Informatik
Hauptcampus

HOCHSCHULE TRIER

Real-Time Volumetric Lighting with Volumetric Shadows

Volumetrische Beleuchtung mit volumetrischen Schatten in Echtzeit

Tim Dörries

Master's Thesis

Advisor: Prof. Dr. Christof Rezk-Salama

Trier, October 7, 2020

Abstract

Physically based solutions for volumetric lighting are a recent addition to many real-time rendering applications such as games. Most of these solutions do not account for shadows cast by the participating media themselves. The aim of this thesis is to detail the implementation of a physically based volumetric lighting technique with volumetric shadows. The proposed solution is based on the well-known Volumetric Fog algorithm and supports seamless integration of participating media volumes, billboard particles, transparent objects and the opaque scene. Focus is placed on improving the performance of computing in-scattered light. This is achieved by applying the concept of checkerboard rendering to Volumetric Fog. Volumetric shadows are realized by adapting Fourier Opacity Mapping to work with both particles and volumes. It is concluded that the discussed solution fulfills the initial goals and is able to provide improved visuals on modern hardware, while still achieving good results with acceptable performance on weaker systems.

Lösungen für volumetrische Beleuchtung mit physikalischem Ansatz zählen zu den neueren Errungenschaften vieler Echtzeitanwendungen, wie etwa Computerspiele. Viele dieser Lösungen vernachlässigen Schatten, welche vom Volumen selbst geworfen werden. Diese Arbeit detailliert die Implementierung einer auf physikalischen Ansätzen basierenden Technik für volumetrische Beleuchtung mit volumetrischen Schatten. Die vorgeschlagene Lösung baut auf dem bekannten Volumetric Fog Algorithmus auf und unterstützt die nahtlose Integration von Volumen, herkömmlichen Partikelsystemen, transparenten Objekten und der restlichen Szene. Besonderer Fokus liegt darauf, die Berechnung des eingestrahlten Lichts zu beschleunigen. Zu diesem Zweck wird das Konzept von Checkerboard Rendering auf Volumetric Fog übertragen. Durch das Anpassen von Fourier Opacity Mapping ist es zudem möglich, volumetrische Schatten für sowohl Partikelsysteme, als auch Volumen zu realisieren. Es stellt sich heraus, dass die beschriebene Lösung die gesetzten Ziele erfüllt und verbesserte visuelle Qualität auf moderner Hardware erzielt und dennoch gute Ergebnisse mit akzeptabler Leistung auf schwächeren Systemen erbringt.

Contents

List of Figures

List of Tables

1 Introduction

Traditionally, most real-time rendering applications only considered interactions between light rays and solid surfaces. However, in reality, light can also interact with particles in the air, resulting in phenomena such as smoke, fog, clouds, light shafts/god rays or the sky being blue. Since they participate in the light transport, volumes of particles are also referred to as participating media. Taking participating media into account when rendering a scene is known as volumetric lighting. Disregarding participating media was motivated by the often prohibitive cost of computing light-particle interactions in real-time. In order to still achieve the effects caused by participating media, real-time rendering applications like video games typically used approximations such as pre-rendered skyboxes, screen space effects or billboard particles. However, recent advances in graphics hardware capabilities have made different techniques for more accurate volumetric lighting feasible.

While volumetric lighting effects have become more common in video games in recent years, most implementations ignore shadows cast by participating media onto itself and the scene. This type of shadowing is also known as volumetric shadows. The aim of this thesis is to detail the implementation of a physically based real-time volumetric lighting effect with volumetric shadows. It should support directional lights, point lights and spot lights, integrate with both opaque and transparent scene elements and be performant enough to run reasonably fast even on older hardware. As there is already a large body of research on atmospheric effects such as atmospheric scattering and cloud rendering, these topics are outside of the scope of this thesis. Figure 1.1 shows how the proposed volumetric lighting solution might be used for a scene in a real-time rendering application.

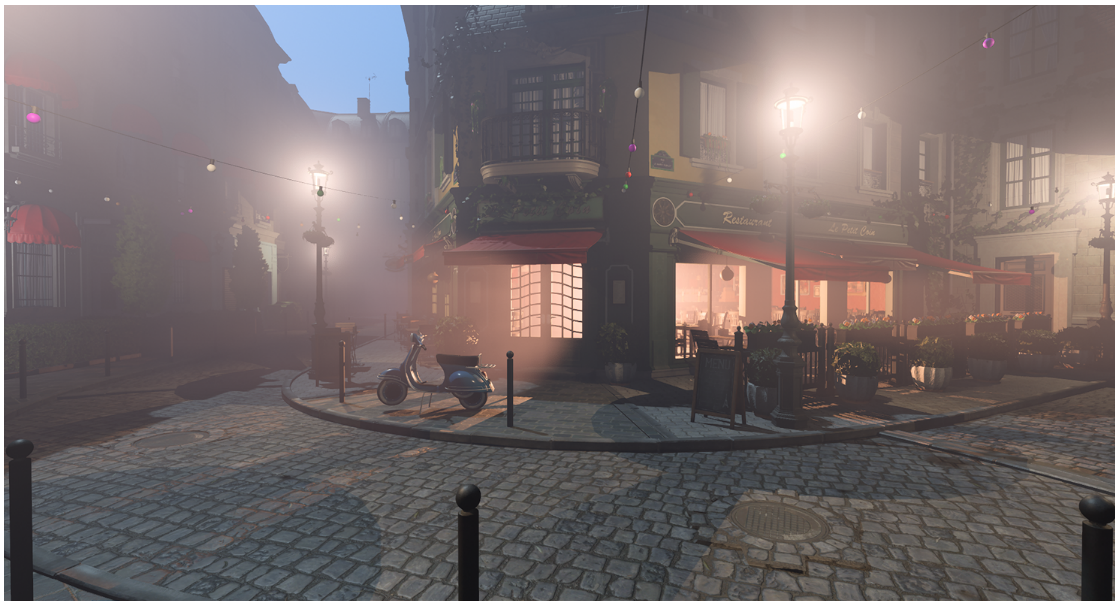


Fig. 1.1: Amazon Lumberyard Bistro scene [Lum17] lit with the volumetric lighting solution detailed in this thesis.

2

Related Work

2.1 Physical Basis

In order to understand the terminology and techniques discussed in the following sections, it is necessary to establish the physical basis of volumetric rendering. Fong et al. give a thorough introduction to volume rendering [FWKH17]:

2.1.1 Volume Properties

Participating media can be considered as volumes of particles of different sizes and densities. When a photon travels through such a volume, it may collide with a particle. Upon collision, the photon is either absorbed or scattered.
In the case of absorption, the energy of the photon is converted into heat or some other form of internal particle energy [FWKH17]. However, for the purpose of volumetric rendering, the photon can be considered to have disappeared.
If the photon is not absorbed but instead scattered, it retains all its energy and simply changes the direction of its path. The new direction is dependent on the phase function of the medium.
The phase function is the angular distribution of scattered light/radiance and depends on the angle θ between the initial direction ω and the direction $\omega^{'}$ after the scattering event [FWKH17]. Phase functions need to be reciprocal and normalized over the sphere [FWKH17]. In a way, they are the equivalent of a bidirectional reflectance distribution function (BRDF) for volumes. Isotropic media have an equal probability of scattering incoming photons in any direction. This behavior can be modeled by the phase function in equation 2.1, where x is the position in the volume and θ the angle between the two directions ω and $\omega^{'}$. Note that this function is constant and thus independent of both parameters.

$$f_p(x, \theta) = \frac{1}{4\pi} \tag{2.1}$$

Participating media with anisotropic scattering behavior require a phase function that takes θ into account. A popular choice is the Henyey-Greenstein phase function shown in equation 2.2, where $-1 < g < 1$ [Hil15][Wro14]. The parameter g

describes the anisotropy of the phase function. A positive value models forward scattering, a value of 0 corresponds to isotropic scattering and a negative value causes backwards scattering. Multiple lobes of the Henyey-Greenstein phase function can be used to model complex scattering behavior [HG41].

$$f_p(x,\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g\,cos(\theta))^{\frac{3}{2}}} \tag{2.2}$$

Under certain circumstances, participating media can emit light. Photons are emitted in all directions and behave the same as photons emitted by any other light source. Figure 2.1 visualizes the four types of photon-particle interactions.

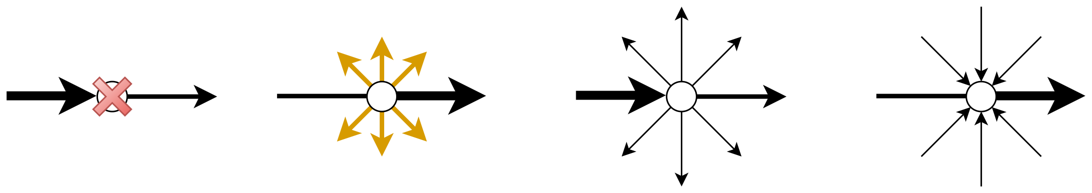


Fig. 2.1: Overview of the four different types of photon-particle interactions. From left to right: Absorption, Emission, Out-Scattering, In-Scattering

Since it is infeasible to track individual particles and photons for the purpose of real-time volume rendering, collisions are modeled stochastically. The chance of a photon collision is defined by a coefficient $\sigma(x)$, which is the probability density of collision per unit distance traveled inside the volume [FWKH17]. Going back to the participating media properties discussed above, they can be parametrized as follows:

- *Absorption* is modeled with the coefficient $\sigma_a(x)$.
- *Scattering* is modeled with the coefficient $\sigma_s(x)$.
- On account of its simplicity, this work uses the Henyey-Greenstein *phase function*. Thus the parametrization is the phase anisotropy parameter g.
- *Emission* is modeled by a radiance field $L_e(x)$.

For the purpose of authoring participating media, it may be desirable to use a more intuitive parametrization than scattering and absorption coefficients.

Both the scattering and the absorption coefficient model a loss of energy along a certain path. Since it is often easier to author participating media with a single parameter controlling the attenuation, both coefficients can be represented by a

single coefficient $\sigma_t = \sigma_s + \sigma_a$. This coefficient σ_t is referred to as *extinction* (and is sometimes informally called density or attenuation factor) [FWKH17].
Single scattering albedo $\alpha = \sigma_s/\sigma_t$ defines how much light is absorbed or scattered in a medium. A value of 1.0 signifies that every collision acts as a scattering event and no light is absorbed (such as in clouds), whereas a value of 0.0 models the opposite case where every collision is an absorption event (e.g. black coal dust). Single scattering albedo allows to author participating media properties more intuitively as it has a similar meaning to the surface albedo in that both are a measure of overall reflectivity determining the amount of scattered light [FWKH17].

To summarize, absorption and scattering coefficients can alternatively be authored with single scattering albedo and an extinction coefficient. Both parametrizations are equivalent and can be converted into one another. It should be noted that light is a spectrum, which means that participating media may have different absorption or scattering coefficients for every wavelength. However, in real-time rendering typically only three wavelengths are considered (red, green, blue). For this reason, absorption and scattering coefficients are often RGB-triples. Table 2.1 gives an overview of the symbols introduced so far.

Symbol	Description
$\sigma_a(x)$	Absorption Coefficient
$\sigma_s(x)$	Scattering Coefficient
$\sigma_t(x)$	Extinction Coefficient ($\sigma_s + \sigma_a$)
α	Single Scattering Albedo (σ_s/σ_t)
$f_p(x, \theta)$	Phase Function

Table 2.1: Overview and description of the introduced symbols.

2.1.2 Light Propagation in Volumes

Using the terminology and principles established above, it is now possible to discuss how light is propagated in volumes.

Radiative Transfer Equation

The radiative transfer equation (RTE) [Cha60] defines the distribution of radiance in volumes. Focusing on a single light beam $L(x, \omega)$ starting at position x with direction ω, the RTE can be derived by describing its individual components.

The derivative of the beam $L(x, \omega)$ in the direction of ω (expressed as $(\omega \cdot \nabla)$) is proportional to the radiance at that point. The proportionality factor is the absorption coefficient σ_a (see equation 2.3) [FWKH17]. This term models the loss of energy due to absorption.

$$(\omega \cdot \nabla)L(x,\omega) = -\sigma_a(x)L(x,\omega) \tag{2.3}$$

Out-scattering describes the loss of energy due to radiance being scattered out of the radiance beam. Similar to absorption, this loss is also proportional to the radiance at the current position. The proportionality factor is the scattering coefficient σ_s. Equation 2.4 shows this term. Note the similarity to equation 2.3, making the motivation for introducing the extinction coefficient clear.
Out-scattered energy is not lost to the system, only to the current beam. Instead, it is scattered into other beams as part of the in-scattering term.

$$(\omega \cdot \nabla)L(x,\omega) = -\sigma_s(x)L(x,\omega) \tag{2.4}$$

In-scattering happens when radiance is scattered into the direction of the current beam, increasing its radiance. The radiance may originate from another beam where it was out-scattered or it may even come directly from a light source. Since radiance might be in-scattered from any direction, an integral over the sphere (expressed by S^2) around the position x is required (see equation 2.5). Similar to equation 2.4, the scattering coefficient σ_s is used to model the scattering of incoming radiance.

$$(\omega \cdot \nabla)L(x,\omega) = \sigma_s(x)\int_{S^2} f_p(x,\omega,\omega^{'})L(x,\omega^{'})d\omega^{'} \tag{2.5}$$

Emission is modeled with a separate radiance field $L_e(x)$ that defines the radiance added to the radiance field $L(x,\omega)$.

$$(\omega \cdot \nabla)L(x,\omega) = \sigma_a(x)L_e(x) \tag{2.6}$$

Putting these terms together and combining the absorption and out-scattering terms using the previously established extinction coefficient gives the full RTE, shown in equation 2.7.

$$(\omega \cdot \nabla)L(x,\omega) = -\sigma_t(x)L(x,\omega) + \sigma_a(x)L_e(x) + \sigma_s(x)\int_{S^2} f_p(x,\omega,\omega^{'})L(x,\omega^{'})d\omega^{'} \tag{2.7}$$

Volume Rendering Equation

For path tracing but also for real-time rendering techniques, there is a more useful formulation of the RTE: the volume rendering equation (VRE) [KvH84]. While the RTE uses gradients to describe the flow of radiance in a forward fashion, the VRE uses integrals to describe where the radiance is coming from. Integrals can be evaluated with Monte-Carlo methods, making the VRE more suitable for techniques such as path tracing.

In order to shorten the notation, the in-scattering term is replaced with $L_s(x,\omega)$ as seen in equation 2.8. Note that the scattering coefficient $\sigma_s(x)$ is not included here to make the final equation more clear.

$$L_s(x,\omega) = \int_{S^2} f_p(x,\omega,\omega^{'})L(x,\omega^{'})d\omega^{'} \tag{2.8}$$

In addition, the term $L_d(d_d,\omega)$ is introduced. This is the light entering the ray at its end point, modeling an opaque surface or another volume. The VRE is then defined as shown in equation 2.9, where $x_d = x - d\omega$ refers to the end point of the ray and $x_t = x - t\omega$ and $x_s = x - s\omega$. Note that with this notation, ω still points into the direction of the radiance flow.

$$L(x,\omega) = \int_{t=0}^{d} exp\Big(-\int_{s=0}^{t}\sigma_t(x_s)ds\Big)\Big[\sigma_a(x)L_e(x_t) + \sigma_s(x)L_s(x_t,\omega) + L_d(x_d,\omega)\Big]dt \tag{2.9}$$

Replacing the exponential integral term with $T(t)$ as defined in equation 2.10 allows to further shorten the VRE. $T(t)$ is known as the *transmittance* term and models the net reduction factor from absorption and out-scattering along a ray segment bounded by x and x_t.

$$T(t) = exp\Big(-\int_{s=0}^{t}\sigma_t(x_s)ds\Big) \tag{2.10}$$

Furthermore, since it is not dependent on t, the $L_d(x_d,\omega)$ term can be moved out of the integral, giving the final formulation of the RTE (see equation 2.11).

$$L(x,\omega) = \int_{t=0}^{d} T(t)\Big[\sigma_a(x)L_e(x_t) + \sigma_s(x)L_s(x_t,\omega)\Big]dt + T(d)L_d(x_d,\omega) \tag{2.11}$$

2.1.3 Application in Real-Time Rendering

Since evaluating the in-scattering integral in equation 2.8 is often too expensive for real-time rendering, out-scattered light from other light beams is commonly ignored. Instead, the integral is replaced with a sum over the direct contributions of all light sources, as shown in equation 2.12 [Hil15]. As a consequence, all out-scattered light is lost to the system. This restriction is known as single scattering.

$$L_{singleScattering}(x,\omega) = \Sigma_{l=0}^{lights} f_p(x,\omega,l)Vis(x,l)L(x,l) \tag{2.12}$$

Note the introduction of the $Vis(x,l)$ term. This is the visibility function, modeling shadows cast by opaque objects, as well as the attenuation caused by the transmittance of the path between x and the light source position (see equation 2.13). For the purpose of this thesis, the latter is also referred to as volumetric shadows. Volumetric shadows effectively describe the shadows cast by participating media onto themselves and then the scene.

$$Vis(x,l) = opaqueShadowFactor(x,l) * volumetricShadowFactor(x,l) \tag{2.13}$$

2.2 Volumetric Lighting

While volumetric shadows are technically a part of volumetric lighting, this section focuses on the related work of rendering participating media itself. Volumetric shadows are discussed in section 2.3.

