Volumetric lighting implementations in games

Karol Osipovic
SAXION UNIVERSITY OF APPLIED SCIENCES
GAME ENGINEERING, SPECIALIZATION ONE
STUDENT 346457
Abstract

This paper researches different volumetric lighting implementations in the modern games. Physical theory behind the atmospheric scattering and five known algorithms of its implementation are introduced. Paper shows the development of a basic Raymarching algorithm and Bilateral Upsampling optimization using Unity Engine. Approach was documented by Vos (Vos, 2014) in a chapter about the volumetric fog in Killzone: Shadow Fall. Paper also includes experiments with the different scattering functions and performance comparisons between different algorithms in Unity. Research also gives hints of how the setup could be developed further with the known resources and algorithms.

*Keywords*: atmospheric-scattering, raymarching, HLSL, C#, Unity Engine, Bilateral Upsampling, Mie-scattering, Rayleigh-scattering, volumetric lighting, volumetric fog.
## Contents

1. Introduction .................................................................................................................. 3
2. Theory ........................................................................................................................... 4
   2.1. Atmospheric Scattering ......................................................................................... 4
   2.2. Volumetric Lighting Implementations ................................................................. 6
       2.2.1. Combination of sky colour scattering and linear fog ..................................... 6
       2.2.2. Post-Process Effects .................................................................................... 6
       2.2.3. Artists Assets .............................................................................................. 7
       2.2.4. Polygonal Volumetric Lighting ................................................................... 7
       2.2.5. Raymarching Solutions ............................................................................... 8
   2.3. Evaluation ................................................................................................................ 10
3. Research Approach and Questions ................................................................................ 10
4. Results ........................................................................................................................... 11
   4.1. Unity Setup ........................................................................................................... 11
       4.1.1. Performance and Results ............................................................................ 13
   4.2. Bilateral Upsampling ........................................................................................... 14
   4.3. Different scattering functions .............................................................................. 16
   4.4. Performance comparisons between different algorithms ..................................... 18
   4.5. Evaluation ............................................................................................................. 19
5. Conclusions .................................................................................................................... 19
6. Discussion ....................................................................................................................... 19
7. References ....................................................................................................................... 21
1. Introduction

We can observe volumetric lighting effects as a natural phenomenon (Figure 1.1(b)). It can be seen indoor as well as outdoor, by scattering of light in such environments as moisty or dusty rooms, or smoky plains. Because modern rendering is trying to achieve natural looking environments, we can add a lot to the atmosphere of a game by creating such effects (Figure 1.1(a)). These effects are needed to reproduce the reality correctly, but it also helps us to perceive the right distance and hide LOD (Level of detail, decreasing complexity of the model the farther it gets).

![Figure 1.1](a) Volumetric Light Effect in a game (b) Volumetric Light as a natural phenomenon

This leads us to our research goal: we will analyse different approaches for implementing volumetric lighting in games. After that we will select one method and implement it into Unity Engine using the HLSL shading language and the C# programming language.

Many developers were introducing different methods within the last decades, constantly evolving and using the capabilities of modern graphical processors. Research and solutions came from individual developers as well as big companies which were trying to produce the optimal solution between realistic look and minimal performance costs.

In the paper below we describe five different methods of implementing volumetric lighting into games. Section 2 will contain the physical theory behind natural volumetric lighting (atmospheric scattering) and a short description with evaluation for each implementation method. Section 3 will show which questions were tried to be answered in the research process. The actual method implementation, optimization and experimentation is shown in Section 4. Respectively Section 5 and 6 will include conclusions and discussion.
2. Theory

Volumetric light effects are basically example of the effects one can get by studying the atmospheric scattering. Atmospheric scattering is a natural phenomenon, it analyses how much light particles are scattered in the atmosphere. By simulating realistic visual results of the sky color, fog, clouds, “god rays”, light shafts or volumetric shadows are got.

Paper will study and try to achieve only the dynamic light shafts with shadows. To achieve understanding of the subject theory behind physical atmospheric scattering will be studied. Furthermore, this section will analyse the algorithms which are used to simulate discussed effects and how modern games dealt with this question.

