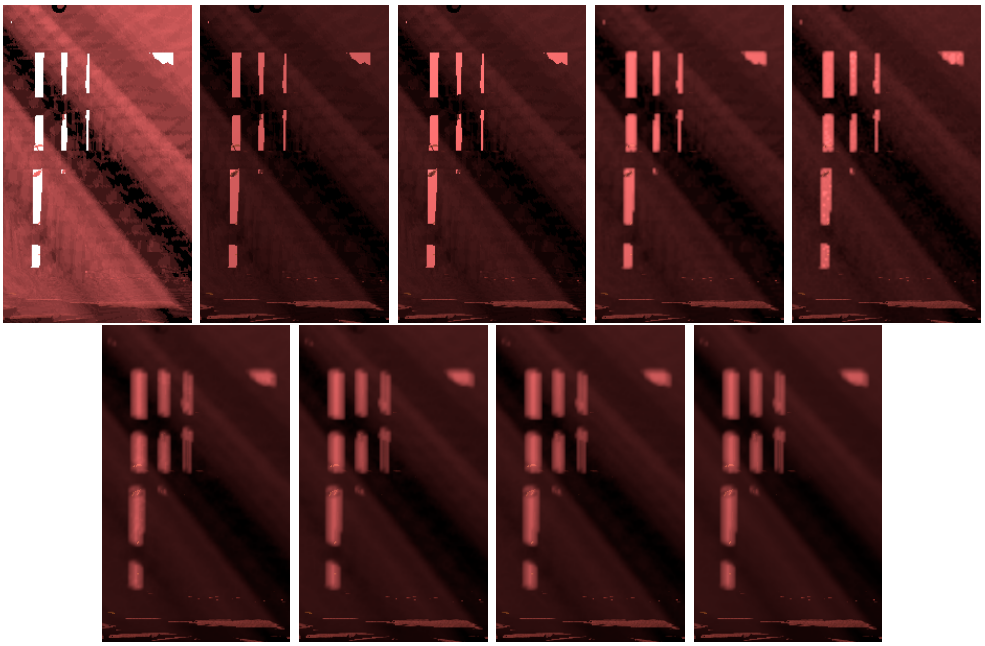

FIGURE 4.11: All stages (excluding animation) displayed sequentially. From top left to bottom right.

In Figure 4.11, the stages, expect phase animation, are displayed sequentially to show the performance and quality changes (near spectator).

FIGURE 4.12: Left side: Clipped. Right side: Not clipped.

Figure 4.12 proves that clipping the sampling ray is the best optimization for this effect. In this example, with only 50 samples (3.12 ms), we can achieve results that required 500 samples (13.7 ms) prior to clipping.