2.2.1 Billboard Particles

Billboard particles are a popular choice for rendering participating media such as smoke. They are implemented by rendering camera facing rectangles with a smoke texture applied to them. This effect can be improved by applying lighting to the particles. Jansen et al. demonstrate how lighting can be computed at lower resolution by evaluating it in the domain shader (tessellation evaluation shader) [JB11]. Sousa et al. render lighting to an atlas texture that is sampled and combined with the particle texture during particle rendering. The advantage of this approach is that it avoids the use of tessellation and allows for better upsampling by using a bicubic filter to sample the lighting atlas [SG16]. Drobot et al. approximate multi-scattering by blurring the lighting atlas [Dro17b].

2.2.2 Screen Space Post-Processing Effect

Mitchell presents an inexpensive method for approximating the effect of volumetric lighting as a post-processing effect in screen space [Mit08]. The technique is mainly intended for rendering sun light crepuscular rays. For each pixel, a ray is marched in the direction of the screen space position of the light source. At each ray marching iteration, the rendered scene is sampled. All samples belonging to the sky are weighted by their distance to the light source and summed up. The final sum is then additively blended with the rendered image. While this approach is very performant, it suffers from a number of limitations. As it relies on screen space information, the effect breaks down if the light source is not visible. Additionally, light shafts are not always properly occluded by on-screen geometry.

2.2.3 Ray Marching

Toth et al. propose a ray marching technique to compute single-scattering in homogeneous participating media [TU09]. They cast rays from the closest depth in the scene toward the camera and compute the in-scattered light at each point on the ray. Lights are shadowed by shadow maps and analytically calculated transmittance towards the light source. In-scattered light is additionally modulated by transmittance towards the camera. All samples acquired this way are added and applied to the scene. In order to reduce the cost of ray marching, Toth et al. use interleaved sampling: Each ray in a neighborhood takes samples at different depths, so that the final result of multiple rays can be combined to give a higher effective sample count. Multiple light sources can be supported by repeating this process for each light and additively applying it to the scene. The primary downside of this technique is that it is not trivial to apply to transparent objects.

Sousa et al. use ray marching to compute volumetric lighting for their video game *Crysis 3* [SWR13]. Density calculation of participating media is based on the fog model proposed by Wenzel [Wen06][Wen07]. Their volumetric lighting is then shadowed by accumulating the shadow contribution along the view ray. This is similar in spirit to the method proposed by Toth et al. [TU09]. Sousa et al. also adopt interleaved sampling: They distribute 1024 samples on a 8x8 grid of half resolution pixels. A gather pass then bilaterally filters and upscales these results.

For the game *Lords of the Fallen*, Glatzel et al. built a volumetric lighting solution for many lights [Gla14], loosely based on the work of Toth et al. [TU09]. They render proxy geometry for the lights to spawn pixel shader invocations which perform the ray marching. The ray is constrained to the bounds of the light source, increasing precision with the same number of samples. They extend this system by adding support for 2D projector textures, 3D noise textures, IES profiles for light sources, as well as anisotropic phase functions. Glatzel et al. additionally employ temporal reprojection to stabilize the results.

The volumetric lighting of the video game *Killzone: Shadow Fall* presented by Valient et al. [Val14a] [Val14b] is also based on view space ray marching, constrained to the bounds of each light source and using interleaved sampling. They improve upon the visuals of the effect by modulating the intensity of ray marching samples with particles that have been rendered to a 3D texture covering the view frustum with a quadratic depth distribution. This texture is at 1/8th of the native resolution and has 16 depth slices. Raymarched volumetric lighting is usually difficult to correctly compose with transparent objects. Valient et al. solve this problem by creating a 3D texture that can be queried for the amount of volumetric lighting between a point in the scene and the camera. This texture is then used to properly blend transparent objects, the opaque scene and volumetric lighting.

2.2.4 Volumetric Fog

Wronski proposes a novel solution for volumetric lighting: *Volumetric Fog* [Wro14]. This technique is truly volumetric in that it produces a 3D texture fitted to the view frustum. This texture contains in-scattering and extinction values that can be used to apply the effect at any point in the scene that is covered by the texture. Wronski uses 64 slices exponentially distributed over a range of 64 meters. Thus, one of the advantages of this approach is the fact that Volumetric Fog can be applied to all types of objects: deferred or forward rendered opaque objects, transparent objects and billboard particles. The volumetric fog texture is generated by first creating a 3D texture holding the density of the participating media in the scene. In-scattered light is computed and stored in another 3D texture. These computations can be either combined or ran in parallel. A final compute shader pass ray marches through the intermediate 3D textures and stores accumulated in-scattering and extinction for every slice along the view ray. The downside of storing volumetric lighting in a 3D texture is that high volume resolutions are prohibitively expensive, both in terms of memory and computation. For this reason Wronski uses a resolution of 160x90x64. While the low resolution along the X/Y-plane is not very noticeable, the small number of slices along the Z-axis is. One solution to this problem proposed by Wronski is to use temporal reprojection similar to Temporal Anti-Aliasing (TAA) to temporally filter the 3D texture.

Hillaire builds upon Volumetric Fog and proposes using physically based parameters such as absorption, scattering, Henyey-Greenstein phase function anisotropy and emission to author participating media [Hil15]. He notes that most of these parameters can be additively blended. While Wronski's Volumetric Fog uses a single global participating medium, Hillaire voxelizes material parameters of multiple media into 3D textures, similar in spirit to a G-Buffer. A second pass computes the in-scattered light at each texel of the 3D texture and a final pass prepares the data for rendering, similar to Volumetric Fog. Additionally, Hillaire presents a new way of integrating the lighting with respect to transmittance over the distance between two samples along the view ray. This leads to improved visuals when participating media exhibit strong scattering.

Delmont et al. also adapt Volumetric Fog for their game *Rise of the Tomb Raider* [DS15]. In addition to temporal filtering, they propose using a 2x2 Bayer matrix to add dithering along the Z-axis of the volume. In order to filter out the noise caused by dithering, they apply a small blur to the 3D texture along the X/Y-plane.

Drobot notes that the usual passes of the Volumetric Fog technique (volume material voxelization, lighting, integration) are all bandwidth bound. He suggests merging those passes into a single compute shader, considerably improving performance [Dro17b]. Performing the integration step in the same shader as the lighting necessitates a new algorithm. Drobot leverages wave-level operations, available on consoles and newer PC graphics APIs, to achieve this.

Lagarde et al. base their volumetric lighting solution on Volumetric Fog, introducing a new algorithm for voxelizing participating media, which exhibits reduced aliasing by computing partial coverage of each voxel [LG18]. Additionally they use Monte Carlo integration methods to evaluate the in-scattering integral of each voxel. However, the final ray marching pass along the view ray is still done as in the original technique. Similar to other implementations of Volumetric Fog, Lagarde et al. use temporal filtering. In order to reduce ghosting caused by view dependency of anisotropic phase functions, they propose additionally storing lighting results computed with an isotropic phase function. This second set of data is used for temporal filtering, such that anisotropy is only introduced for the current frame.

Instead of using a single bilinear texture sample to apply the effect to the scene, they propose a biquadratic filter with four bilinear texture samples.

Bauer et al. present their system for volumetric lighting used in the game *Red Dead Redemption 2* [Bau19]. In contrast to the usual temporal filter applied to the in-scattering texture, they temporally filter the volume material texture and a special 3D texture holding the sun light shadow. Additionally, they use blue noise to offset the texture coordinate when sampling the final result texture. TAA is then used to blur out the noise. This effectively combats aliasing introduced by undersampling the sun light shadow maps without suffering from ghosting due to anisotropic phase functions. Their participating media can be authored as three distinct types that control how their material parameters are blended. They support additive blending, alpha blending and particles. The first type corresponds to the way of blending materials as proposed by Hillaire [Hil15]. The second type is used to place smaller participating media volumes inside of larger volumes. This is useful for creating different participating media properties for interiors. Particles are used analogously to billboard particles and are blended last. Clouds and long range volumetric lighting past the limit of Volumetric Fog are achieved with ray marching. For performance reasons, local lights and local participating media are ignored during ray marching. This pass produces a half-resolution image by temporally upsampling ray marched results at quarter-resolution.

Cho et al. also use ray marching for long range volumetric lighting and introduce a novel way of upscaling the result from quarter-resolution to full resolution [CGK19]: They sort all 16 full resolution depth samples corresponding to a quarter resolution ray by depth and write out intermediate ray marching results to each full resolution pixel once its respective depth has been reached.

2.3 Volumetric Shadows

Volumetric shadows are shadows that are cast by a participating medium onto itself, other participating media and opaque surfaces. Since they are such an important aspect of realistic volumetric lighting, there is a large body of research about this topic. A selection of influential works in this area is discussed in this section. Solutions for volumetric shadows must be able to determine the value of the transmittance function at any distance from the light source. Most volumetric shadow techniques use some form of shadow mapping, where a transmittance function is stored per shadow map texel.

Deep Shadow Maps by Lokovic et al. [LV00] is one of the most well known techniques for volumetric shadows. It builds on the classic shadow mapping algorithm by storing transmittance values for multiple depths. Generating these depth/-transmittance value pairs involves managing per-pixel linked lists with unbounded memory requirements, which is why this technique is mostly used in offline rendering.

An alternative to Deep Shadow Maps are *Opacity Shadow Maps* (OSM) by Kim et al. [KN01]. Opacity Shadow Maps are similar to Deep Shadow Maps in that they also store transmittance values for multiple depths. The key difference is that OSM store a fixed number of transmittance values for fixed depths. Since the depths are fixed, they do not need to be stored explicitly. In practice Opacity Shadow Maps can be implemented using 3D textures or texture arrays. This means that the memory requirements are not unbounded, making this technique suitable for real-time rendering. The limitation of OSM is that a large number of texture slices may be required to achieve good results.

Half-Angle Slicing, introduced by Kniss et al. [KPHE02], is an entirely different approach to volumetric shadows. This technique involves alternating rendering slices of a participating medium into a shadow map and into the final image, using the intermediate shadow map results to shadow each slice of the main view. Half-Angle Slicing is mostly used in volume rendering applications, where there are few light sources and no opaque scene geometry [EMK$^+$06]. The downside of Half-Angle Slicing is that it requires changing the render target for every slice, which makes this technique unsuitable for high performance real-time applications.

Deep Opacity Maps by [YK08] are an improved version of OSM where the transmittance is stored only for depths past the closest translucent object. This technique requires fewer slices and therefore less memory to achieve the same quality as Opacity Shadow Maps. It is suited for rendering objects such as hair, where finding the depth of the closest translucent object can be trivially achieved using a depth buffer. However, this cannot be easily done for participating media such as smoke or clouds.

Salvi et al. introduce *Adaptive Volumetric Shadow Maps* and a new streaming compression algorithm in order to implement Deep Shadow Maps on a GPU [SVLL10]. Their algorithm uses atomic operations in the pixel shader to maintain the required per-pixel linked lists. These lists are sorted and compressed using a compression algorithm that generates a fixed number of nodes with variable error. This is in contrast to Deep Shadow Maps, where the compression step generates a variable number of nodes with a fixed upper bound on the error.

Fourier Opacity Mapping (FOM) by Jansen et al. [JB10] is a new approach to volumetric shadows, where the transmittance function is projected into a fourier basis using a limited set of coefficients. A FOM is created by additively rendering particles into a shadow map storing coefficients. These coefficients are generated by projecting the local transmittance (inverse opacity of the particle) into a fourier basis. The resulting coefficients can be used to reconstruct the transmittance function for all depths. The advantages of this technique are that it gives a smooth, continuous transmittance function and that it does not require sorting of the particles. Unlike many other algorithms for volumetric shadows, FOM are prefilterable. The disadvantage of FOM and other algorithms that use the same principle is that the resulting transmittance function tends to exhibit ringing and false self shadowing if the covered range is too large or if particles are almost opaque. The reason for this is that the reconstructed transmittance function is a sum of sine and cosine waves, with each additional coefficient adding waves of higher frequency. Limiting the number of coefficients preserves low frequency waves but fails to capture high frequency detail in the signal.

Another technique of the same class of algorithms as FOM are *Extinction Transmittance Maps* by Gautron et al. [GDM11]. Instead of using a discrete fourier transform (DFT), a discrete cosine transform (DCT) is used. This technique builds on *Transmittance Function Mapping*, introduced by Delalandre et al. [DGMF11]. In contrast to FOM and Extinction Transmittance Maps, Transmittance Function Mapping is based on marching through a participating medium to generate the DCT coefficients. Since particles are not necessarily sorted by their distance along the ray being ray marched, Transmittance Function Mapping is not trivially applicable to particles. Transmittance Function Mapping has been used for real-time visualization of special effects in movies [EPK12] [EPG$^+$12].

While it is not a solution to the general problem of volumetric shadows, Kasyan et al. [KSS11] and Persson [Per12] propose a simple solution for having particles cast shadows on opaque objects. They render the opacity of particles with alpha blending into an 8-bit render target that is looked up similar to a shadow map during rendering of the main view. In order to avoid back-projection (casting shadows on objects closer to the light than the particles), they use depth buffering while rendering the translucent shadow map.

Bavoil et al. present a version of Opacity Shadow Maps which can be used for particles [BJ13]. They call their solution *Particle Shadow Mapping*. It involves using a geometry shader to render particles to different render target layers of a layered framebuffer. Local particle transmittance is then multiplicatively blended into the render target. In a next step, a compute shader generates the global transmittance function by marching through the slices of the texture and multiplicatively accumulating the local transmittance.

Hillare proposes a unified solution for both particles and volumes [Hil15]. Extinction of particles and volumes is voxelized into three cascaded volume textures centered around the camera. For each light with volumetric shadows, a small Opacity Shadow Map is created by ray marching through the voxelized extinction with a compute shader. Particles are voxelized using atomic operations. Being a version of Opacity Shadow Maps, this technique exhibits the same limitations as OSM.

3

Implementation

3.1 Implementation Framework

The techniques in this thesis have been implemented in a custom framework using C++ and the Vulkan 1.2 graphics API. All shaders have been authored in HLSL and compiled to SPIR-V using the DirectX Shader Compiler. SPIR-V is the byte code shader format that is consumed by the Vulkan runtime. The framework uses a clustered forward renderer and includes features such as physically based rendering, real-time reflection probes, shadowed point-, spot- and directional lights and Temporal Anti-Aliasing (TAA).

3.2 Volumetric Lighting

As outlined in section 2.2, current state of the art solutions for volumetric lighting are based on either ray marching or Volumetric Fog. As Volumetric Fog can be applied to any object, including transparent objects and billboard particles, it was chosen as a basis for this thesis. In this section the initial implementation of Volumetric Fog is described first. This first version of the effect only supports directional lights and a single global participating medium. Afterwards, incremental improvements on the basic effect are detailed, finally arriving at the full implementation. Volumetric shadows are discussed in section 3.3.

3.2.1 Participating Media Materials

Similar in spirit to Hillaire [Hil15], participating media are parametrized by *albedo*, *extinction*, *emissive* and *phase anisotropy*. In particular, albedo and emissive are RGB-tuples, while extinction is a monochromatic value. Extinction could trivially be expanded to also be an RGB-tuple, however this would consume more memory and bandwidth, especially with respect to volumetric shadows, where the memory requirements would be tripled. While most participating media do not emit light, the emissive parameter can be useful for artists to fake indirect light [Hil15]. *phase anisotropy* describes the eccentricity parameter of the Henyey-Greenstein phase function. Participating media can also be configured to have height dependent density, an effect known as height fog. In addition, an optional 3D texture can be used to vary the density inside the volume. Table 3.1 summarizes these parameters and their types.

Parameter Name	Type
Albedo α	float3
Extinction σ_t	float
Phase Anisotropy g	float
Emissive	float3
Density Texture	Texture3DHandle
Height Fog Enabled	bool
Height Fog Start Height	float
Height Fog Falloff	float

Table 3.1: Material parameters of participating media.

3.2.2 Initial Implementation

The initial implementation of the Volumetric Fog technique was closely based on the work of Wronski [Wro14] and Hillaire [Hil15]. Initially, only directional lights and a single global participating medium were supported.
The Volumetric Fog algorithm makes use of multiple intermediate 3D textures and a final result 3D texture. These textures are fitted to the view frustum and cover a range of 64 meters. The individual slices of the 3D textures use an exponential distribution along the Z-axis, resulting in a higher resolution close to the camera. The algorithm can be broken down into three distinct steps. First, the properties of the participating media are voxelized and stored in one of the frustum-aligned textures. A second step reads this data and computes the in-scattered light. In a final step, the result texture is created by marching through the texture generated by the previous step and accumulating in-scattered light and transmittance for every texel of the final texture. This process is similar to creating a prefix sum. These steps are visualized in figure 3.1. Their details are discussed in the following.

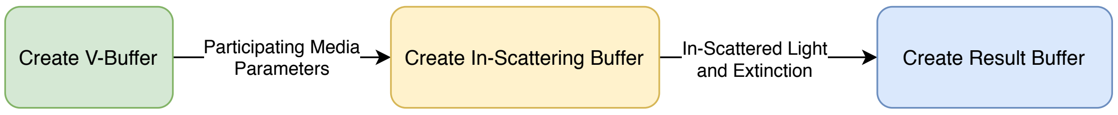

Create V-Buffer
Participating Media Parameters
Create In-Scattering Buffer
In-Scattered Light and Extinction
Create Result Buffer

Fig. 3.1: Overview of the Volumetric Fog shaders and the flow of data.