2.1. Atmospheric Scattering

In simple light rendering scenarios it is assumed that light moves in a vacuum: direct lighting, where no radiance is lost or gain on the light path. But if the light path from a physical side is analysed it is plain that there are a few more factors which have a direct influence on the incoming light result: In-Scattering, Out-Scattering and Absorption (Naty Hoffman, 2012).

Absorption is the easiest factor to specify, it tells how much light is absorbed by the particles in the atmosphere. Absorbed light particles are transformed into thermal energy and are not visible to the eye (Figure 2.1(a)).

Out-Scattering is the factor which indicates how much light is bounced-off the particles into different directions than your eye path (Direction from your eye to something you look at). Figure 2.1(b) illustrates the process.

In-Scattering is the factor which specifies how much light is scattered into your eye path from different particles in the atmosphere. Basically it tells how much light is out-scattered from other particles into your eye path (Figure 2.1(c)).

![Arrows indicate the light path (a) Absorption of light particles (b) Out-Scattering of light particles (c) In-Scattering of light particles (Naty Hoffman, 2012)](image)

Depending on the participating media (the type of particles which light is scattered through) the scattering has different physical models which describe the process. Two main scattering models are Rayleigh scattering and Mie scattering.

The Rayleigh model is mostly used to measure the scattering through very small particles like the atmosphere air. It is very isotropic (same value when measured in different directions) (Figure 2.2) and has almost no absorption capability. Thanks to Rayleigh scattering we are able to see the blue sky color.
The Mie model on the other hand is used to measure scattering of bigger participating media like dust or fog for example. Comparing to the Rayleigh model it has much higher forward scattering factor (Figure 2.2) as well as higher absorption factor.

![Rayleigh Scattering and Mie Scattering](image)

**Figure 2.2** Scattering types shown in the pictures above, where arrows visualize scattering direction after particle hit.

Physically light particles can out-scatter and in-scatter multiple times until they enter your eye vision path, but it is a very complicated process to compute in real time. So research will use single-scattering model in its calculations. Common way to approximate the light in-scattering model is to use a phase function.

The phase function which will be used to approximate Mie scattering is the Henyey-Greenstein phase function (Greenstein, 1941). This function is pretty simple and does not put a lot of difficulty on the graphical computation. It is described using following formula:

\[
p(\theta) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g \cos \theta)^{3/2}}
\]

G – the value which describes how much light is scattered forward (ranges from -0.99 to 0.99), where -0.99 gives reverted lighting effect, 0 gives isotropic results (Rayleigh scattering) and 0.99 makes light stay in one spherical place (no scattering forward).

P – returned light per sample (ranges between 0 and 1).

\(\Theta\) – the angle between light direction and view direction.

Furthermore, to calculate the out-scattering we will use the Beer-Lambert law. It is an exponential function of travelled distance by light, in a predetermined participating media. This function uses the extinction coefficient which is in other words optical thickness (how thick participating media is). In Figure 2.3 the transmittance with distance using different extinction coefficients is visualised.

![Transmittance Graph](image)

**Figure 2.3** Graph of transmittance with distance using Beer-Lambert law and different optical thicknesses (0.5, 1.0, and 2.0) of the property which light is travelling through. (Vertical axe – transmittance, horizontal axe – distance)
2.2. Volumetric Lighting Implementations

In this section different algorithms and theory behind volumetric lighting implementations in games will be described. Different research papers and discussions will be summarized. Because limited time is available for implementation, a good understanding of each technique is needed.