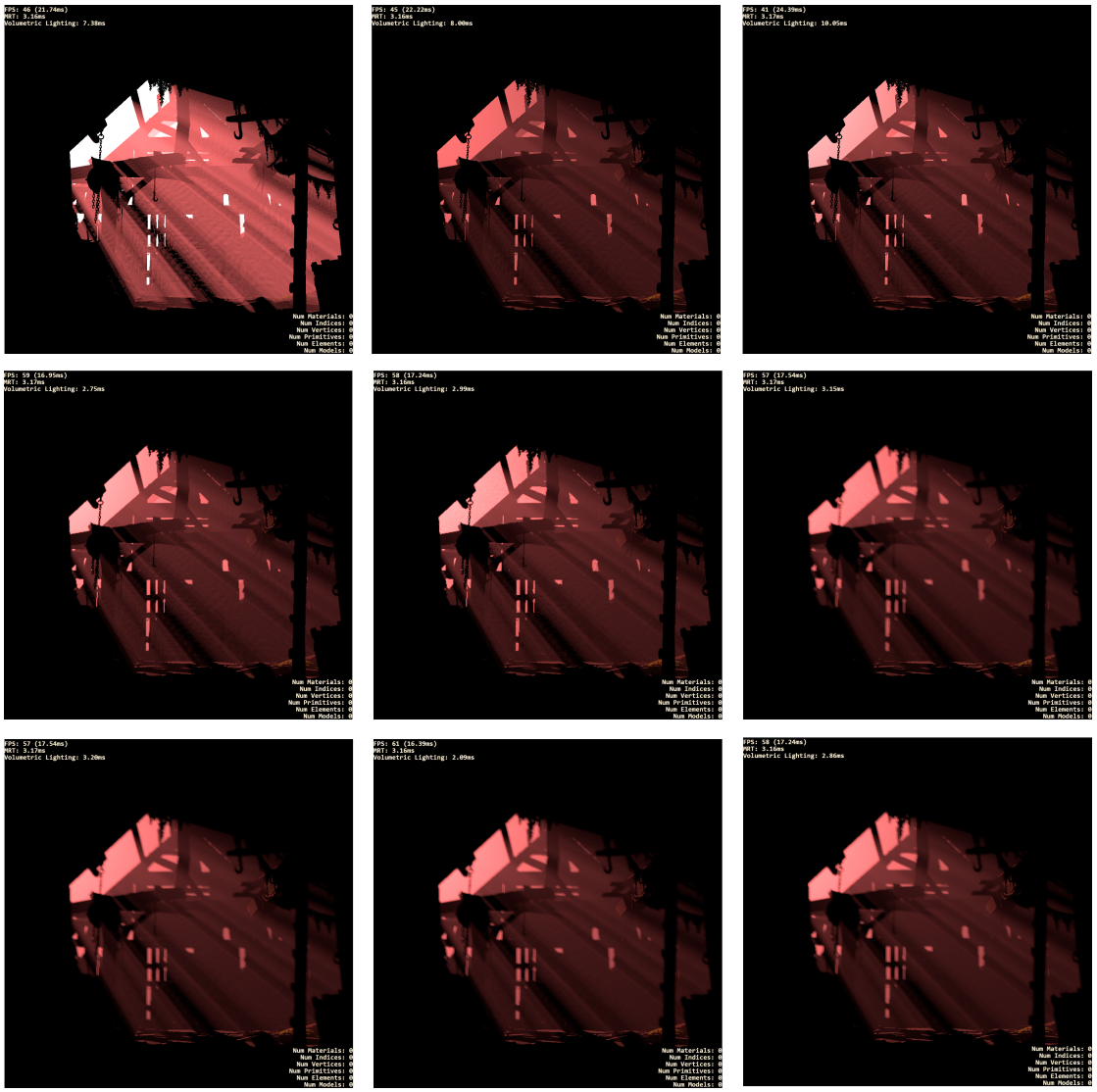

FIGURE 4.13: All stages (excluding animation) displayed sequentially. From top left to bottom right.

In Figure 4.13, the stages, expect phase animation, are displayed sequentially to show the performance and quality changes (distant spectator).

Volumetric Lighting Future Improvement

- More types of light sources:

  In XEngine, apart from spot lights, there are parallel and omni-directional types of lights. A future extension for this implementetion would be to include this effect to all of the above-mentioned light sources.

- Improve phase function animation quality or integrate a particle system:

  Phase function animation could be further improved and optimized. Alternatively, to reduce complexity of this issue, we could integrate a particle system in the engine to simulate the participating media inside the light volume. We could then combine both of those techniques to create realistic animations of the movement of such particles.

- Depth sensitive Gaussian blur:

  Gaussian blur in conjunction with buffer resolution reduction and jittering is probably one of the best optimization processes we employed in volumetric lighting. However, notice in Figure 4.14 that in depth changes, this technique causes extreme blurring. This originates from mixing non light volume information with the light shafts. To combat this issues, we could apply some sort of depth sensitive Gaussian blur to avoid blending data near those areas.

FIGURE 4.14: Extreme Blurring in depth changes.