V-Buffer

The Volume-Buffer (V-Buffer) is named after the Geometry-Buffer (G-Buffer) used in deferred shading. Similar to a G-Buffer, which stores material parameters of the closest surface, it holds the material parameters of the participating media in the frustum. In contrast to a G-Buffer, a V-Buffer needs to store values for any point in space and therefore uses a 3D texture, not a 2D texture. While the original Volumetric Fog technique only stores volume density, the implementation for this thesis stores *albedo*, *extinction*, *emissive* and *phase anisotropy* in a set of two RGBA16F 3D textures according to the scheme shown in table 3.2.

	RGB	A
Texture A	Albedo	Extinction
Texture B	Emissive	Phase

Table 3.2: V-Buffer Layout

The V-Buffer is filled by dispatching a compute shader such that each thread computes the set of parameter values for one V-Buffer texel and writes them to the corresponding locations in the two textures. Since the initial implementation supported only a single global volume, computing the participating media parameters involves only looking up the parameters of the volume and applying height fog and an optional density texture. For completeness, this is shown in listing 3.1.

```
ConstantBuffer<Constants> g_Constants;
Texture3D g_Textures[TEXTURE_ARRAY_SIZE];
SamplerState g_LinearSampler;
RWTexture3D<float4> g_VBufferA;
RWTexture3D<float4> g_VBufferB;

float volumetricFogGetDensity(GlobalParticipatingMedium medium,
                              float3 position)
{
    float density = position.y > medium.maxHeight ? 0.0 : 1.0;
    // height fog
    if (medium.heightFogEnabled != 0 && position.y > medium.heightFogStart)
    {
        float h = position.y - medium.heightFogStart;
        density *= exp(-h * medium.heightFogFalloff);
    }
    // density texture
    if (medium.densityTexture != 0)
    {
        float3 uv = frac(position * medium.textureScale + medium.textureBias);
        density *= g_Textures[medium.densityTexture - 1].
                              SampleLevel(g_LinearSampler, uv, 0.0).x;
    }
    return density;
}

[numthreads(4, 4, 4)]
void main(uint3 threadID : SV_DispatchThreadID)
{
     const float3 position = calcWorldSpacePos(threadID);
     float density = volumetricFogGetDensity(g_Constants.medium, position);
     float3 scattering = g_Constants.medium.scattering * density;
     float extinction = g_Constants.medium.extinction * density;
     float3 emissive = g_Constants.medium.emissive * density;

     // RGB: albedo, A: extinction
     g_VBufferA[threadID] = float4(scattering / extinction, extinction);
     // RGB: emissive, A: phase anisotropy
     g_VBufferB[threadID] = float4(emissive, g_Constants.medium.phase);
}
```

Listing 3.1: Calculation of the V-Buffer.

In-Scattering Buffer

The In-Scattering Buffer holds the light which reaches each volume texel and is scattered towards the camera. It is stored in the RGB components of a RGBA16F 3D texture. Additionally, in order to avoid sampling multiple textures in later passes, extinction is copied from the V-Buffer and stored in the alpha channel. Calculating in-scattered light involves multiple steps. First, the participating media parameters of the current texel need to be sampled from the V-Buffer. Then the in-scattered light needs to be summed up by iterating over all light sources, computing the portion of the light which reaches the position of the current texel and then applying a phase function to it. Recalling the definition of the VRE (see equation 2.11), the incoming light needs to be multiplied by the scattering coefficient σ_s to arrive at the light scattered towards the camera. Since albedo is defined as $\alpha = \sigma_s/\sigma_t$, the scattering coefficient can be recovered by multiplying α with the extinction coefficient σ_t. The implementation of these steps is shown in listing 3.2.

```
ConstantBuffer<Constants> g_Constants;
RWTexture3D<float4> g_ResultImage;
Texture3D<float4> g_VBufferA;
Texture3D<float4> g_VBufferB;

[numthreads(4, 4, 4)]
void main(uint3 threadID : SV_DispatchThreadID)
{
    const float3 texelCoord = calcTexelCoord(threadID);
    const float3 worldSpacePos = calcWorldSpacePos(texelCoord);
    const float4 albedoExtinction = g_VBufferA.Load(threadID);
    const float4 emissivePhase = g_VBufferB.Load(threadID);
    const float3 V = normalize(g_Constants.camPos - worldSpacePos);

    // sum inscattered lighting
    float3 lighting = emissivePhase.rgb;

    // directional lights
    for (uint i = 0; i < g_Constants.directionalLightCount; ++i)
    {
        DirectionalLight directionalLight = g_DirectionalLights[i];
        lighting += directionalLight.color
        * henyeyGreenstein(V, directionalLight.direction, emissivePhase.w);
    }

    // shadowed directional lights
    for (uint i = 0; i < g_Constants.directionalLightShadowedCount; ++i)
    {
        DirectionalLight directionalLight = g_DirectionalLightsShadowed[i];
        float shadow = getShadow(directionalLight, worldSpacePos);
        lighting += directionalLight.color
        * henyeyGreenstein(V, directionalLight.direction, emissivePhase.w)
        * shadow;
    }

    const float scatteringCoeff = albedoExtinction.rgb * albedoExtinction.a;
    lighting *= scatteringCoeff;

    g_ResultImage[threadID] = float4(lighting, scatteringExtinction.w);
}
```

Listing 3.2: Calculation of in-scattered light.

As pointed out by Drobot [Dro17b], the V-Buffer does not need to be stored explicitly. Instead, voxelizing participating media parameters and computing in-scattered light can be merged into a single shader. This keeps the V-Buffer data in shader registers and avoids having to write to and read from intermediate V-Buffer textures. As a result, this optimization saves bandwidth. The downside is that this merged shader must be able to evaluate all participating media materials, resulting either in an über-shader or a smaller set of fixed functionality, potentially limiting artistic freedom. This may be a problem if hand-authoring material shaders for participating media is desired. For the sake of performance, the restriction to a limited set of fixed functionality was chosen for this thesis. Figure 3.2 visualizes how merging both shaders simplifies the pipeline.

Integrated Scattering Buffer

The goal of the Volumetric Fog technique is to produce a 3D texture which can be used to query the in-scattered light and the transmittance for the ray segment defined by the camera position and any point in the frustum. This 3D texture is called

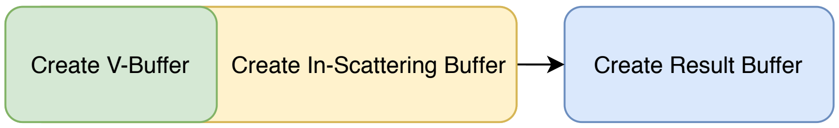


Fig. 3.2: The V-Buffer shader can be merged with the In-Scattering shader.

the (Integrated) Scattering Buffer. Having computed the In-Scattering Buffer in the previous pass, the result image can be computed by marching through the slices of the In-Scattering Buffer. This ray marching process starts at the first slice at the near plane of the camera. At each step the data from the next slice is read, accumulating in-scattered light and transmittance at each iteration. At the end of each iteration, the accumulated results are written to the corresponding location in the resulting 3D scattering image. For the accumulation step, the in-scattered light at each slice must be modulated by the transmittance along the ray segment between the current slice and the camera position before it can be added to the total in-scattered light along the ray. The transmittance itself must also be updated at every step.

A simple way to do this would be to update the accumulated in-scattering value by multiplying the in-scattering of the current slice with the accumulated transmittance and then adding it to the accumulated in-scattering. In a second step, the transmittance would be multiplicatively accumulated. Listing 3.3 demonstrates this:

```
float4 scatterStepSimple(float3 accumLight, float accumTransmittance,
             float3 sliceLight, float sliceExtinction, float stepLength)
{
    sliceExtinction = max(sliceExtinction, 1e-5);
    float sliceTransmittance = exp(-sliceExtinction * stepLength);

    accumLight += sliceLight * accumTransmittance;
    accumTransmittance *= sliceTransmittance;

    return float4(accumLight, accumTransmittance);
}
```

Listing 3.3: Simple accumulation step.

However, Hillaire notes that this is incorrect, as the in-scattered light of the current slice also depends on the extinction of the current slice. Given a single light sample S and extinction sample σ_t, he proposes to integrate the in-scattered light analytically with respect to transmittance over the depth interval D corresponding to a texel in the volume [Hil15]. Equation 3.1 gives the details:

$$\int_0^D e^{-\sigma_t x} S \, dx = \frac{S - S e^{-\sigma_t D}}{\sigma_t} \tag{3.1}$$

Since this approach is more correct, it has been widely adopted in other work and is also used in the implementation of this thesis. Listing 3.4 shows how the calculation of the Integrated Scattering Buffer is implemented.

```
RWTexture3D<float4> g_ResultImage;
Texture3D<float4> g_InputImage;

float4 scatterStep(float3 accumulatedLight, float accumulatedTransmittance,
            float3 sliceLight, float sliceExtinction, float stepLength)
{
    sliceExtinction = max(sliceExtinction, 1e-5);
    float sliceTransmittance = exp(-sliceExtinction * stepLength);

    float3 sliceLightIntegral = (-sliceLight * sliceTransmittance
                         + sliceLight) * rcp(sliceExtinction);

    accumulatedLight += sliceLightIntegral * accumulatedTransmittance;
    accumulatedTransmittance *= sliceTransmittance;

    return float4(accumulatedLight, accumulatedTransmittance);
}

[numthreads(8, 8, 1)]
void main(uint3 threadID : SV_DispatchThreadID)
{
    float4 accum = float4(0.0, 0.0, 0.0, 1.0);

    float3 texelCoord = float3(threadID.xy + 0.5, 0.0);
    float3 prevWorldSpacePos = calcWorldSpacePos(texelCoord);

    for (int z = 0; z < VOLUME_DEPTH; ++z)
    {
        texelCoord = float3(threadID.xy + 0.5, z + 1.0);
        float3 worldSpacePos = calcWorldSpacePos(texelCoord);
        float stepLen = distance(prevWorldSpacePos, worldSpacePos);
        prevWorldSpacePos = worldSpacePos;
        int4 pos = int4(threadID.xy, z, 0);
        float4 slice = g_InputImage.Load(pos);
        accum = scatterStep(accum.rgb, accum.a, slice.rgb, slice.a, stepLen);
        g_ResultImage[pos.xyz] = accum;
    }
}
```

Listing 3.4: Calculation of Integrated Scattering Buffer.

Finally, as demonstrated in listing 3.5, the Scattering Buffer can be easily looked up during shading and used to apply the effect to the scene.

```
float3 applyVolumetricFog(float3 sceneColor, float3 worldSpacePos,
        Texture3D scatteringTex, SamplerState linearSampler)
{
    float3 tc = computeTexCoord(worldSpacePos);
    float4 fog = scatteringTex.SampleLevel(linearSampler, tc, 0.0);
    return sceneColor * fog.a + fog.rgb;
}
```

Listing 3.5: Applying the effect to the scene.

3.2.3 Temporal Filter

The implementation described so far suffers from a major problem. Due to the very low resolution of the used 3D textures (in particular along the Z-dimension), the effect is subject to strong undersampling artifacts. This is especially noticeable when the shadow maps contain high frequency signals, which is often the case in practice. Such artifacts are visible in figure 3.3.

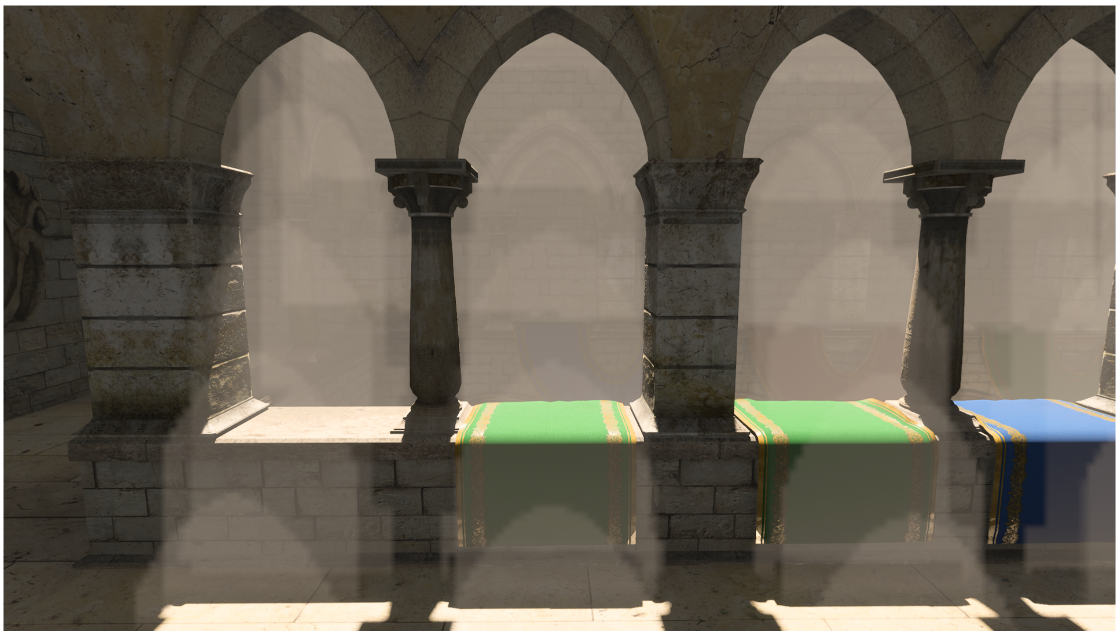


Fig. 3.3: Volumetric Fog undersampling artifacts. The individual slices of the 3D texture are clearly visible.

Wronski mitigates this problem by using prefiltered exponential shadow maps (ESM) instead of ordinary shadow maps [Wro14]. The prefiltered ESM are generated by converting regular shadow maps to ESM and then downsampling them from 1024x1024 to 256x256. Finally, a gaussian blur is applied. The resulting shadow map has much lower frequencies, resulting in reduced aliasing. The obvious drawback of this approach is the increased memory consumption and the overhead of prefiltering all required shadow maps.
Instead of filtering the shadow maps, it is also possible to use more samples to create the In-Scattering Buffer. Temporal filtering is an inexpensive way of increasing the effective sample count without actually taking more samples per frame. This is also the approach chosen by Hillaire [Hil15] and Lagarde et al. [LG18].
Temporal filtering tries to find corresponding texel(s) in the previous already filtered frame for each texel of the current frame. Both values are then used to compute an exponential moving average (EMA), giving smooth results. An EMA computes a new filtered result S_t by linearly interpolating between the previous filtered result S_{t-1} and the current sample Y_t according to some value α with $0 < \alpha < 1$. The formula for this is shown in equation 3.2.

$$S_t = \begin{cases} Y_1 & t = 1 \\ \alpha Y_t + (1.0 - \alpha) S_{t-1} & t > 1 \end{cases} \tag{3.2}$$

A small α value results in stronger filtering, making the resulting effect more robust against undersampling artifacts. If the value is too small, artifacts of the temporal filtering itself can appear. These artifacts are discussed in more detail in subsection 3.2.6. Figure 3.4 shows how the temporal filter fits into the Volumetric Fog algorithm.

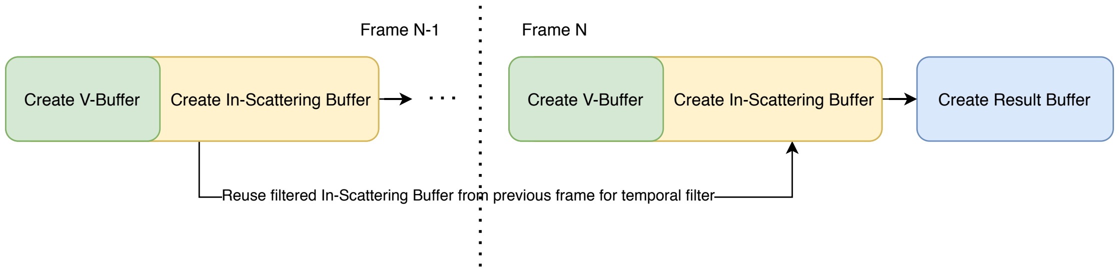

Frame N-1
Frame N
Create V-Buffer
Create In-Scattering Buffer
· · ·
Create V-Buffer
Create In-Scattering Buffer
Create Result Buffer
Reuse filtered In-Scattering Buffer from previous frame for temporal filter

Fig. 3.4: After computing in-scattered light, the filtered result from the previous frame is sampled and combined with the current result.

In order to attain different samples each frame, it is necessary to jitter the sample positions. Otherwise, for a still camera, every frame would contain the same information, making temporal filtering pointless. There are a number of different ways to jitter the sample positions. A good jittering scheme should have consecutive sample positions that are far away from each other to ensure that each frame adds new information. Furthermore, the samples should be randomly placed so that they do not suffer aliasing due to using a regular pattern. The problem of temporally filtering the In-Scattering Buffer is closely related to Temporal Anti-Aliasing (TAA), which means that the lessons learned from TAA also apply here. A popular choice for jittering samples for TAA is the Halton sequence, a low-discrepancy or *quasirandom* sequence that fulfills the requirements stated above [Kar14][Xu16]. For a given base, a Halton sequence gives an infinite number of deterministic one-dimensional values. However, jittering the TAA samples requires two-dimensional offsets and jittering the In-Scattering Buffer requires three-dimensional offsets. For TAA, it is common to use a Halton sequence of base 2 for the X-axis offset and a sequence of base 3 for the Y-axis offset. For best results, the base should be a prime number [PHJ17]. Extending this to three dimensions can be trivially done by using another sequence of base 5. C++ code for computing a single value of a sequence of a certain base is given in listing 3.6.

```cpp
float halton(size_t index, size_t base)
{
    float f = 1.0f;
    float r = 0.0f;

    while (index > 0)
    {
        f /= base;
        r += f * (index % base);
        index /= base;
    }

    return r;
}
```

Listing 3.6: C++ function to compute a value of a Halton sequence.