2.2.1. Combination of sky colour scattering and linear fog. Method implemented in CryEngine2 by Wenzel (Wenzel, 2006) and described in his paper about real-time atmospheric effects in games. This approach is simplified a lot compared to the methods used ten years later because of lacking compute power at that time. The method is a combination of real time sky light and global volumetric fog. For sky light a mixed CPU/GPU approach is chosen, where a loop executes for each sampled point on sky hemisphere to calculate Mie and Rayleigh scattering. Calculations for the 128x64 sample points are done on the CPU and passed to the GPU as a floating point texture which colours the sky. Full update on that texture takes 15-20 seconds and is distributed over several frames. To calculate the height/distance based fog exponential function similar to Beer-Lambert law is used, which results in a linear fog with an exponential falloff. Finally the fog colour is matched to the colour of the sky and that results in appropriate sun halo on the horizon. Figure 2.4 illustrates sunset shots with different settings for Mie/Rayleigh/global density. This approach unfortunately does not provide shadowing and varying medium density. In addition it has problems with transparent objects.

![Figure 2.4 Different sun-set shots using simple scattering method described by Wenzel](image1)

2.2.2. Post-Process Effects. Next method is simpler. By using the post-process effects light rays (sun shafts) and simulation of volumetric lighting can be achieved with a bit of cheating. Radial blur and bloom effects with some additional calculation results in a “god ray” effect which arises when a very bright light source is partly obscured. Unity has such effect in the default package Effects under name “SunShafts” (Figure 2.5(b)). Method was also used in UE3 and Cryengine. This approach

![Figure 2.5 (a) Example of “Sun shafts” provided in Unity docs, made with post process effects. (b) Volumetric Light Scattering using Post-Process from GPU Gems 3](image2)
Volumetric Lighting Implementations in Games | Karol Osipovic

however has a lot of disadvantages, it is not physically based, it does not produce shadows and it disappears if light source is not visible on the screen. Nevertheless, in the “GPU Gems 3” (Mitchell, 2007) a different post-process approach is shown. This method applies physical scattering and shadowing. By using 2D projections on the screen space method gathers light samples across the projections with applied physical scattering. Projections start on light centre position and go through all screen pixels. Furthermore, method checks if a 2D ray has occluded geometry and as a result produces shadow effects (Figure 2.4(b)).

2.2.3. Artists Assets. Another simple way to add volumetric light effects to your game are assets created by artists. A simple camera facing billboard or particles with a fadeout on the camera intersection. Obvious disadvantages of such approach is dependability on artists, not being dynamic enough (no change with other lighting or shadows, not following the positioning of the sun). Figure 2.6 (a) shows an example of such effect.

![Billboard method implemented in a static environment of a game](image1)

![Visualization of the geometry mesh for the volumetric light in the polygonal volumetric lighting](image2)

**Figure 2.6** (a) Billboard method implemented in a static environment of a game (b) Visualization of the geometry mesh for the volumetric light in the polygonal volumetric lighting

2.2.4. Polygonal Volumetric Lighting. This method was introduced by Hoobler in 2016 GDC presentation (Hoobler, 2016) and was developed to integrate into Fallout 4, but the libraries and source code is published on GitHub. What the method offers is a fast, flexible and physically based integration of volumetric lighting. It can be easily implemented into existing engines and has low compute costs. Its implementation is easily understood if you would imagine a geometric volume which fills the lighted volume of the scene as a mesh (Figure 2.6(b) visualizes the geometric volume with one light source and eye position).

To calculate the volume in real-time rendering this approach uses a depth map information encoded in the shadow map of the light combined with tessellation (Vertex processing where patches of vertex data are subdivided into smaller primitives). The calculation uses sums and differences of intervals taken from lighted and shadowed parts of the scene. The other way they describe this: method subtracts light on the outside faces of the volumetric geometry and adds light on the inside faces of it. This algorithm provides solutions for directional/omnidirectional and spotlights. The discussed issues of such technique are: getting intense effects without washing out the scene, shadow map inconsistencies become much more noticeable, flickering caused by anti-aliasing and poor performance in specific views (ex. sunset through woods – complex volumetric geometry). Figure 2.7 visualizes the effects of volumetric lighting in Fallout 4.
2.2.5. Raymarching Solutions. Last but not least the most popular approach is based on raymarching. There are a lot of different algorithms which use a raymarching approach, but this section will describe two advanced algorithms which are introduced in “GPU Pro” book series. First one was used in Killzone: Shadow Fall and was described by Vos (Vos, 2014). Second approach was developed by Wronski and implemented in Assassin’s Creed 4: Black Flag (Wronski, 2016).