Environment Mapping Evaluation

Environment mapping irradiance map pre-filtering evaluation:

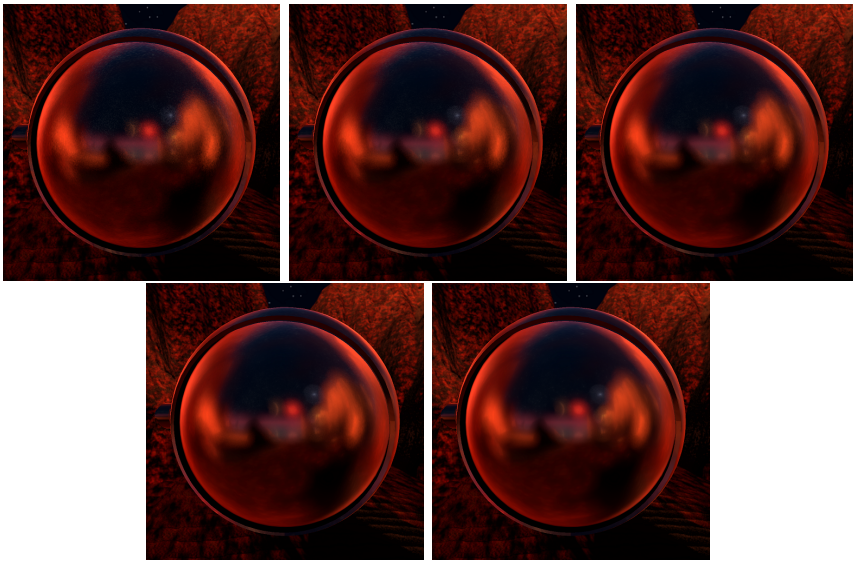

FIGURE 4.15: Irradiance map results for different samples and generation time. Top-Left: [20 samples, 3 seconds]. Top-Middle: [50 samples, 7 seconds]. Top-Right: [100 samples, 14 seconds]. Bottom-Left: [200 samples, 28 seconds]. Bottom-Right: [500 samples, 80 seconds].

Figure 4.15: After a certain amount of samples the result is virtually the same. Therefore, we should not use more than 200 samples for the pre-filtering process.

Environment mapping BRDF integration:

FIGURE 4.16: Left side: Irradiance maps. Middle-Left side: Lambert diffuse BRDF. Middle-Right side: Microfacet Beckmann specular BRDF. Right side: Irradiance map · (Lambert diffuse BRDF + Microfacet Beckmann specular BRDF)

Figure 4.15: When applied, the specular and diffuse BRDFs in conjunction with the appropriate irradiance map manage to reproduce the greatest part of reflections based on the surfaces' material properties.

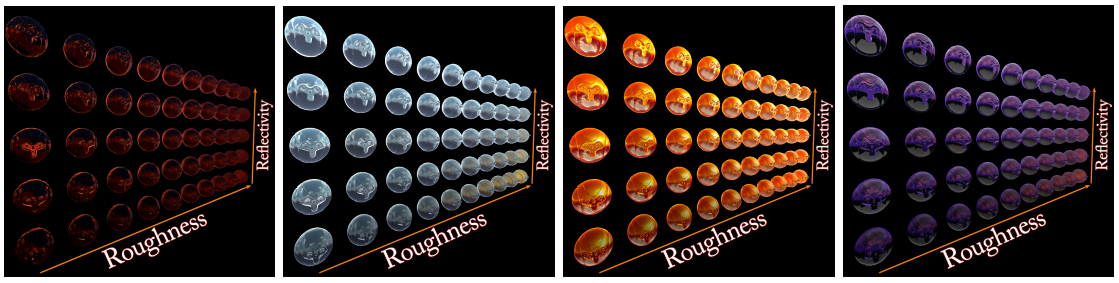

FIGURE 4.17: Roughness and reflectivity influence on the environment reflections.

Figure 4.17: After the integration of the Micro-facet BRDF there are many different material properties that now affect the reflections. Specular reflectivity is one of them and this example shows how it influences the result at every roughness level

Environment Mapping Future Improvement

- Improvements on irradiance map pre-filtering:

  Instead of uniform cone sampling, we could use the BRDF micro-facet distribution function to sample the main environment map for more accurate results. Also we could utilize the GPU for faster texture LOD generation (instead of the CPU). Finally, we could also apply some post process effects on the generated textures to enhance their quality.

- Reflection Occlusion:

  In this implementetion, we have not accounted for the fact that some of the reflections may be impossible as they can be obstructed by surrounding geometry. Therefore, to improve the effect's level of realism, we could apply some type of reflection occlusion using the shadow maps from light sources on the scene.