The Volumetric Fog implementation of this thesis precomputes 32 three-dimensional Halton values once at startup and then iterates through them each frame. An EMA α value of 0.05 gives sufficiently smooth results. Figure 3.5 shows how temporal filtering can remove undersampling artifacts.

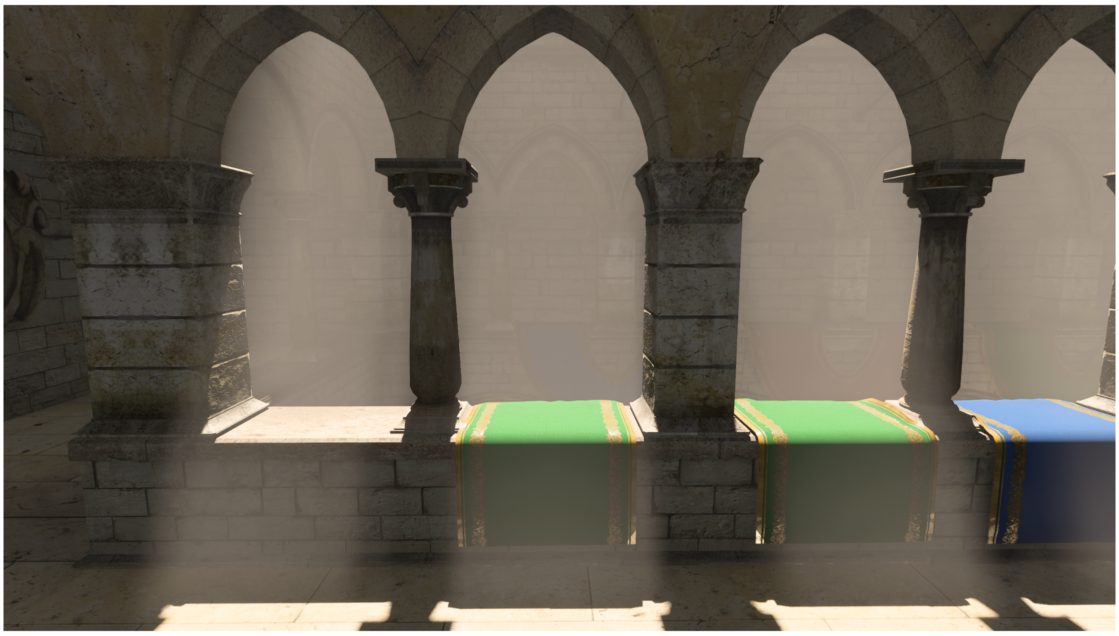


Fig. 3.5: Temporal filtering successfully removes undersampling artifacts.

3.2.4 Local Lights

The initial implementation supports only directional lights. However, in order to create convincing scenes, it should also be possible to support local lights such as spot lights and point lights. Evaluating local lights is very similar to directional lights: Instead of using the light intensity directly, it must first be attenuated according to the parameters of the local light. In particular, a distance based falloff should be applied. In the case of spot lights, the attenuation of the cone should also be considered. While area lights can also be considered as local lights, the framework used for this thesis only supports analytical punctual lights. HLSL code for evaluating a punctual light is given in listing 3.7.

```
float3 evaluatePunctualLight(uint index, float3 worldSpacePos,
                             float3 V, float phase)
{
    PunctualLight light = g_PunctualLights[index];

    const float3 L = normalize(light.position - worldSpacePos);
    const float dist = distance(light.position - worldSpacePos);
    float att = getDistanceAtt(dist, light);

    if (light.isSpotLight != 0)
    {
      att *= getAngleAtt(L, light);
    }

    return light.color * att * henyeyGreenstein(V, L, phase);
}
```

Listing 3.7: Evaluation of a punctual light.

However, the major difference between directional lights and local lights is that directional lights affect the whole scene while local lights do not. Since there could be thousands of local lights in a scene, it would be a waste of performance to iterate over all of them. Instead, an acceleration structure should be used to only evaluate lights that are likely to have an effect on the current sample point. This problem is closely related to the problem of evaluating only relevant lights during surface shading. Fortunately, this is a well researched subject with lots of viable solutions. Tiled lighting and clustered lighting are popular techniques to solve this problem [OBA12][And09][Per15]. Tiled lighting divides the screen into small tiles and computes for each tile the set of lights overlapping that tile. When shading a pixel, only the lights of the corresponding tile are evaluated. This technique works for both forward and deferred shading but is especially suitable for deferred shading when the set of overlapping lights can be limited to lights that are actually intersecting the closest surface. Otherwise, a potentially large number of lights that do not intersect the (opaque) geometry will be evaluated. The downside of this approach is that the tiled lighting acceleration structure cannot be reused for forward rendered objects such as transparent meshes.
Clustered lighting extends the concept of tiled lighting to three dimensions and determines the set of lights overlapping a certain depth range of a two-dimensional tile. The advantage of this approach is that it does not require a depth buffer to improve the culling accuracy, which also means that the resulting acceleration structure can be used for both deferred and forward shaded objects. The downside of clustered lighting compared to tiled lighting is that it requires more memory and potentially more computational power to cull lights against clusters. For this reason, the screen space size of clusters is often larger than that of the tiles used in tiled lighting. As a consequence, clusters are likely to overlap with more lights, decreasing the effectiveness of culling. Note that clustered lighting can still be preferable to tiled lighting, especially for applications that cannot rely on a depth

buffer to improve culling.
Since the Volumetric Fog algorithm needs to evaluate local lights in a three-dimensional grid fitted to the camera frustum, clustered lighting seems like an optimal solution.
Drobot presents a novel solution for clustered lighting [Dro17a]: He assigns lights to 2D screen space tiles similar to tiled lighting. This is done with hardware rasterization of light proxy geometry and atomic operations. In addition, all lights are sorted by their view space depth on the CPU. The view space Z-axis is then split into equal-sized buckets, each holding the minimum and maximum index of the sorted lights overlapping the depth range of the bucket. During shading, the lights overlapping the pixel's 2D tile are looked up. This set of lights is then further reduced by only considering lights overlapping the current depth bucket.
This hybrid tiled/clustered lighting algorithm was already present in the implementation framework of this thesis and is used for shading geometry with lights and reflection probes. As such it can be easily reused for supporting volumetric lighting for local lights: Looking up relevant lights for a sample in the In-Scattering Buffer is done by determining the corresponding screen space tile and depth bin. Figure 3.6 shows two local lights, a spot light and a point light.

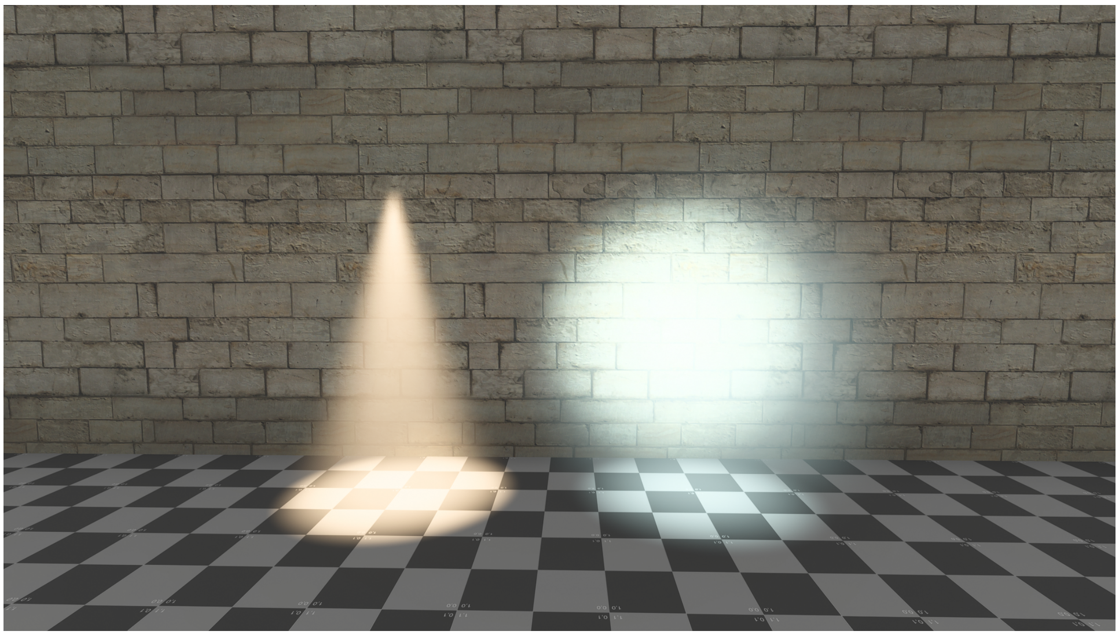


Fig. 3.6: Two local lights, a spot light and a point light, interacting with participating media.

3.2.5 Local Participating Media

In addition to unbounded, global participating media, it is desirable to also support local, bounded media. Such local media could be used to create a smoky bar

or a sauna full of steam. The participating media would then be limited to the interior of such a room. The local participating media implemented for this thesis are bounded by either a box or an ellipsoid. The bounding shape can be translated, rotated and scaled to fit a given scene. Except for the bounding volume, they expose the exact same set of parameters as global media, making the authoring process consistent.
In order to speed up local media evaluation, the same clustered lighting algorithm that is used for lights is also used for local participating media. The proxy geometry is either spherical or box-shaped, depending on the shape of the bounding volume.
Since the clustered lighting algorithm culls conservatively and because the proxy geometry is also slightly larger than the actual bounding volume, each volumetric lighting sample must be tested against the bounding volume of every local medium before the medium parameters can be evaluated. This is done by storing a transformation matrix for every local participating media that transforms from world space to bounding volume space. In this space the bounding volume is at the origin, is axis-aligned and has an extent of 1.0 in all directions. This makes it easy to test a point in world space against the volume: First the point is transformed into local space. If the bounding volume is spherical, the distance of the transformed point to origin of the coordinate system must be less than or equal to 1.0. If the bounding volume is a box, the transformed point must be in the -1.0 to 1.0 interval on all axes. As an optimization, only the first three rows of the matrix need to be stored for the transformation. An implementation of such a test is given in listing 3.8.

```
bool isInsideMedium(LocalParticipatingMedium medium, float3 worldSpacePos)
{
    // transform point from world space to local space
    float3 localPos;
    localPos.x = dot(medium.worldToLocal0, float4(worldSpacePos, 1.0));
    localPos.y = dot(medium.worldToLocal1, float4(worldSpacePos, 1.0));
    localPos.z = dot(medium.worldToLocal2, float4(worldSpacePos, 1.0));

    // box-shaped
    if (medium.spherical == 0)
    {
        return all(abs(localPos) <= 1.0);
    }
    // spherical
    else
    {
        // sqrt(dot(localPos, localPos)) is not necessary
        // because the sqrt of values <= 1.0 is also <= 1.0
        return dot(localPos, localPos) <= 1.0;
    }
}
```

Listing 3.8: Testing a world space point against a local participating medium bounding volume.

The parameters of participating media are blended additively except for the phase anisotropy, where the average of all media at the current sample point is used. The reasoning behind this is that each additional medium is considered to add particles, which linearly increases the particle density. Figure 3.7 shows two local participating media volumes lit by a point light.

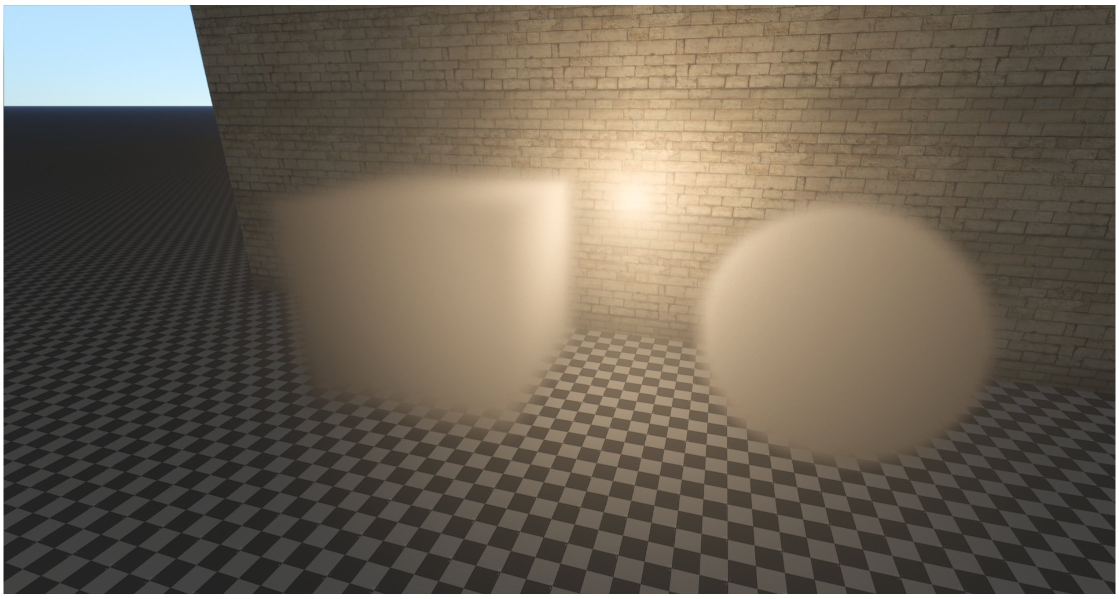


Fig. 3.7: Two local participating media volumes (box-shaped and spherical), lit by a point light.

3.2.6 Ghosting

Temporal filtering of the In-Scattering Buffer causes an artifact known as ghosting. Ghosting happens when lights move through the scene or when the camera moves through a medium with strong phase anisotropy. It manifests itself as visible trails behind the light source. An instance of ghosting is shown in figure 3.8.
This happens because samples obtained with reprojection are not tested for validity. The assumption behind temporal reprojection is that only the camera can move. This assumption is violated as soon as the lights themselves move: A sample in the current frame is not lit by a light source but the reprojected sample in the previous frame is. This causes the light to slowly fade out over several frames. In addition, it is assumed that the light scatters the same in the previous frame as in the current frame, which is only the case for an isotropic phase function or a phase anisotropy of 0.0.

A simple way of reducing ghosting is to increase the α value of the exponential moving average. Increasing this value increases the influence of the current sample and decreases the influence of previous samples, minimizing ghosting. However, past a certain value jittering becomes apparent, making this strategy impractical. Instead of temporally filtering the In-Scattering Buffer, Bauer et al. filter the V-Buffer and a 3D texture storing the sun shadow. The final result texture is then

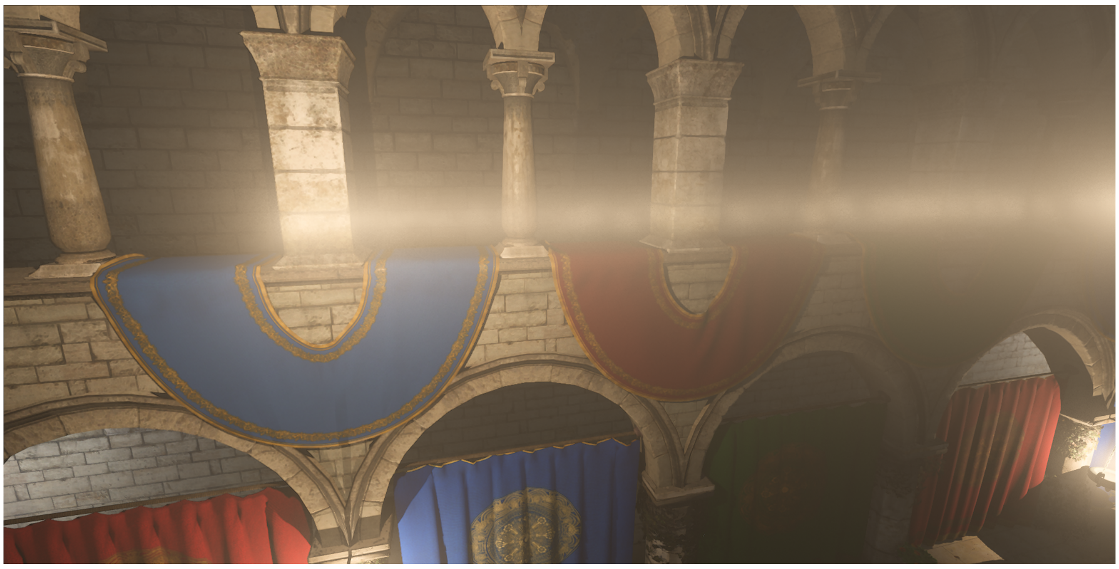


Fig. 3.8: A moving light source leaves a trail caused by ghosting.

sampled with jittered offsets. They rely on TAA to clean up the resulting noise [Bau19]. While this solution avoids ghosting by not filtering the In-Scattering Buffer, it only works properly for a single directional light. This is because every light would need its own 3D shadow texture, which is impractical for most real-time applications.