Starting with the simpler method: how Killzone: Shadow Fall tackled the problem. To explain the core of the algorithm raymarching and its implementation should be introduced. The discussed method implements the algorithm using deferred rendering. To render the volumetric light effect, a shape representing the light (sphere for point, screen quad for directional) should be rendered. Each shape is rendered using an additional Shader which does the calculations. Volumetric lighting calculations are done separately from the scene rendering, using only shadow maps and depth textures. To calculate the scattering for each pixel of the rendered volumetric shape we project a ray from the camera position to the world position. After that we enter ray marching loop, where we collect samples along the ray by a predefined number of steps (Figure 2.8(a)). After adding all samples the final colour for the volumetric lighting is delivered. That colour is later added to the normally rendered scene color. Of course rendering at full resolution is very heavy on the performance, so the method uses lower resolution rendering (half of the actual). Performance is increased a lot from using it. Bilateral upsampling (See Section 4.2) is used to scale the results to full resolution when combining it with the scene. Furthermore, method uses interleaved sampling (See Section 6) to reduce number of required ray steps. Combined with a custom blur effect the
difference from the starting full screen projection is minimal, but the resolution of rendering is reduced by half, iterations of samples reduced from 256 to 32.

To control when, where and how much light should be scattered *Killzone* uses a 3D texture which holds the scattering amount over depth for each pixel on the screen. The 3D texture is eight times smaller than the original render resolution and has 16 depth slices. Distance between slices is smaller near camera and grows bigger further it gets from the camera (up to 128 meters). Raymarching algorithms have an issue with rendering transparent objects because depth map does not represent them correctly. To solve the issue this algorithm uses an additional 3D texture to calculate the light intensity. Figures 2.8 b and c show the final results from this method of volumetric lighting.

Next method explained by Wronski and implemented in *Assassin’s Creed 4: Black Flag* tries to solve the problems of precision and performance in standard raymarching solutions. The core ideas behind it are usage of compute Shaders and 3D textures to store the volumetric data of scattering. Wronski explains that standard solutions operate in loops, which is counter-productive on modern GPU’s which can launch thousands extra thread waves if operated manually. By using compute Shaders and UAV (Unordered Access Views) which provide ability to arbitrarily (based on random choice than system) read/write memory at any point in rendering pipeline instead of being forced to follow strict order of it (imagine that pixel Shader can only write to one location per render target – its own pixel, with UAV pixel Shader can write to random locations in whatever way UAV has bound it), this method is able to run different raymarching steps parallel to each other. To store the volumetric data method uses earlier mentioned 3D textures sized 190x90x128 or 190x90x64, which are aligned to camera frustum.

![Figure 2.10](a) Screenshot of the volumetric fog from the Assassin’s Creed 4: Black Flag. (b) Volumetric fog of Unreal Engine 4

The algorithm is done in 4 steps: estimating density of the participating media, calculating in-scattering, ray marching and applying the effect to shaded objects.

*First*, in the estimation of density method uses exponential height fog distribution (assuming that fog is thicker near ground level) and purely for art direction it adds 3D Perlin noise to the calculations, resulting in 3D texture.

*Second*, In-Scattering is calculated by launching a thread per Texel in the 3D texture, results are stored in a second 3D texture combined with regular ambient lighting.
Third, having results in 3D textures, 2D raymarching is started to combine the results and add some final calculations per Texel. Contrary to second step, slices are calculated serially, launching group of threads per Texel of the slice size.

Fourth, the results are applied to the pixel color. After that additionally some Under-Sampling and Aliasing problems are solved. Figure 2.10 (a) illustrates a screen of volumetric fog from Assassin’s Creed 4: Black Flag.

Similar approach can be found in the Unreal Engine 4, where you have volumetric fog which you can add from default assets of the engine (Figure 2.10 (b)).