Bibliography

[Aby14] Pearl Abyss. Black Desert Online. *https://www.blackdesertonline.com/*, 2014. [Online; accessed September 19, 2017]. (cit. on 1.4)

[BSS93] Philippe Blasi, Bertrand Saec, and Christophe Schlick. A Rendering Algorithm for Discrete Volume Density Objects. *Computer Graphics Forum*, 12(3):201–210, aug 1993. (cit. on 2.1.2)

[Cor08] Nvidia Corporation. Volume Light. 2008. *http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/VolumeLight/doc/VolumeLight.pdf*. (cit. on 3.2.3)

[CT82] R. L. Cook and K. E. Torrance. A Reflectance Model for Computer Graphics. *ACM Transactions on Graphics*, 1(1):7–24, jan 1982. (cit. on 2.2.2)

[DS13] Walt Disney and Animation Studios. BRDF Explorer - Development and analysis of bidirectional reflectance distribution functions (BRDFs). *https://www.disneyanimation.com/technology/brdf.html*, 2013. [Online; accessed September 19, 2017]. (cit. on 2.12, 2.13, 3.24)

[E.m07] E.mil.mil. Dust dancing in the sunlight. *https://commons.wikimedia.org/wiki/File:Dust_dancing_in_the_sunlight_Me_074a.jpg*, 2007. [Online; accessed September 19, 2017]. (cit. on 3.11)

[Fel15] Markus Felix. Eindhoven Epic Metal Fest Moonspell. *https://commons.wikimedia.org/wiki/File:20151122_Eindhoven_Epic_Metal_Fest_Moonspell_0024.jpg*, 2015. [Online; accessed September 19, 2017]. (cit. on 1.1)

[Fou08] Blender Foundation. Big Buck Bunny. *https://commons.wikimedia.org/wiki/File:Big_Buck_Bunny_-_forest.jpg*, 2008. [Online; accessed September 19, 2017]. (cit. on 1.1)

[Gar01] Ian Garbett. Light attenuation and exponential laws, 2001. *https://plus.maths.org/content/light-attenuation-and-exponential-laws*. (cit. on)

[Geo13] George7378. Environment mapping. *https://commons.wikimedia.org/wiki/File:Environment_mapping.png*, 2013. [Online; accessed September 19, 2017]. (cit. on 1.7)

[HG41] Louis George Henyey and Jesse Leonard Greenstein. *Diffuse radiation in the galaxy. Astrophysics.* 1941. (cit. on 2.1.2)

[Ima03] Free Image. FreeImage The productivity booster. *http://freeimage.sourceforge.net/*, 2003. (cit. on 3.3.1)

[Ina08] Brocken Inaglory. Crepuscular rays in Golden Gate Park, San Francisco. *https://en.wikipedia.org/wiki/Crepuscular_rays#/media/File:Crepuscular_rays_in_ggp_2.jpg*, 2008. [Online; accessed September 19, 2017]. (cit. on 1.1)

[Jar08] Wojciech Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media.* PhD thesis, UC San Diego, September 2008. *https://cs.dartmouth.edu/~wjarosz/publications/dissertation/dissertation.pdf*. (cit. on 2.1.2)

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH 86*. ACM Press, 1986. (cit. on 2.1.1)

[Kat64] M. Katzin. The scattering of electromagnetic waves from rough surfaces. *Proceedings of the IEEE*, 52(11):1389–1390, 1964. (cit. on 2.2.2, 2.2.2)

[Lam60] J. H. Lambert. *Photometria, sive, De mensura et gradibus luminis, colorum et umbrae.* Augsburg, Germany: V. E. Klett, 1760. (cit. on 2.2.2)

[Liv13] Joshua Livingston. Crysis 3. *https://www.flickr.com/photos/xavierwilkinson/8488594019/in/photostream/*, 2013. [Online; accessed September 19, 2017]. (cit. on 1.1)

[Lor90] Ludvig Lorenz. *Lysbevægelser i og uden for en af plane lysbølger belyst kugle*. Videnskabernes Selskabs Skrifter, 1890. (cit. on 2.1.2)

[Mak11] Custom Map Makers. I map, therefore I am. *http://www.custommapmakers.org/skyboxes.php*, 2011. [Online; accessed September 19, 2017]. (cit. on 3.3.2)