An important observation of Bauer et al. is that TAA is more capable of reducing ghosting than the temporal filter commonly employed for Volumetric Fog. Unfortunately, TAA alone is not enough to eliminate aliasing and visible jittering caused by shadow maps containing high frequency signals. In order to avoid the additional memory requirements of maintaining a shadow 3D texture, the implementation for this thesis makes a compromise: It still uses temporal filtering of the In-Scattering Buffer but also uses jittered lookups and TAA to clean up the noise. Since TAA acts as an additional temporal filter, it is possible to increase the EMA α value of the temporal filter used on the In-Scattering Buffer. This effectively defers some of the temporal filtering to TAA. As noted above, increasing α directly decreases ghosting. With this hybrid approach, it is possible to safely use α values as high as 0.2 without causing visible artifacts in most scenes. While this does not solve the problem completely, it is a big improvement. Another advantage of this solution is that it incurs almost no performance degradation: The only additional overhead lies in having to sample a noise texture to offset the texture coordinate used to look up the final result texture. One potential downside is that this solution requires TAA. However, many modern real-time renderers already use TAA. Additionally, other effects, such as screen space reflections and screen space ambient occlusion can also profit from TAA in the same way.

Temporal Anti-Aliasing

Since it is central to the solution described above, the TAA implementation of this thesis is briefly detailed in the following. It is based on the work of Karis [Kar14] and Salvi [Sal16]. It uses 16 Halton samples to jitter the projection matrix. Similar to the solution presented by Karis, it uses a simple reversible tone mapping operator to tone map all involved color values (see equations 3.3 and 3.4) [Kar14].

$$T(color) = \frac{color}{1 + luma(color)} \tag{3.3}$$

$$T^{-1}(color) = \frac{color}{1 - luma(color)} \tag{3.4}$$

However, instead of clipping against the color AABB, it uses variance clipping (VC), introduced by Salvi. VC constructs an improved AABB from the first two moments of the local color sample distribution [Sal16]. The AABB is centered on the mean value and has an extent determined by the standard deviation. The extent can additionally be scaled by a scaling factor γ. A higher γ value increases temporal stability but may introduce ghosting. Experiments showed that variance clipping is more successful at reducing noise introduced by using jittered lookups for the Integrated Scattering Buffer than neighborhood clipping. Listing 3.9 demonstrates how VC can be used to construct the AABB.
In order to reduce numerical diffusion caused by linearly sampling the filtered history buffer, a bicubic filter is used. Usually such a filter requires 9 texture samples. However, Jimenez shows that the 4 corner samples have little influence on the final result and can be ignored, bringing down the cost to 5 texture samples [Jim16].
At the end of the TAA shader, the inverse tone mapping operator (equation 3.4) is applied to filtered result, giving a linear HDR value.

```
// currentColor and historyColor are tone mapped with simpleTonemap()
float3 varianceClipping(float2 texCoord, float2 texelSize,
                        float3 currentColor, float3 historyColor)
{
    float3 neighborhoodMin = currentColor;
    float3 neighborhoodMax = currentColor;

    // determine first and second moments of neighborhood
    float3 m1 = 0.0;
    float3 m2 = 0.0;
    for (int y = 0; y < 3; ++y)
    {
      for (int x = 0; x < 3; ++x)
      {
            float2 tc = texCoord + (float2(x, y) - 1.0) * texelSize;
            float3 tap;
            tap = g_InputImage.SampleLevel(g_LinearSampler, tc, 0.0).rgb;
            tap = simpleTonemap(tap);

            m1 += tap;
            m2 += tap * tap;
      }
    }

    float3 mean = m1 / 9.0;
    float3 stddev = sqrt(max((m2 / 9.0 - mean * mean), 1e-7));

    const float wideningFactor = 1.0; // gamma

    neighborhoodMin = -stddev * wideningFactor + mean;
    neighborhoodMax = stddev * wideningFactor + mean;

    // clip history against current frame neighborhood
    return clipAABB(historyColor, neighborhoodMin, neighborhoodMax);
}
```

Listing 3.9: Using variance clipping and a reversible tone mapping operator to clip the history color against the current frame neighbborhood.

3.2.7 Leaking

Light leaking is an artifact commonly encountered in certain global illumination (GI) techniques. It manifests itself as light coming through solid walls and is often caused by the GI technique not being aware of walls. Volumetric Fog suffers from a conceptually similar artifact: Fog can leak through thin walls. This happens because the effect is looked up from a 3D texture using linear interpolation. It is possible that a lookup interpolates between a texel belonging to the inside of a dark room and a brightly lit texel outside of the room. Since the algorithm is not aware of the wall separating those two texels, the bright value is erroneously used in the interpolation, giving the appearance of bright fog leaking through the wall. Since the slices in the 3D texture are distributed in such a way that they cover larger distances as they get further from the camera, leaking also gets worse the further away a wall is from the camera. Figure 3.9 shows an instance of bright fog leaking through thin geometry.
A simple solution for this problem is to apply a bias to the texture coordinate used to look up the volume: The coordinate is moved 1.5 texels closer towards the camera. In addition, the lookup jitter extent along the Z-axis is halved. These

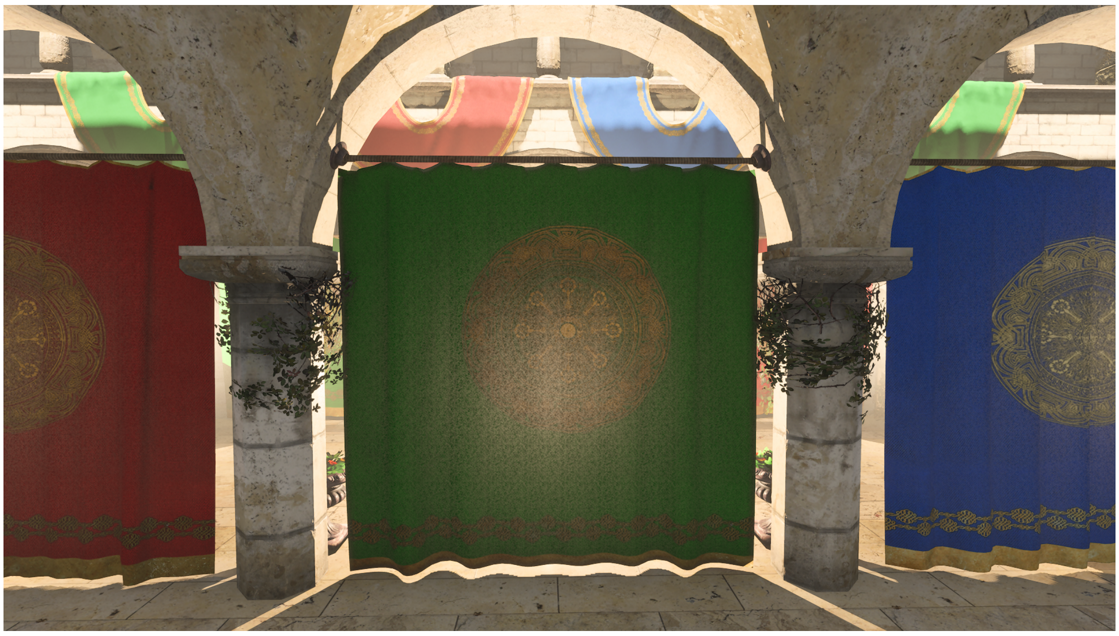


Fig. 3.9: Bright fog is leaking through the curtain.

measures solve most instances of leaking at the cost of introducing some error. For the purpose of this implementation, this trade-off was deemed acceptable. The result of applying this simple fix is shown in figure 3.10. Leaking can still happen when the camera is moving forward. Forward motion causes the reprojection in the temporal filter to sample data from slices that were further away from the camera in the previous frame. As distant slices cover a larger depth range, this is another source of leaking.

3.2.8 Long Range Volumetric Lighting

The range of the Volumetric Fog effect is commonly limited to a maximum distance to the camera of approximately 64 meters. This limitation is necessary because covering larger distances requires more slices in all involved 3D textures, increasing both memory consumption and computation time. However, the consequence of this is that objects further away than the maximum range of the effect do not have proper in-scattering and transmittance data available. Depending on the scene, this may or may not pose a problem: In small scenes it might not even be noticeable. However, for larger scenes such as the one shown in figure 3.11, a solution for long range volumetric lighting is necessary.
This solution would need to start where the range of the Volumetric Fog ends and cover the whole distance up to the point where the view ray intersects the scene (or the camera far plane, if it does not intersect the scene).
In order to meet the performance goal, a few compromises need to be made. Similar in spirit to Bauer et al. [Bau19], long range volumetric lighting ignores local lights and local participating media. While it would be possible to support these features (the hybrid tiled lighting algorithm supports a range of up to 8 km), supporting

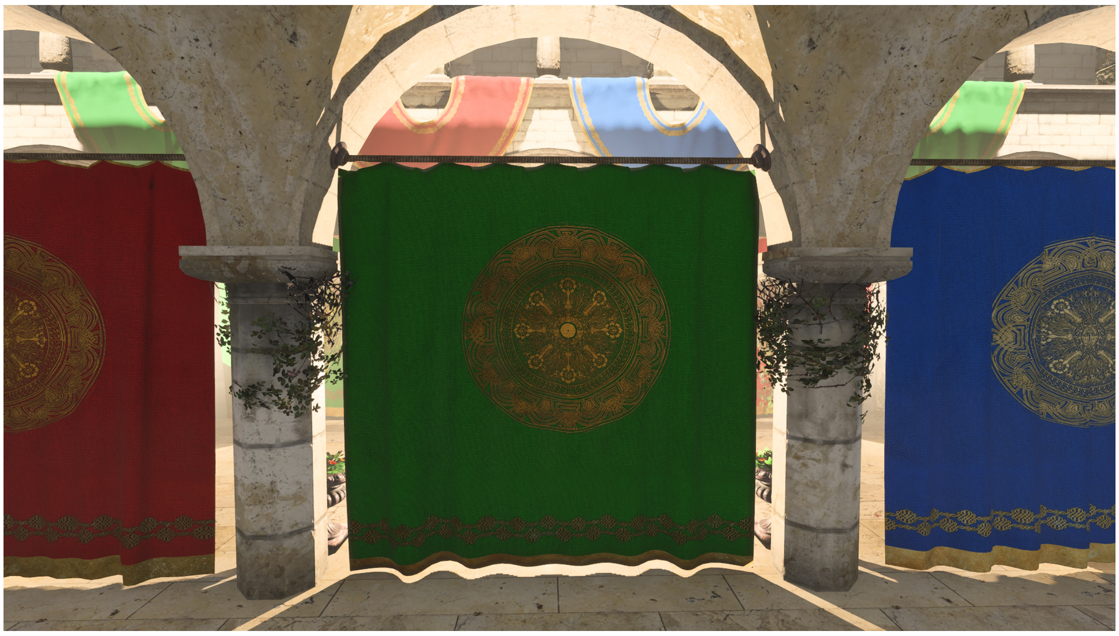


Fig. 3.10: Applying a bias to the lookup coordinate fixes most leaking.

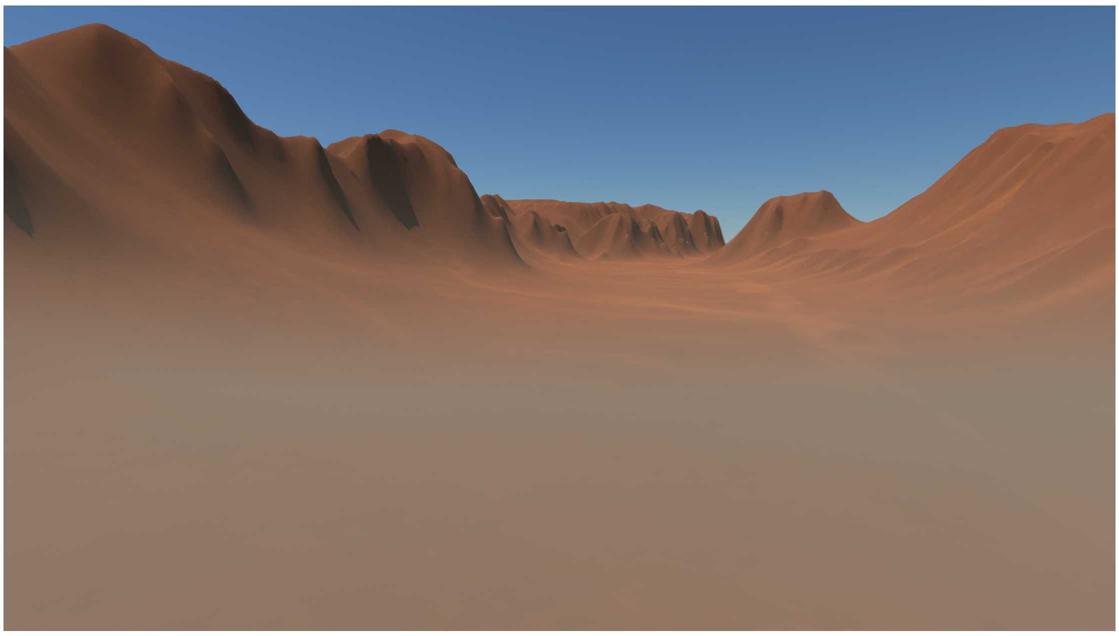


Fig. 3.11: The finite range of Volumetric Fog is clearly visible. Scenes with long view distances might need a fallback for Volumetric Fog.

them increases memory bandwidth requirements and register pressure. Given the observation that the distances that need to be covered by this fallback solution can vary greatly and that 3D textures are unsuitable for covering very large ranges, computing a 2D texture with ray marching seems like a suitable approach. Unfortunately, since the 2D texture only stores results for a single depth, long range volumetric lighting is only available for opaque objects. In summary, the fallback solution only supports directional lights and global participating media and is restricted to opaque objects. These limitations are similar to the work presented by Bauer et al. [Bau19] and Cho et al. [CGK19].

Since the effect is computed in half resolution and needs a depth buffer sample per result texel, the full resolution depth buffer needs to be downsampled. Using linear filtering as is commonly done when downsampling color textures is incorrect because the depth values are non-linear and the resulting values would correspond to none of the original full resolution values. A better solution is to pick one of the 4 full resolution texels corresponding to a single half resolution texel. There are several strategies on which value to pick. One could always pick the top left value, pick a different value every frame or pick the minimum or maximum value. Another option, which is the one used in this implementation is to alternate between minimum and maximum downsampling in a checkerboard fashion. Simple HLSL code for this is given in listing 3.10. While each of the listed downsampling methods has its advantages and disadvantages, min/max checkerboard downsampling seems to produce the most pleasing results for the purpose of long range volumetric fog.

```hlsl
float depthCheckerboardDownsample(float2 texCoord, uint2 threadID)
{
    float4 depths = g_DepthImage.GatherRed(g_PointSampler, texCoord);

    // apply min/max filter in checkerboard
    float result = (((threadID.x + threadID.y) & 1) == 0)
        ? min(min(depths.x, depths.y), min(depths.z, depths.w))
        : max(max(depths.x, depths.y), max(depths.z, depths.w));

    return result;
}
```

Listing 3.10: Checkerboard downsampling the depth buffer.

Long range volumetric lighting is then implemented with a compute shader computing a 2D texture. Each thread uses 16 steps to ray march a given view ray. As stated above, the ray starts where the range of the Volumetric Fog effect ends and ends at the depth given by the depth buffer. Ray marching can be skipped if the depth buffer sample is within the range covered by the Volumetric Fog effect.
Each ray marching step evaluates the global participating media at the current sample position and then accumulates all in-scattered directional lights, as well as an ambient term. At the end of each iteration the in-scattered lighting and extinction of the current sample is accumulated with the in-scattering and transmittance of the view ray. This step is essentially the same as the *scatterStep()* function used

in the integration shader of the Volumetric Fog effect (see listing 3.4). After all samples have been processed, the accumulated values represent the in-scattering and transmittance along the whole ray segment. This result is then stored to a RGBA16F texture and later used when volumetric lighting is applied to the scene. In general, 16 samples may be too few to accurately sample the volumetric lighting. However, increasing the number of samples drastically increases the computation time. A common solution is to jitter the samples along the ray and apply a bilateral blur to the resulting texture. In this context, this is known as interleaved sampling [TU09][SWR13][Gla14][Val14a]. Instead of a bilateral blur, Bauer et al. note that is also possible to jitter the texture coordinates used to sample the texture and use TAA to resolve the resulting noise [Bau19]. The same approach is used in the implementation for this thesis. Figure 3.12 shows the result of using this fallback solution to fix the range limitation of Volumetric Fog.

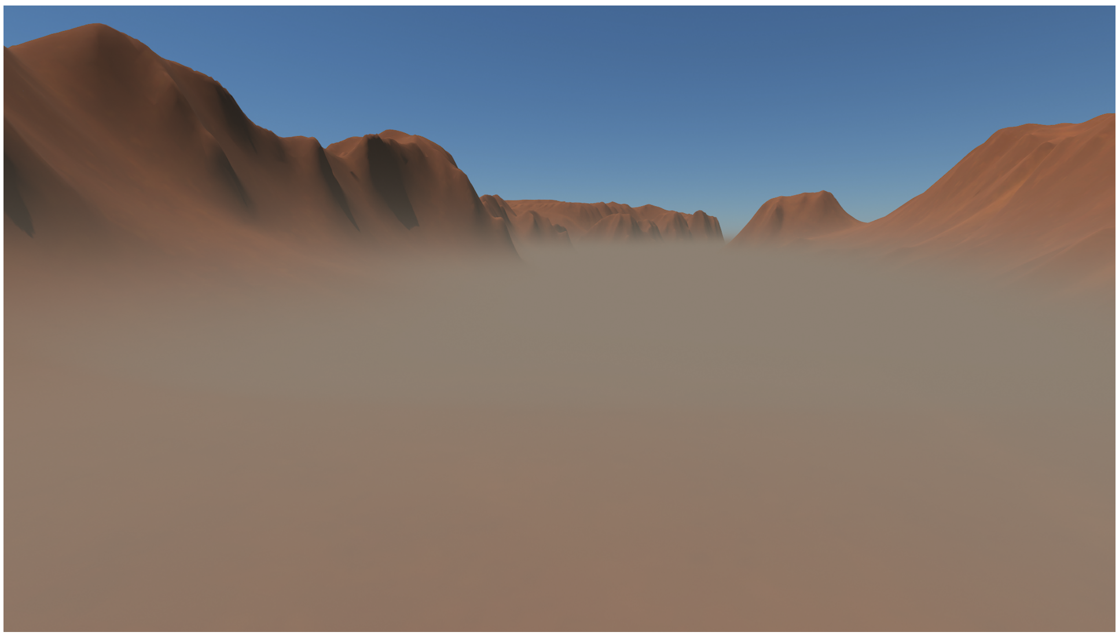


Fig. 3.12: Ray marched long range volumetric lighting takes over past the maximum range of Volumetric Fog, fixing the artifact in figure 3.11.