2.3. Evaluation

In this section the physical theory behind atmospheric scattering was discussed and most used formulas were introduced. Following it, five known methods of volumetric lighting in modern games were described. Based on performance and visual looks Raymarching and Polygonal solutions were far better algorithms compared to other methods. Research showed that Wronski Raymarching and Hoobler polygonal solutions were too complicated to implement in given time frame of a few weeks. That left the Vos Raymarching solution which when separated into smaller steps can be implemented easily and extended by adding optimizations and visual extensions to the basic algorithm.

3. Research Approach and Questions

The problem statement discussed in the paper has a broad spectrum. To gain an understanding and ability to practically implement the topic we decided to keep it simple. Main research question which was asked is: How to implement volumetric lighting into the Unity Engine? This question has clear variables which need to be tackled. Unity Engine is one of those. From here comes the first sub question: How to setup Unity for the discussed algorithm implementation? After this research basic results should be in a working setup. Because research operates in rendering pipeline scope it will need a good optimization. Second sub question is exactly about that: How to optimize implementation of a basic Raymarching algorithm? To be able to freely operate with setup after that, our research needs a deeper understanding of what can be done with the algorithm. Third question which helps to answer the main one is: What possibilities different scattering functions provide? Finally to get the scope and compute difficulties of different volumetric lighting algorithms, paper will compare performances of those implemented into the Unity project. Following that, the final sub question is: How different volumetric lighting algorithms perform in Unity Engine?
4. Results

As a result of the research method described by Vos in his article about volumetric lighting in *Killzone: Shadow Fall* (Vos, 2014) was chosen to be implemented. A big help of structure and some pieces of code was provided by Skalsky in his project of the Volumetric Lighting for the Unity 5 (Skalsky, 2015). To fit into the time frame implementation is provided only for point lights and it uses deferred rendering. This section will start with explanation of how to setup Unity for the chosen approach. After that, paper will show implemented optimization strategies using lower resolution rendering and a section with different implementations of atmospheric scattering formulas.

4.1. Unity Setup

To implement the researched approach into a Unity Engine project we need to imagine how our basic algorithm works and what we need to make it work. Figure 3.1 illustrates the basic steps which are taken to setup the discussed algorithm. The setup collects data per volumetric light and adds it to a texture containing information about all such lights. For the final result algorithm sums up volumetric lights texture with the scene texture which deferred camera provides. If Figure 3.1 steps are considered separately, we can specify what we need from Unity to make the setup work.

![Figure 3.1](image.png)

**Figure 3.1 Image above illustrates basic steps in our algorithm for volumetric lighting**

First of all, setup will use the *RenderTexture* of Unity Engine as a base for the volumetric lights texture. *RenderTexture* is a texture type in Unity to which we can render to. One usage of such texture can be setting it as target for camera rendering, this will make a camera render into a texture instead of the screen. Considering setup needs to render lights into texture at precise time one more Unity feature needs to be introduced: *CommandBuffer*. This feature extends Unity rendering pipeline and can be used to execute actions at specific time. *Command buffers* hold list of rendering commands and can be executed at various points during camera rendering. To get results per light, algorithm needs to do mathematical calculations for raymarching by casting ray from camera position and calculate attenuation by using light shadow map. To use shadow map calculated by Unity algorithm adds command buffer after a shadow map is calculated and uses it as a global texture (Figure 3.2 shows basic implementation of discussed process). Calculations per light are done in a custom vertex/fragment Shader. Finally, to add volumetric lights texture to scene texture setup uses Unity *ImageEffect* function for post process effects.
To implement this setup two classes (C#) and two Shaders (HLSL) were written:

**VolumetricRenderer** a global script for the camera responsible for storing a volumetric lights texture, executing events in light objects which do all calculations and combining final textures. It has two updating functions. One is a default Unity function `OnPreRender` which is executed before a camera renders the scene. Here a low quality view/projection matrix is generated for the light Shader (it is sent as a parameter in the event), command buffer is executed before lighting calculations and events for volumetric lights are triggered. Second function is a Unity function for image effects `OnRenderImage`. Here after all calculations the scene texture and volumetric lighting texture are combined. A simple `Blit` Shader does the adding of the textures.