[Mie08] Gustav Mie. Beiträge zur optik trüber medien, speziell kolloidaler metallösungen. *Annalen der Physik*, 330(3):377–445, 1908. (cit. on 2.1.2)

[Nam16] Bandai Namco. Dark Soul III. *https://www.darksouls3.com/us/*, 2016. [Online; accessed September 19, 2017]. (cit. on 1.4)

[NMN87] Tomoyuki Nishita, Yasuhiro Miyawaki, and Eihachiro Nakamae. A shading model for atmospheric scattering considering luminous intensity distribution of light sources. *ACM SIGGRAPH Computer Graphics*, 21(4):303–310, aug 1987. (cit. on 2.1.2)

[NW09] Greg Nichols and Chris Wyman. Multiresolution splatting for indirect illumination. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games - I3D 09*. ACM Press, 2009. (cit. on 2.2.2)

[PB09] Markus Persson and Jens Bergensten. Minecraft. *https://minecraft.net/en-us/*, 2009. [Online; accessed September 19, 2017]. (cit. on 1.5)

[PJH17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. Monte carlo integration. In *Physically Based Rendering*. Elsevier, 2017. (cit. on 2.2.2, 2.2.3)

[PV13] Georgios Papaioannou and Konstantinos Vardis. XEngine Documentation and API Reference Version 1.0, 2013. *http://graphics.cs.aueb.gr/graphics/downloads/xenginedoc/XEngine.html*. (cit. on 1.1, 3.1, 3.2, 3.4, 3.6, 3.7, 3.8, 3.9, 3.10, 3.11, 3.12, 3.13, 3.14, 3.15, 3.16, 3.17, 3.19, 3.20, 3.22, 3.26, 3.27, 3.28, 3.29)

[Rá10] Daniel Rákos. Efficient Gaussian blur with linear sampling. *http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/*, 2010. (cit. on 2.1.4)

[Sch94] Christophe Schlick. An Inexpensive BRDF Model for Physically-based Rendering. *Computer Graphics Forum*, 13(3):233–246, aug 1994. (cit. on 2.2.2)

[Sob94] Ilya M Sobol. *A Primer for the Monte Carlo Method*. CRC Press, 1994. (cit. on 2.1.1, 2.2.3)

[Str71] Hon. J.W. Strutt. On the scattering of light by small particles. *Philosophical Magazine*, 61:447–454, 1871. (cit. on 2.1.2)

[Stu06] Bethesda Game Studios. The Elder Scrolls IV: Oblivion. *https://elderscrolls.bethesda.net/en/oblivion*, 2006. [Online; accessed September 19, 2017]. (cit. on 1.4)

[Tec15] Unity Technologies. The Blacksmith Environments. *https://www.assetstore.unity3d.com/en/#!/content/39948*, 2015. (cit. on 3.6)

[TS67] K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces∗. *JournaloftheOpticalSocietyofAmerica*, 57(9) : 1105, *sep*1967. (*cit.on* 2.2.2)

[Var16] Konstantinos Vardis. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. PhD thesis, Athens University of Economics Business, December 2016. *http://kostasvardis.com/files/research/PhD_Thesis_Kostas_Vardis_book_online_lowres.pdf*. (cit. on 1.3)

[Wei09] Friedel Weinert. Compton experiment (or compton effect). In *Compendium of Quantum Physics*, pages 115–117. Springer Berlin Heidelberg, 2009. (cit. on 2.1.3)

[Wei17a] Eric W Weisstein. Disk Point Picking. *http://mathworld.wolfram.com/DiskPointPicking.html*, 2017. From MathWorld–A Wolfram Web Resource. (cit. on 3.3.3)

[Wei17b] Eric W Weisstein. Sphere Point Picking. *http://mathworld.wolfram.com/SpherePointPicking.html*, 2017. From MathWorld–A Wolfram Web Resource. (cit. on 3.3.3)

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques - SIGGRAPH 78*. ACM Press, 1978. (cit. on 2.1.5)

[WMLT07] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet Models for Refraction through Rough Surfaces. In Jan Kautz and Sumanta Pattanaik, editors, *Rendering Techniques*. The Eurographics Association, 2007. (cit. on 2.2.2)