3.2.9 Checkerboard Rendering

Analyzing the performance of the different steps of Volumetric Fog reveals that the computation of the In-Scattering Buffer is usually the most expensive step. Since the number of visible lights and participating media is variable, its performance is also very scene dependent. These characteristics make it a good candidate for optimization.

Checkerboard rendering is a technique to reduce the cost of shading in deferred or forward rendering [El 16][Wih17]. It works by only shading every other pixel in a checkerboard fashion. Even frames shade “white” pixels and odd frames shade “black” pixels. The missing pixels are then filled in with temporal reprojection, using the (reconstructed) full resolution image of the previous frame. While the reconstruction step incurs some additional overhead, shading only half of all pixels is a performance win.

Using checkerboard rendering for Volumetric Fog is a novel approach and one of the contributions of this thesis. The idea is to only compute every other texel of the In-Scattering Buffer. In contrast to regular checkerboard rendering, the checkerboard pattern of every other slice is flipped. Essentially, the pattern has been expanded to the third dimension. This halves the required computations and the required bandwidth of sampling shadow maps and loading light data. The missing texels are then filled in during the temporal filter stage: If a texel was not computed in the current frame, it is filled in by reprojecting into the previous frame. If reprojection fails, its value is obtained by linearly interpolating the 4 neighbor texels in the same slice (which were all computed in the current frame).
However, one downside of this technique is that temporal filtering can no longer be done in the same shader which computes the In-Scattering Buffer. The reason for this is that the neighboring texels of a missing texel may be needed. This necessitates writing out the results of the current frame to memory. Fortunately, since only half of all texels are computed, only half the memory and bandwidth is needed to write out the results. Figure 3.13 shows the Volumetric Fog passes when checkerboard rendering is added.

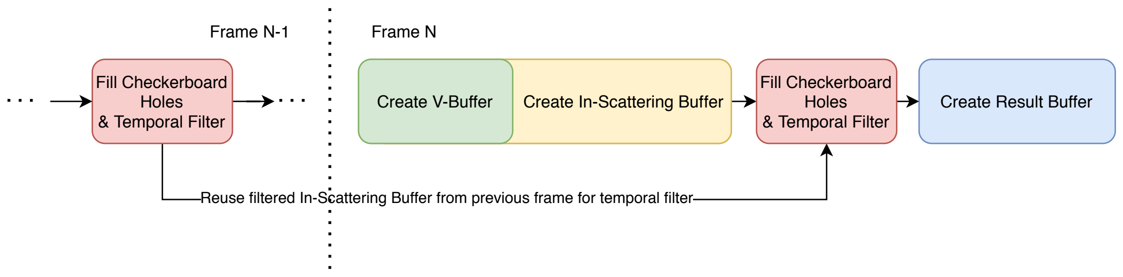

Frame N-1
Frame N
Fill Checkerboard Holes & Temporal Filter
Create V-Buffer
Create In-Scattering Buffer
Fill Checkerboard Holes & Temporal Filter
Create Result Buffer
Reuse filtered In-Scattering Buffer from previous frame for temporal filter

Fig. 3.13: Implementing checkerboard rendering requires the temporal filter to be moved into its own pass.

To summarize, the In-Scattering pass is dispatched for only half of all texels and no longer does temporal filtering. Instead, the unfiltered results are written out to an intermediate texture and sampled in a subsequent pass, which performs temporal filtering and fills in the missing texels.
Listing 3.11 shows how to compute the corresponding texel coordinate of a thread in the In-Scattering pass.

```
// jitter is computed with a 3D Halton sequence
// cbCondition flips between 1 and 0 every frame
float3 getTexelCoord(uint3 threadID, float3 jitter, bool cbCondition)
{
    float3 texelCoord = threadID;
    texelCoord.z *= 2.0;
    bool isOffset = (((threadID.x + threadID.y) & 1) == cbCondition);
    texelCoord.z += isOffset ? 1.0 : 0.0;
    texelCoord += jitter;
    return texelCoord;
}
```

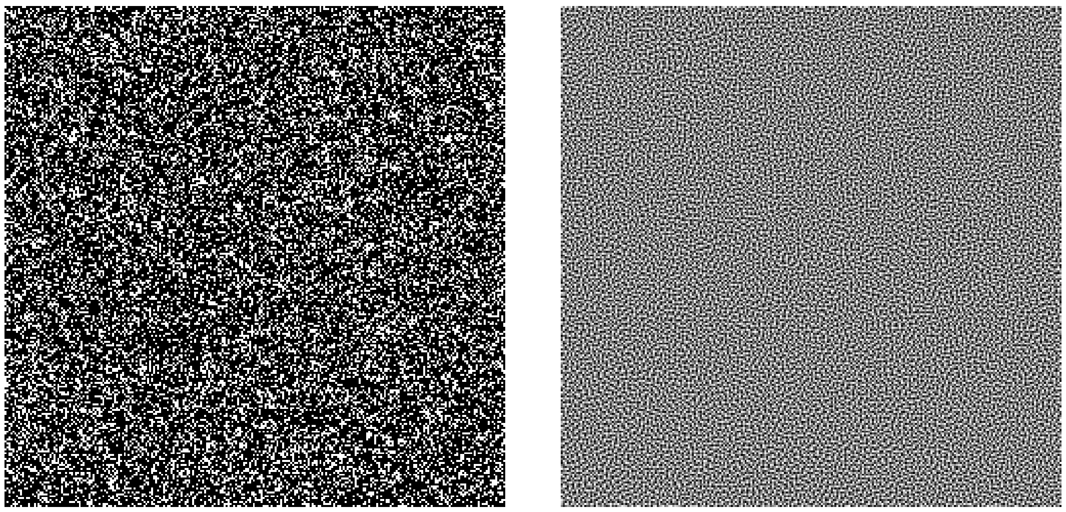
Listing 3.11: Computing the texel coordinate with checkerboard rendering.

3.2.10 Rendering

Volumetric Fog and the fallback solution for long range volumetric lighting are applied to the scene in a full-screen pass after all shading of opaque objects has been done. While this could be combined with other passes, it was moved into its own pass to profile the performance impact more easily.

Blue Noise

As discussed in subsection 3.2.6 and 3.2.8, both volumetric lighting result textures are looked up with jittered offsets. In particular, blue noise is used to obtain the offsets. Figure 3.14 shows what a blue noise texture looks like compared to ordinary white noise.



(a) White Noise (b) Blue Noise

Fig. 3.14: Blue noise is more pleasant to the human eye than white noise. Source: Peters [Pet16].

It is sampled from a 64x64 array texture with 64 layers, where each consecutive frame samples from a different layer. The noise texture is sampled in such a way

that the noise tiles over the whole screen. Blue noise is suitable for this application because it distributes samples uniformly and has weak low-frequency components, making it look more pleasant to the human eye than other types of noise.
The noise is scaled into the -1.5 to 1.5 range and specifies offsets in texel units. Listing 3.12 shows how the noise is sampled from the blue noise array texture.

```
float4 getNoise(uint2 texelCoord, uint frame)
{
    // wrap around every 64 frames
    uint layer = frame & 63;
    // manual wrapping (the image is 64x64)
    uint2 coord = texelCoord & 63;
    float4 noise = g_BlueNoiseImage.Load(uint4(coord, layer, 0));
    // transform from 0.0 .. 1.0 range to -1.5 .. 1.5 range
    noise = (noise * 2.0 - 1.0) * 1.5;
}
```

Listing 3.12: Sampling the blue noise array texture.

Long Range Volumetric Lighting

Since long range volumetric lighting is computed at half resolution, it needs to be upsampled with a bilateral filter to full resolution. Instead of using the closest four samples to each full resolution pixel, blue noise offsets are used to bilinearly sample the texture 4 times. Each sample is weighted by its associated downsampled depth with respect to the depth of the current full resolution pixel. Samples with a depth difference of up to 10 % are accepted. Samples that have a larger difference still get a minimal weight to ensure that not all samples are rejected. In order to reduce the number of texture accesses and to not introduce unnecessary memory dependencies, a four component blue noise value is sampled once and then reused for all other texture samples by swizzling the components every iteration. This upsampling scheme is demonstrated in listing 3.13.

```
// texCoord is the UV coordinate of the fullscreen quad/triangle
float4 getLongRangeVolumetrics(float2 texCoord, float depth, float4 noise)
{
    float2 texelSize;
    g_RaymarchedVolumetricsImage.GetDimensions(texelSize.x, texelSize.y);
    texelSize = rcp(texelSize);

    float4 result = 0.0;
    float totalWeight = 0.0;

    for (int i = 0; i < 4; ++i)
    {
      float2 tc = texCoord + noise.xy * texelSize;

      // get linear depth of sample
      float sampleDepth = g_RaymarchedVolumetricDepthImage.SampleLevel(
          g_Samplers[SAMPLER_POINT_CLAMP], tc,      0.0).x;
      sampleDepth = linearDepth(sampleDepth);

      // samples within 10% of the center depth are acceptable.
      float weight = 1.0 - (abs(depth - sampleDepth) / (depth * 0.1));
      // ensure a minimum weight so that we get a totalWeight > 0
      weight = max(1e-5, saturate(weight));

      result += g_RaymarchedVolumetricsImage.SampleLevel(
          g_Samplers[SAMPLER_LINEAR_CLAMP], tc, 0.0) * weight;
      totalWeight += weight;

      // swizzle noise to get a different offset in the next iteration
      noise = noise.yzwx;
    }

    return result * rcp(totalWeight);
}
```

Listing 3.13: Upsampling long range volumetric lighting with jittered offsets.

Volumetric Fog

Volumetric Fog is applied to the opaque scene in the same pass. Unlike long range volumetric lighting, it does not employ depth based weights, making it less involved. In order to improve the visual quality when sampling from it, the Integrated Scattering Buffer holds in-scattering values that are tone mapped with the simple tone mapping operator used for TAA (see equation 3.3). This tone mapping operation needs to be reversed at the end of the shader. Code for sampling the Volumetric Fog texture is given in listing 3.14.

```
float4 getVolumetricFog(float3 volumetricFogTexCoord)
{
    float3 imageDims;
    g_VolumetricFogImage.GetDimensions(imageDims.x, imageDims.y, imageDims.z);
    float3 texelSize = rcp(imageDims);

    float4 result = 0.0;

    for (int i = 0; i < 4; ++i)
    {
        float3 tc = volumetricFogTexCoord + noise.xyz * texelSize;
        fog += g_VolumetricFogImage.SampleLevel(
            g_Samplers[SAMPLER_LINEAR_CLAMP], tc, 0.0) / 4.0;
        noise = noise.yzwx;
    }

    result.rgb = inverseSimpleTonemap(result.rgb);
    return result;
}
```

Listing 3.14: Sampling the Volumetric Fog texture.

Combining the volumetric lighting contributions of both techniques is trivial. Transmittance can be combined by simple multiplication. Since long range volumetric lighting is only computed for distances past the limit of Volumetric Fog, its in-scattering needs to be attenuated by the transmittance sampled from the Volumetric Fog texture. Afterwards, both in-scattering terms can be added. Listing 3.15 demonstrates how to combine both terms and apply them to the scene.

```
float4 applyVolumetricLighting(float3 sceneColor, float4 volumetricFog,
        float4 longRangeFog)
{
    float4 volumetrics;
    volumetrics.rgb = (longRangeFog.rgb * volumetricFog.a)
                        + volumetricFog.rgb;
    volumetrics.a = longRangeFog.a * volumetricFog.a;

    return sceneColor * volumetrics.a + volumetrics.rgb;
}
```

Listing 3.15: Combining both volumetric lighting terms and applying them to the scene.

Transparent objects are rendered at a later point and sample the Volumetric Fog texture directly in their forward pixel shader. Since it only has in-scattering and transmittance values for a single depth, long range volumetric lighting is omitted for transparent objects.

3.3 Volumetric Shadows

Techniques for volumetric shadows are commonly split into two categories: volumetric shadows cast by particles and volumetric shadows cast by volumes. One of the goals of the implementation for this thesis is to create a unified solution for both particles and volumes. As such, a solution for volumetric shadows must support both kinds of participating media representations. Figure 3.15 shows a scene with billboard particles, global and local participating media without volumetric shadows, demonstrating the importance of a solution for unified volumetric shadows. Since particles may not necessarily be sorted, this requirement makes techniques that depend on sampling the media in a sorted order less suitable. For the purpose of this thesis, two techniques have been considered: Opacity Shadow Maps generated by ray marching through a voxelized extinction volume and Fourier Opacity Mapping.

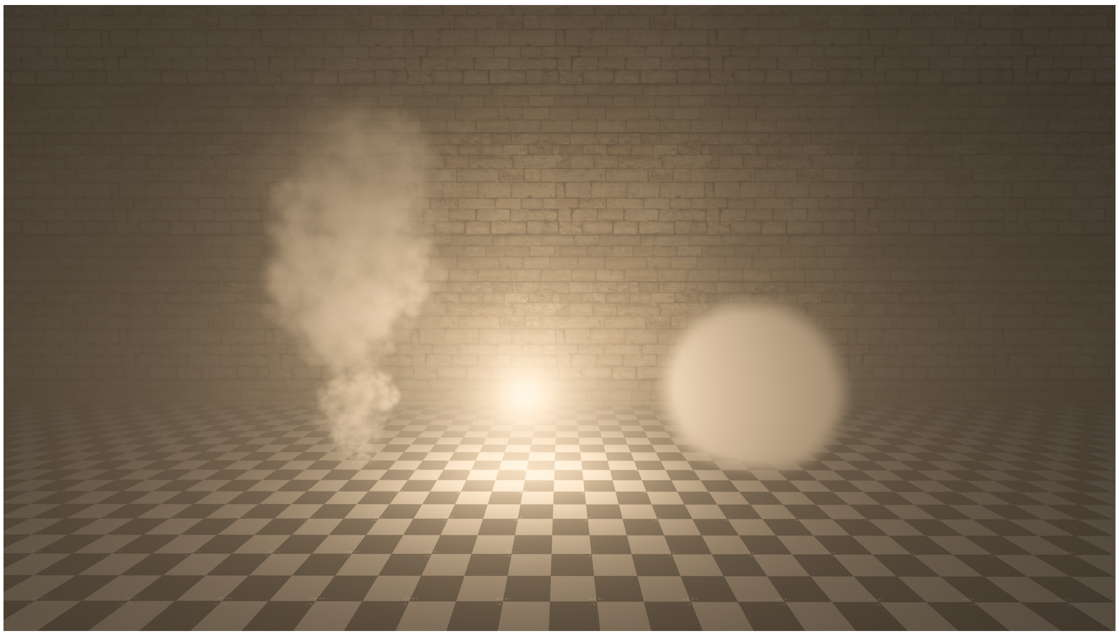


Fig. 3.15: Billboard particles, a global participating medium and a local volume lit by a point light without volumetric shadows.

3.3.1 Ray Marching Voxels

Hillaire et al. solve the problem of a unified volumetric shadow solution with *Opacity Shadow Maps* [Hil15]. They voxelize the extinction coefficient of their participating media volumes into a set of cascading 3D textures centered around the camera. The extinction coefficient of particles is estimated based on their opacity and an artist-configurable parameter. Since most graphics hardware does not support atomic operations for data types other than unsigned integers, adding the extinction coefficient of particles into the extinction cascades requires some additional work.
The extinction value is mapped to the 0 to 2048 range of a 32 bit unsigned integer and then atomically added to a separate set of 3D textures of unsigned integer format. After these secondary texture have been filled with the contributions of all particles, they are combined with the actual extinction cascades in a separate pass.
In a next step, OSM are generated for every local light with enabled volumetric

shadows. Each such light is assigned a 32x32x32 volume out of a larger texture atlas. OSM are generated by marching through the voxelized extinction and storing transmittance in the OSM texels. For spot lights this is done in a similar way as when marching through the In-Scattering Buffer when computing the Integrated Scattering Buffer for Volumetric Fog. This means that only 32x32 threads are necessary. For point lights, they also use a 32x32x32 texture, but dispatch a thread for every texel and do not use intermediate results to fill other texels, as is done for spot lights. Generating OSM for point lights is therefore a bit more expensive.