**VolumetricLight** a script which is attached to each volumetric light object. It is responsible for calculating volumetric lighting per light and filling the global texture with the results. Each light object has a material (with `shVolumetricLight` Shader) which is doing the actual scattering calculations. Here parameters of the scattering, extinction, number of raymarch samples and Mie G are adjusted. It has its own `CommandBuffer` which is executed after light shadow map is calculated by Unity. It is also responsible for sending all parameters into Shader (data about light/ camera/ scattering/ MVP matrix etc.).

**ShVolumetricLight** a Shader which does calculations for each light to calculate the scattering using raymarch approach. Algorithm projects a ray from the camera position to the world position of a current fragment and executes `RayMarch` function using its start, length and direction. It uses Unity default camera depth texture to determine the end of ray and if it hits something.

Ray marching function is pretty simple and reflects the physical theory we discussed in the Theory Framework chapter. It determines step size from the sample count and ray parameters after that function executes a loop to calculate light contributions within steps (Figure 3.3). Algorithm
calculates light attenuation using a shadow map provided by Unity (code part from Skalsky), which provides shadows for volumetric lightning because it multiplies attenuation with light contributions. Exponential function is used for extinction and calculation of a phase function with Mie scattering follows. Lastly, the final light is updated and algorithm moves a step further.

**ShBlit** is the Shader which is responsible for summing the two textures. Simple code which multiplies main and source textures in the fragment Shader.

### 4.1.1. Performance and Results

This basic algorithm gave good looking results (Figure 3.3), unfortunately at very high costs. With 256 samples the frame rate dropped to 14 FPS (Figure 3.3(b, c)), it took the Camera.Render thread 64 Ms to calculate a frame. If number of samples is reduced to 60 scene runs on 30 FPS (Figure 3.3(a)), where Camera.Render thread uses 28 Ms for its calculations per frame, but results contain visible artifacts.

These results show that setup needs optimization and reduced performance costs. Setup used a low quality graphic card, nevertheless results are not optimal.

**Figure 3.4**
(a) Setup uses 60 samples (30 FPS) (b) Setup uses 256 samples (14 FPS), higher Mie g parameter (~0.65) (c) Setup uses 256 samples (14 FPS), lower Mie g parameter (~0.4)
4.2. Bilateral Upsampling

To improve performance, as most approaches suggest lower resolution rendering was used for current setup. To put it in a simple way, the resolution of global RenderTexture to which volumetric lights are rendered to was reduced. In addition to that a camera depth texture was down sampled as well. Experiments with half / quarter and eighth of actual resolution were done. After volumetric texture is filled with lights, Bilateral Upsampling is used to upsample volumetric texture to the full resolution.

Bilateral Upsampling is type of a filter to upscale images to full resolution. It uses the four closest low resolution texels in a 2x2 bilinear footprint (Figure 3.5) to determine the color of the high resolution texel. It adds additional weights for depth differences of the low and high resolution depth textures.

![Figure 3.5](image)

Figure 3.5 (Left) Low resolution 2x2 texels (Right) High resolution 4x4 texels, where shaded texel illustrates how we get the higher resolution from the lower (Sampling 4 nearest texels).

To implement this approach additional BilateralUpsampling Shader with 2 passes was developed, one for down sampling of depth texture and second for bilateral upsampling (the function for upsampling is from Skalsky’s project (Skalsky, 2015)). Additional calculations are done in VolumetricRenderer class, where in OnPreRender function the depth texture is down sampled and in OnRenderImage function volumetric texture is up-scaled to full resolution before applying it to the scene.

Figure 3.6 illustrates the results and performance comparisons between different resolutions. We can see that rendering at half of the original resolution, volumetric lightning effect is similar to original resolution. On the other hand, quarter resolution leaves us with slightly worse looking results with some pixelish artifacts. Lastly eighth of the actual resolution is definitely unacceptable. If we speak about the optimal performance and visual results, we would choose between half and quarter of the original resolution, depending on the specific scene and sample count we get pretty decent performance results already.

Calculations for the quarter and eighth resolution were done using the same code but providing different sizes of the downscaled textures. Multiple passes for up-sampling were not needed because those produced the same result as one pass with the few times down-sampled texture.
Figure 3.6 Up we have 3 rows of images illustrating results with different resolutions of volumetric lighting textures. First row, half resolution, 256 samples, 28 FPS. Second row, quarter resolution, 256 samples, 42 FPS. Third row, eighth of original resolution, 256 samples, 62 FPS.
4.3. Different scattering functions

To understand what type of light scattering can be simulated with current setup experiments with different physical scattering functions were done. As it was discussed in Section 2.1 setup uses a Henyey-Greenstein phase function in the current version. This function has a $g$ parameter which gives opportunity to simulate different Mie scattering patterns (dense fog, light fog or even bigger particles), but it is approximate. Figure 3.7 visualises scene with the different $g$ parameter values. As it can be seen dense fog or no fog are the results which can be produced. It is also interesting to know that negative $g$ value makes light particles to scatter backwards as Figure 3.7 (c) illustrates.

![Figure 3.7 Images above illustrate the scene rendered with different Mie $g$ parameter. (a) $g = 0.65$ (b) $g = 0$ (c) $g = -0.8$](image)

Mie scattering is a solution which Mie developed in XIX century. We have tried to implement two equations (Mie-Lorenz solutions) which represent his light/dense fog (Figure 3.9 (a, b)) and compare the results with simulation of our current phase function (Figure 3.9(d)). Furthermore, the Rayleigh scattering equation was implemented (Figure 3.9(c)) and experimented with as well. Overall almost identical results (Figure 3.8(a, b and c)) could be simulated. Nevertheless, the light fog looked slightly more natural with the Mie-Lorenz equation. It is also visible that the Rayleigh equation produces slightly nicer results.

Overall, it is questionable if equations are better than the phase function, almost identical results could be simulated, but if we consider direct lighting which covers bigger fields, those small differences can be big enough to consider which approach is better. We should also think how expensive is power of 32 in the dense fog equation comparing to the phase function calculations.

Lastly, Rayleigh scattering was combined with the light fog (Figure 3.8 (a left)) as well as with the dense fog (Figure 3.8(a right)). Combination of different scatterings visualize more realistic results, but it should be considered depending on the art direction or the scene size.

\[
\begin{align*}
(a) & \quad \frac{1}{4\pi} \left( \frac{1}{2} + \frac{9}{2} \left( \frac{1 + \cos \theta}{2} \right)^4 \right) \\
(b) & \quad \frac{1}{4\pi} \left( \frac{1}{2} + \frac{33}{2} \left( \frac{1 + \cos \theta}{2} \right)^{32} \right) \\
(c) & \quad \frac{3}{16\pi} \left( 1 + \cos^2(\theta) \right) \\
(d) & \quad \frac{1}{4\pi} \left( 1 + g^2 - 2g \cos \theta \right)^{3/2}
\end{align*}
\]

![Figure 3.9 Equations for atmospheric scattering. (a) Mie-Lorenz light fog (b) Mie-Lorenz dense fog (c) Rayleigh scattering (d) Henyey-Greenstein phase function](image)
Figure 3.8 Fog comparisons between equations (left) and phase function $g$ value (right), (a row) dense fog, $g = 0.67$, (b row) light fog, $g = 0.59$, (c row) Rayleigh scattering, $g = 0$, (d row) left – combination of the Rayleigh sc. with the light fog, right – combination of the Rayleigh sc. with the dense fog.
4.4. Performance comparisons between different algorithms

To get an idea of performance costs between discussed volumetric lighting methods three of those have been tested in Unity scene. First approach was Wronsiki algorithm implemented by Kanapickas (Kanapickas, 2016) as an asset from the Unity store. This method gave the best looking results (Figure 3.10 (a)) and run at 45 FPS. Comparing with similar Raymarching solution of Vos, setup got slightly different results. Using original Skalsky (Skalsky, 2015) implementation of the Vos algorithm scene run at 30 FPS and produced softer results (Figure 3.10 (b)) with more light covering the scene. Finally, the post process implementation of Unity image effect package had really fast compute speed and run at 75 FPS, but visual results were looking dissatisfactory (Figure 3.10 (c)).