While this technique supports both participating media volumes and particles and also exhibits good performance characteristics, it has a number of downsides.
Using a set of cascading 3D textures to voxelize extinction implies discretization: The voxel grid has (in most cases) a lower resolution than the input signal, leading to aliasing. Moving participating media can exhibit popping in the volumetric shadows when they are suddenly voxelized into a voxel they did not cover in the previous frame. This problem can be mitigated by using higher resolution cascades. However, doubling the resolution increases memory requirements by a factor of 8, making this not very scalable.
Furthermore, voxelizing large particles can get expensive. Hillaire et al. use point and trilinear voxelization. Point voxelization writes the particle extinction coefficient to the voxel that is closest to the center of the particle. Trilinear voxelization adds partial contributions to the 8 closest voxels. If a large particle is to be properly voxelized, its extinction would have to be added to all voxels covered by the particle, making this approach costly. This gets even more expensive when the resolution of the voxel grid increases. Another downside of the cascades is that they limit the effect to a certain radius around the camera. Depending on the application, this might be an acceptable limitation.
Despite the disadvantages, voxelizing extinction is an interesting solution as it enables other volumetric shadow algorithms that rely on taking samples in a sorted order to also support particles.
Due to the low resolution of 32x32x32, the OSM cannot encode very detailed shadow information. While this may be acceptable for the XY-plane, depending on the radius of the light source, 32 depth slices might not be enough. This can manifest itself as erroneous self-shadowing. Similar to the cascaded extinction volumes, memory consumption increases at an unfavorable rate with increased resolution.
Another limitation of this technique is that it only supports local lights, but not directional lights. For directional lights, a much larger OSM would be needed, especially along the Z-dimension.
On account of these limitations, in particular the low resolution of the OSM and the discretization caused by voxelization, this approach was not further pursued.

3.3.2 Fourier Opacity Mapping

Fourier Opacity Mapping (FOM), introduced by Jansen et al. [JB10] is a technique primarily intended for rendering volumetric shadows cast by billboard particles.

They show how their algorithm can be used to reformulate the extinction function $\sigma_t(z)$ (z is the distance along the ray) as a fourier series. The coefficients of the fourier series can then be used to reconstruct the value of the transmittance function for a given distance to the light source. With an infinite number of coefficients, the original signal can be reconstructed precisely. However, since this is not feasible, the number of coefficients needs to be limited. Fortunately, the transmittance function of participating media is often very smooth, allowing a small number of coefficients to still give a satisfying approximation. Jansen et al. note that 8 coefficients are a good choice (see [JB10] for a quality comparison of different coefficient counts).
The coefficients are actually pairs of coefficients. Each pair k is computed by iterating over all (billboard particle) samples i and accumulating their opacity α_i and depth d_i values as shown in equations 3.5 and 3.6.

$$a_k = -2\sum_i ln(1-\alpha_i)cos(2\pi k d_i) \tag{3.5}$$

$$b_k = -2\sum_i ln(1-\alpha_i)sin(2\pi k d_i) \tag{3.6}$$

In practice, these coefficients are stored in fourier opacity maps, a set of textures with four coefficients each. These textures are used in a similar way as regular shadow maps: In a first step, the textures are cleared to zero. Afterwards, billboard particles are rendered into them as if rendering to a shadow map. Finally, the textures can be looked up during scene rendering to reconstruct the transmittance at a given depth. An example of such a set of textures is shown in figure 3.16.

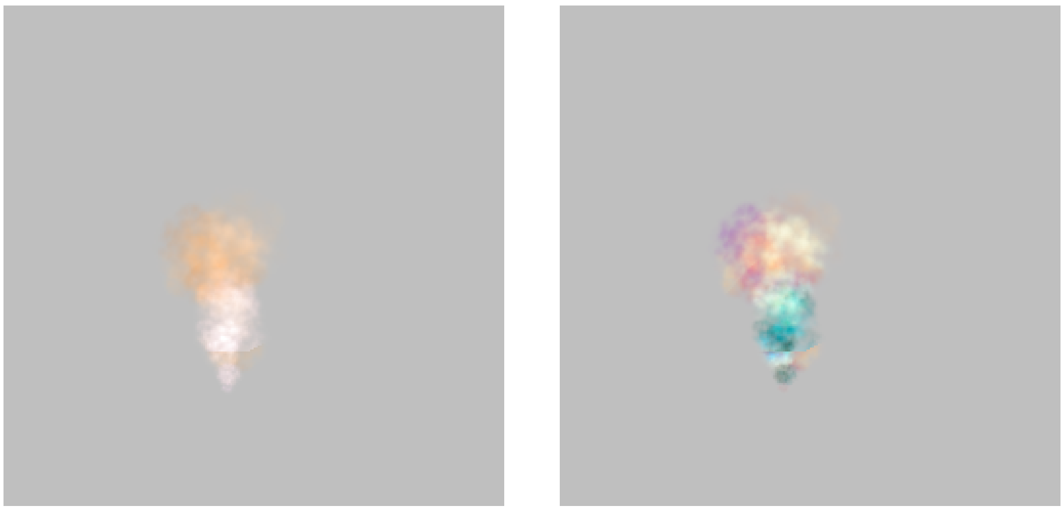

(a) The first set of four coefficients. (b) The second set of four coefficients.

Fig. 3.16: Two slices of a texture with eight FOM coefficients stored in the color channels.

Using the multiple render targets feature of modern graphics hardware, all fourier opacity maps can be rendered to at once. In the pixel shader, equations 3.7 and 3.8 are evaluated to compute the contribution to each coefficient pair. The results are then written to the texture with additive blending.

$$\delta a_{i,k} = -2ln(1-\alpha_i)cos(2\pi k d_i) \tag{3.7}$$

$$\delta b_{i,k} = -2ln(1-\alpha_i)sin(2\pi k d_i) \tag{3.8}$$

Since equations 3.5 and 3.6 do not rely on a particular order of the samples i, particles can be rendered in any order to the fourier opacity maps. A translation of equations 3.7 and 3.8 to HLSL is given in listing 3.16. It assumes that *depth* is a linear value between 0.0 and 1.0 specifying the distance of the current sample from the light source. Transmittance is equivalent to $1-\alpha_i$. As proposed by Jansen et al., the recurrence relations shown in equations 3.9 and 3.10 are used to only calculate sin() and cos() once. Note that the b coefficient of the first set is always zero, so in practice, one channel of the fourier opacity map is unused.

$$sin((n+1)\theta) = sin(n\theta)cos(\theta) + cos(n\theta)sin(\theta) \tag{3.9}$$

$$cos((n+1)\theta) = cos(n\theta)cos(\theta) - sin(n\theta)sin(\theta) \tag{3.10}$$

```
void fourierOpacityAccumulate(float depth, float transmittance,
    inout float4 result0, inout float4 result1)
{
    transmittance = max(transmittance, 1e-5);
    const float depthTwoPi = depth * 2.0 * PI;

    const float lnTransmittance = -2.0 * log(transmittance);

    // a = -2 * log(transmittance) * cos(2 * PI * k * depth)
    // b = -2 * log(transmittance) * sin(2 * PI * k * depth)

    // cos(0.0) == 1.0
    result0.r += lnTransmittance;// * cos(depthTwoPi * 0.0);

    // sin(0.0) == 0.0
    // result0.g += lnTransmittance * sin(depthTwoPi * 0.0);

    float sine, cosine;
    sincos(depthTwoPi, sine, cosine);
    result0.b += lnTransmittance * cosine;
    result0.a += lnTransmittance * sine;

    const float sine2 = sine * cosine + cosine * sine;
    const float cosine2 = cosine * cosine - sine * sine;
    result1.r += lnTransmittance * cosine2;
    result1.g += lnTransmittance * sine2;

    const float sine3 = sine2 * cosine + cosine2 * sine;
    const float cosine3 = cosine2 * cosine - sine2 * sine;
    result1.b += lnTransmittance * cosine3;
    result1.a += lnTransmittance * sine3;
}
```

Listing 3.16: Accumulating a depth-transmittance sample with the FOM coefficients.

During scene rendering, the fourier opacity maps can be looked up to calculate the volumetric shadow term. Equation 3.11 shows how the coefficients are used to compute an approximation to the integral of the extinction function along the ray starting at the light source and ending at distance d.

$$\int_0^d \sigma(z)_t dz \approx \frac{a_0}{2}d + \sum_{k=1}^{n} \frac{a_k}{2\pi k} sin(2\pi kd) + \sum_{k=1}^{n} \frac{b_k}{2\pi k}(1 - cos(2\pi kd)) \tag{3.11}$$

Beer's law then gives the transmittance at distance d (see equation 3.12), which can be used as the volumetric shadow term.

$$T(d) = exp(-\int_0^d \sigma(z)_t dz) \tag{3.12}$$

Listing 3.17 demonstrates how to use equations 3.11 and 3.12 to compute the transmittance for a set of coefficients and a given depth. Note that the recurrence relations from equations 3.9 and 3.10 are used here as well. An important detail of this code is that *saturate()* is called on the result. As the transmittance function is always in the range of 0.0 to 1.0, this call is necessary to avoid values outside this range. Such values can arise as a consequence of ringing, an inherent artifact of FOM. Ringing manifests itself as fluctuations in the reconstructed signal and happens when insufficient coefficients are used to encode a signal with high frequencies.

```
float fourierOpacityGetTransmittance(float depth, float4 fom0, float4 fom1)
{
    float lnTransmittance = fom0.r * 0.5 * depth;

    float sine, cosine;
    sincos(depth * 2.0 * PI, sine, cosine);
    lnTransmittance += fom0.b / (2.0 * PI * 1.0) * sine;
    lnTransmittance += fom0.a / (2.0 * PI * 1.0) * (1.0 - cosine);

    const float sine2 = sine * cosine + cosine * sine;
    const float cosine2 = cosine * cosine - sine * sine;
    lnTransmittance += fom1.r / (2.0 * PI * 2.0) * sine2;
    lnTransmittance += fom1.g / (2.0 * PI * 2.0) * (1.0 - cosine2);

    const float sine3 = sine2 * cosine + cosine2 * sine;
    const float cosine3 = cosine2 * cosine - sine2 * sine;
    lnTransmittance += fom1.b / (2.0 * PI * 3.0) * sine3;
    lnTransmittance += fom1.a / (2.0 * PI * 3.0) * (1.0 - cosine3);

    return saturate(exp(-lnTransmittance);
}
```

Listing 3.17: Retrieving transmittance from FOM coefficients.

FOM has some useful properties that make it suitable for the purpose of implementing a unified solution for volumetric shadows. Unlike OSM or other techniques using piece-wise linear functions, the reconstructed signal is always smooth. Additionally, FOM are more stable under translation and less sensitive to variations of depth range [JB10]. While Jansen et al. demonstrate their algorithm with billboard particles, it can be used for any participating media representation, including volumes.
However, as already hinted at, FOM can suffer from ringing. This artifact can be mitigated by reducing the depth range, using smaller α values or less particles. These strategies cause the input signal to have less high frequency components, which is easier to encode with a limited set of coefficients, resulting in reduced ringing. Figure 3.17 shows how a too large depth range can result in ringing, visible as erroneous self shadowing artifacts.

Despite this disadvantage, FOM was chosen as basis for the unified volumetric shadow solution for this thesis. Deciding factors were the fact that it gives smooth results, is stable and that it can be used for both volumes and particles without an intermediate representation (such as the voxelized extinction cascades). As recommended by Jansen et al. eight coefficients spread over a set of two RGBA16F textures are used. The following subsections describe how the FOM algorithm is used to implement volumetric shadows for all supported light sources and medium representations.

Participating Media Volumes

Since FOM works with depth-transmittance samples and is independent of the sample order, volumes can be trivially supported by taking multiple samples in-

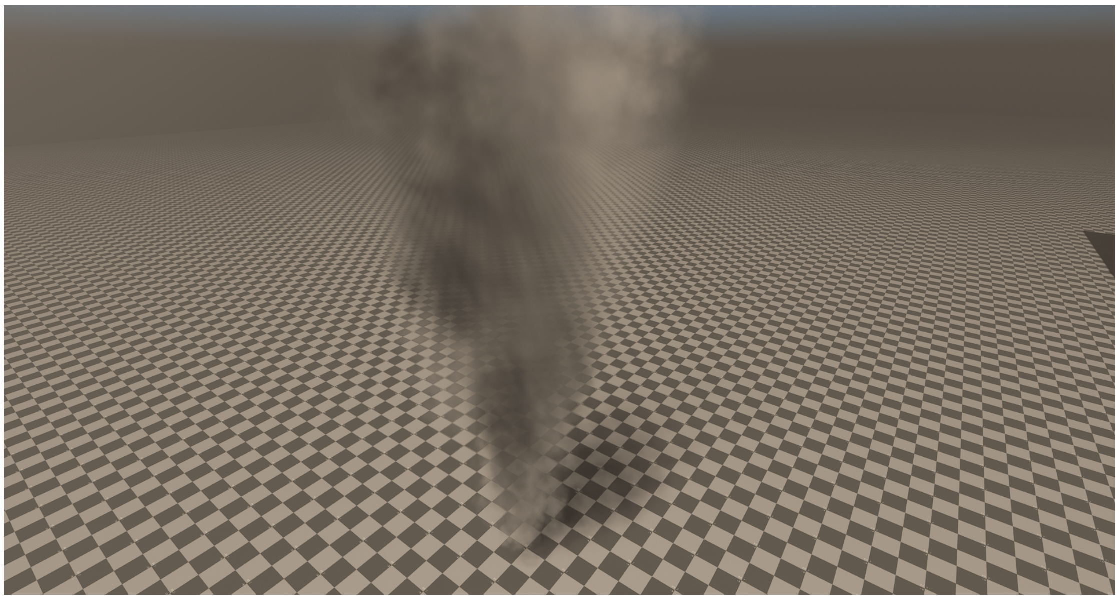


Fig. 3.17: Erroneous self shadowing of billboard particles caused by ringing.

side the volume and accumulating them inside the shader. Samples are taken by intersecting a ray originating from the light source with the bounding geometry of the volume and marching along the intersecting ray segment. At each sample position, the extinction value of the medium is determined. Assuming constant extinction along the current ray interval, the transmittance can be computed from the extinction sample using Beer's law and the ray marching step size. The distance to the light source is implicitly given by the distance parameter t of the ray. At the end of each iteration, these two values along with the current set of coefficients are passed into the *fourierOpacityAccumulate()* function given in listing 3.16. See listing 3.18 for a short implementation of this ray marching scheme.

```
float fourierOpacityGetTransmittance(float3 o, float3 d, float t0,
        float depthScale, int stepCount, float stepSize,
        out float4 result0, out float4 result1)
{
    result0 = 0.0;
    result1 = 0.0;
    for (int i = 0; i < stepCount; ++i)
    {
        float t = i * stepSize + t0;
        float3 rayPos = o + d * t;

        float extinction = getExtinction(rayPos);

        float transmittance = exp(-extinction * stepSize);
        float depth = t * depthScale;
        fourierOpacityAccumulate(depth, transmittance, result0, result1);
    }
}
```

Listing 3.18: Retrieving transmittance from FOM coefficients.

3.3.3 Local Lights

Local lights include spot lights and point lights. Depending on the light's size in screen space a FOM resolution of 128x128 or 256x256 is used. Each light allocates its FOM out of a larger atlas using a quadtree allocator. In order to avoid a naive cube map approach with six FOM, point lights use the octahedron mapping function proposed by Meyer et al. [MSS+10]. This function maps any point on the unit sphere to a point inside a square. See figure 3.18 for a visualization of this process.

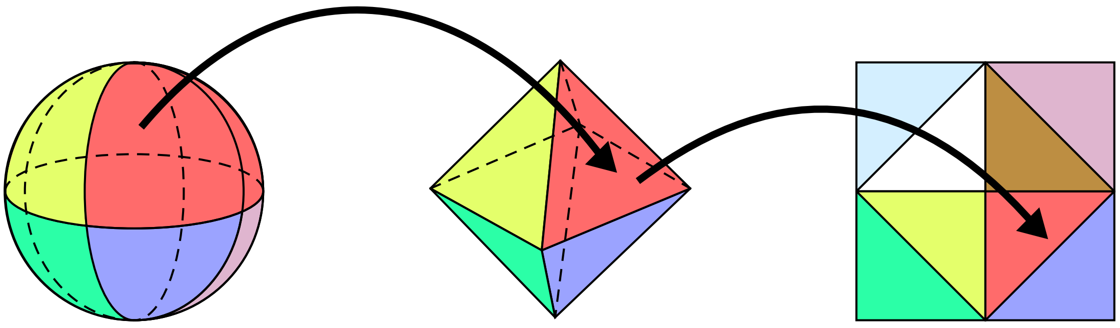


Fig. 3.18: Mapping a point on a sphere to a point in a square with octahedron mapping. Source: Cigolle et al. [CDE+14], modified.

Since a normalized direction vector is equivalent to a point on the unit sphere centered around the origin, octahedron mapping can be used to map any direction vector to a coordinate inside a square texture. The downside of this approach is that hardware rasterization can no longer be used to fill the FOM. Instead, a compute shader marches along the ray segment given by the light source position, the direction corresponding to each FOM texel and the light source radius. In each iteration, every local participating media volume is tested against the current sample position. The extinction value of all overlapping volumes is added and used to update the coefficients as shown in listing 3.18.
Particles are processed in the same pass: Each thread iterates over all particles and tests for intersection between the particle and the ray corresponding to the texel of the current thread. If the ray intersects the particle, the opacity at the intersection is computed. Opacity (converted to transmittance) and the distance to the intersection point are then used to update the coefficients with *fourierOpacityAccumulate()*.
Spot lights are computed with the same shaders. The only difference is that the ray is computed with an inverse (spot light) shadow matrix, not by using the octahedron mapping function.
In order to make this approach scalable, particles and volumes outside the radius of the light should be culled on the CPU. Besides the simpler handling of volumetric point light shadows, bad hardware rasterizer efficiency at low resolutions is another motivating factor for choosing compute shaders.
As mentioned in the discussion of FOM, the technique is depth range dependent.