Furthermore, in contrast with these three methods polygonal volumetric lighting of Hoobler is slightly more expensive than Wronsiki algorithm. From resources information the Wronsiki algorithm takes 1.1 Ms to calculate on Xbox One graphic card and Hoobler algorithm takes 1.2 Ms on GTX 980Ti. Later video processor is 1.3 times faster than Xbox one. Based on that we see that Hoobler algorithm is more expensive.

Figure 3.10 Images above show different volumetric lighting implementation in Unity. (a) Wronsiek Raymarching method (45 FPS). (b) Vos Raymarching Method (30 FPS). (c) Unity post process sun shafts (75 FPS).
4.5. Evaluation

In this section a simple volumetric lighting implementation in Unity was developed and discussed. Unity Engine provided a lot of simplifications to make the algorithm possible (precalculated shadow maps, camera depth texture, possibility to interact with Unity rendering pipeline at wanted points etc.). Implementation was simple, but run with questionable compute speed. Following that an optimization strategy was introduced which cut the performance costs almost by half, resulting in almost the same visual results. To get a deeper understanding of atmospheric scattering possibilities experimentation with different scattering function were done. Results showed that combination of functions provided splendid looking volumetric lights. Finally, performance comparisons of different volumetric lighting approaches were introduced based on resources.

5. Conclusions

Throughout the research we tried to answer the question of how to implement volumetric lighting into Unity Engine. In Section 3 paper showed the approach which we thought was the best for the given time framework. In Section 4 paper answered questions about setup of the algorithm, optimization strategies, performance comparisons and experimentations. As a result, an easy Raymarching algorithm was developed. We can see that Unity Engine provides perfect opportunities to implement volumetric lighting algorithms. From Section 4.1 it is visible that using deferred rendering, few classes and custom Shaders results in a simple implementation of the researched method. Unity provided us with easy access to its rendering pipeline and good communication between Shaders and C# classes. Furthermore, Section 4.4 results showed that others implemented most popular Raymarching solutions in Unity with good performance and visual results. Based on that, research showed that Unity provides tools to develop magnificent atmosphere of volumetric lighting. So far we tried to answer a broad question, but with research and development we can say that to implement volumetric lighting into Unity Engine we need simple things: time and topic knowledge. With a good base of knowledge about atmospheric scattering and Unity pipeline we can implement different volumetric lighting algorithms with ease.

6. Discussion

In the process of research a lot of information was analysed. With bigger time frame we could optimise the setup to the point where it would not be heavy to compute at all. However, before further optimizing this setup, we would need to research what is more time consuming between this method optimizations and developing Wronski method from zero. Wronski algorithm uses compute Shaders and gives faster results that Vos method. But if we want to further develop current setup these techniques would prove to be useful:

**Interleaved Sampling** (Heidrich & Keller, 2000). The technique focuses on reducing number of sampling per pixel. Method makes the samples of a whole ray to be distributed among several adjacent pixels. In a simple explanation one pixel in a small grid of adjacent pixels will take some parts of the needed samples, while other pixels will take the rest (Figure 6.1 (b)).
**Epipolar Sampling** (Yusov, 2014). Compared to our approach when we reduced the screen resolution this method reduces resolution between Epipolar lines (Figure 6.1 (a)) and dramatically reduces the sample count for volumetric lighting.

![Figure 6.1](image-url)  
*Figure 6.1 (a) Epipolar sampling visualization of sampling (red) (b) Visualization of interleaved sampling. First image: no interleaved method; Second image: after applying method; Third image: applying blur effect*

In addition to that we should remember that our setup only works with point lights. To have all options of working with volumetric lighting we need to extend the setup to directional/spot lights.
7. References