For local lights it is assumed that the light radius is relatively small and rarely exceeds distances of 16 meters. If this assumption is violated, ringing and erroneous self-shadowing artifacts can occur. These artifacts can be fixed by either using more coefficients are limiting the depth range. Figure 3.19 shows the same scene as figure 3.15, but with volumetric shadows enabled. The local spherical volume to the right casts a shadow onto itself, the opaque scene and the global medium. The billboard particles cast clearly visible shadows onto themselves. Shadows cast onto the scene and the global medium are present, but not very visible.

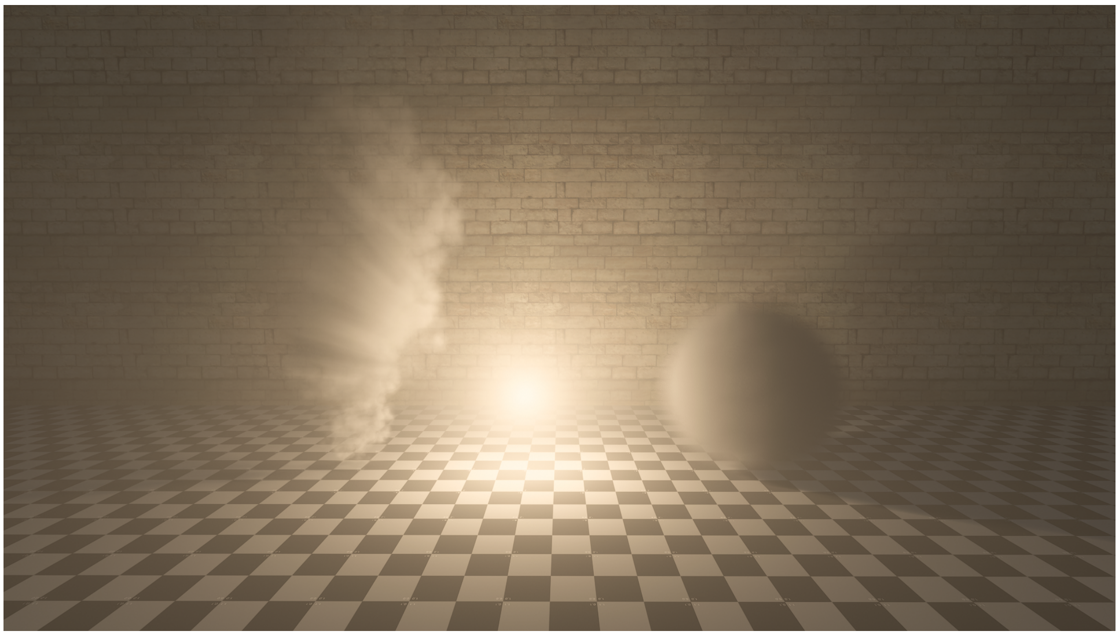


Fig. 3.19: Billboard particles, a global participating medium and a local volume lit by a point light with volumetric shadows.

Quadtree Allocator

The aforementioned quadtree allocator is described in the following. Its purpose is to manage a quadratic domain of a given size. It tries to satisfy requests for free areas of different sizes inside this domain, keeping track of which areas are already in use. A request specifies the desired size of the area and upon success an offset into the quadratic domain managed by the allocator is returned. Such an allocator can be used to manage the contents of a texture atlas. In the implementation for this thesis it is used to manage the shadow map atlas and FOM atlas for local lights. See figure 3.20 for a visualization of the shadow atlas. Note that it contains shadow maps of different resolutions.

Internally, the allocator is implemented in terms of a quadtree. The quadtree is a tree structure where every node evenly divides its area into four child nodes. Each

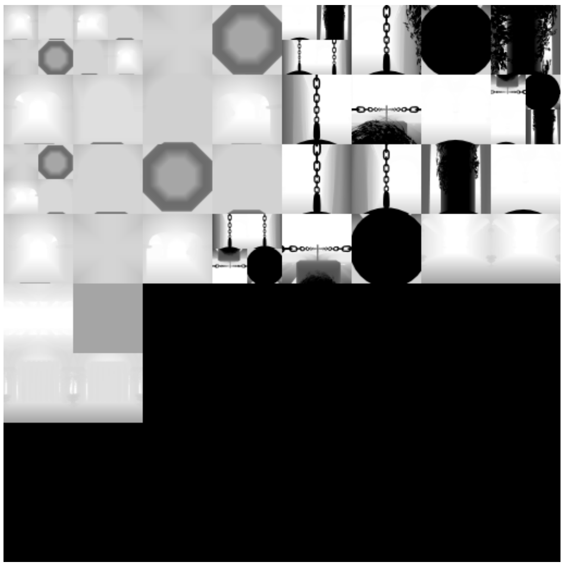


Fig. 3.20: The shadow atlas for local lights is managed with a quadtree allocator.

node then stores the size of the largest free area inside its own area. Allocating a free area is done by recursively traversing the tree starting at the root node. All child nodes are traversed until a free node of the requested size is found. If the largest free area inside the area of a given node is too small to satisfy the request, its children do not need to be traversed, speeding up the process. If a free node was found, the largest free area value of each node is updated to account for the now allocated node. This is done while unwinding the stack, ensuring that only the parent nodes of the allocated node update their value. Freeing a previously allocated area works in a similar way.
Since shadow maps are commonly square, using a quadtree to manage the free area is a reasonable choice. In order to limit the maximum recursion depth as well as the required memory, a lower bound for the requested sizes is used. As shadow maps smaller than e.g. 64x64 are unpractical and rarely useful, this lower bound does not affect the practical usefulness of the quadtree allocator.

3.3.4 Directional Lights

Unlike local lights where it can be assumed that the depth range is fairly limited, directional lights commonly have very large depth ranges. Furthermore, directional lights need to cover both close and far away scene elements with sufficient FOM resolution.
The latter problem can be solved in a similar way as with regular shadow maps. Cascaded shadow maps (CSM) are a solution to this problem. With CSM the frustum is split into a small set of cascades, where every cascade covers a larger area than the previous one. Each cascade is then assigned a shadow map of the same resolution, where cascades close to the camera effectively get a higher shadow map resolution on account of less space being covered by the shadow map. The implementation framework for this thesis already supports CSM. Extending this

system for FOM is trivial: The cascade shadow matrices can simply be reused for rendering FOM into a separate set of textures. In order to simplify sampling these textures, they are combined into an array texture. A resolution of 256x256 gives sufficiently detailed results.
The large depth range poses two problems. Firstly, it makes ray marching the whole depth range with a sufficiently small step size prohibitively expensive. Secondly, it violates the assumption of a reasonably small depth range when computing the FOM coefficients, leading to very visible artifacts. The first problem can be solved by only ray marching along the ray segment intersecting a participating medium volume. This is done by rendering proxy geometry for each volume. Box-shaped volumes use a transformed unit cube mesh and spherical volumes use a simple sphere mesh. The pixel shader invocations spawned by the proxy geometry then analytically compute the ray intersections with the volume and do the ray marching.
A solution to the second problem is to reduce the depth range by bounding it to the minimum and maximum participating media volume distances. A naive but reasonably fast way of doing this is to render the proxy geometry to a set of two depth buffers per cascade. One depth buffer stores the closest depth and the other one the furthest. Sampling both textures gives the tight depth bounds needed to minimize FOM artifacts. Unfortunately, this means that for four cascades, eight depth buffers must be prepared. Additional overhead is caused by having to sample these textures during FOM rendering and during scene rendering, when the FOM is looked up. Figure 3.21 shows the same scene as figure 3.17, but with a bounded depth range, successfully removing the ringing artifacts.

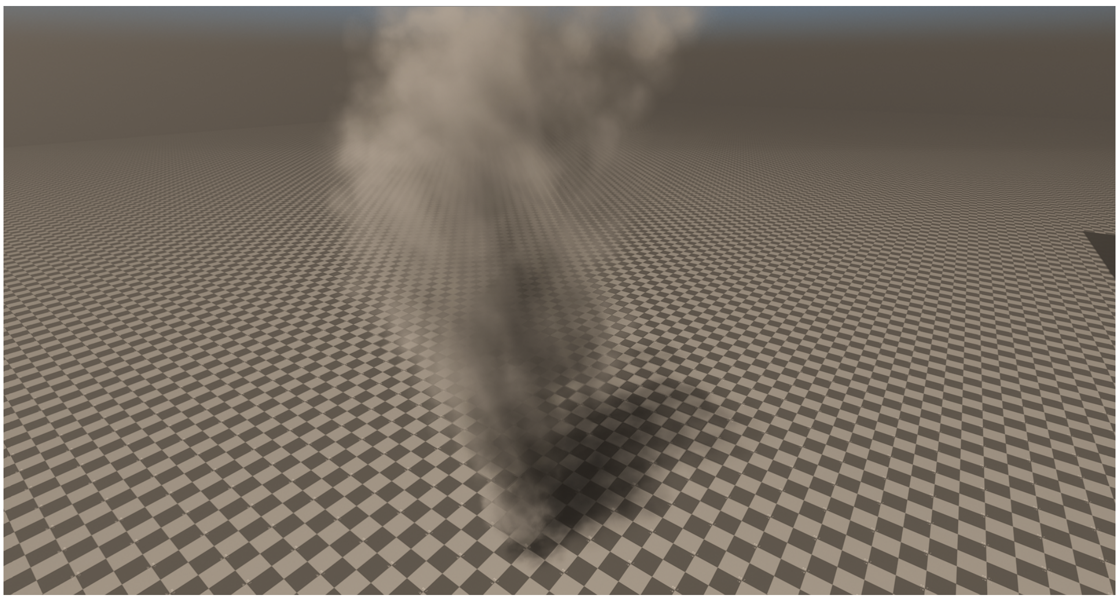


Fig. 3.21: Bounding the depth range can fix ringing artifacts.

A limitation of this approach for rendering FOM for directional lights is that it is incompatible with global participating media. Global media do not have any bounds, which means that the depth range cannot be reduced.

3.4 Measurement Method

The aim of the measurement method discussed in the following is to produce profiling data to evaluate the performance of volumetric lighting, volumetric shadows and the proposed checkerboard rendering technique. Furthermore, the data is used to confirm that merging the V-Buffer and In-Scattering shaders is indeed a performance improvement. Finally, visual quality and applicability of the implemented solution is discussed in chapter 4.

3.4.1 Test Configurations

The framework used to implement the techniques discussed in this thesis uses a thin abstraction over the modern low level graphics APIs Vulkan and DirectX 12. These APIs require the application to explicitly manage synchronization on the GPU. In order to make this task easier on the programmer, a render graph system is used. All resources are registered with the render graph prior to rendering a frame. The render graph is then informed by the programmer about how each resource is used in the frame. This gives the render graph global knowledge of the resource dependencies in a frame and allows it to automatically deduce the required barriers needed to properly synchronize access to the resources. Executing all passes is also done by this system, inserting synchronization commands between passes as required.
Since it orchestrates synchronization and recording of all other GPU commands, it can also automatically insert GPU timestamps to measure the elapsed time of each pass. This ensures that all GPU work can be profiled without having to manually manage timestamps. The measurements obtained this way also include time spent waiting on barriers automatically issued by the render graph. This is a deliberate choice, as the barriers are required in any real-world scenario and therefore give more meaningful measurements of the total cost of each effect.

Collection of measurement data is done automatically to increase reproducibility. Since Volumetric Fog is independent of the geometry in the scene, only a single scene is tested. This scene has nine point lights with a large radius. All point lights cast regular and volumetric shadows. The participating media consist of one global volume and a local volume in the center of the scene. The local volume in the center uses a scrolling 3D density texture. This setup was chosen to represent a typical game scene. Furthermore, up to 512 additional unshadowed point lights can be added to to test the scalability of the technique. Figure 3.22 shows the test scene.
While Volumetric Fog is independent of the scene geometry, long range volumetric lighting is not. In the test scene shown in figure 3.22 the geometry is close enough

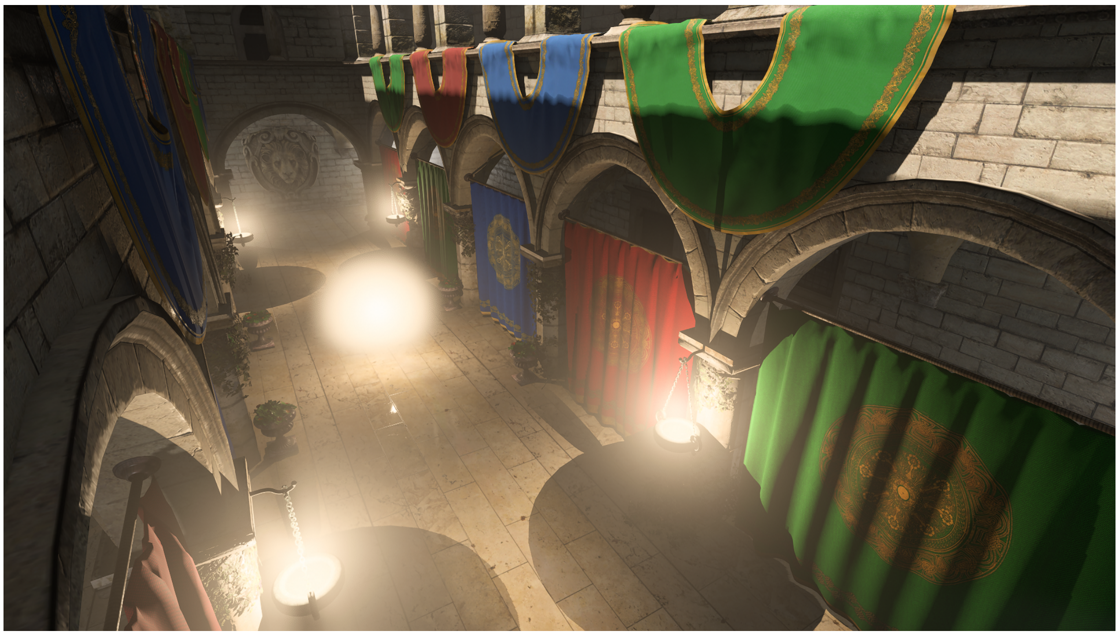


Fig. 3.22: Volumetric Fog test scene.

to be completely inside the range of Volumetric Fog. As a consequence, the ray marching shader for long range volumetric lighting skips ray marching. In order to evaluate the performance impact of the fallback technique, a second scene was set up. This scene features long view distances, ensuring that a ray is marched for almost all pixels. Figure 3.23 shows what this scene looks like.

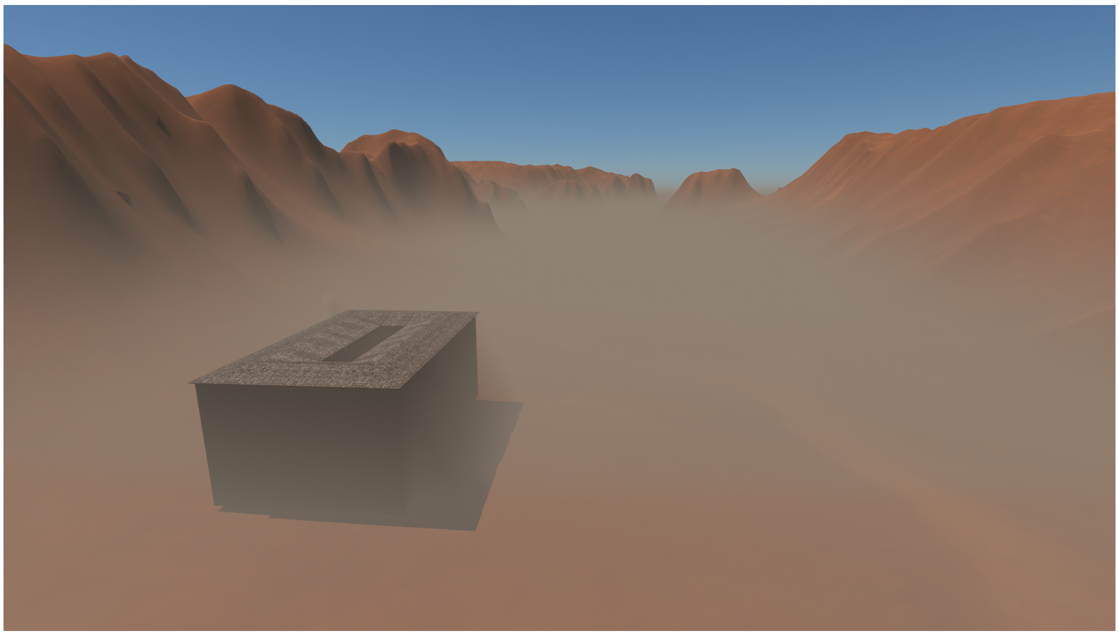


Fig. 3.23: Long range volumetric lighting test scene.

Each scene is profiled with eight different combinations of enabled features, as shown in table 3.3. Each configuration is profiled for 256 frames. The measurements gathered this way are then averaged and written to a file. Profiling is done in full screen mode at a resolution of 1920x1080. This process is done for two different Volumetric Fog volume resolutions: 160x90x64 and 240x135x128. The first resolution corresponds to the one proposed by Wronski when he first introduced the technique [Wro14]. The latter was chosen to reflect the higher quality results achievable by modern hardware. A single texel of this resolution corresponds to a 8x8 pixel tile when rendering at 1920x1080.

Config.	Merged V-Buffer & In-Scatter	Checkerboard Rendering	Volumetric Shadows
1	no	no	no
2	yes	no	no
3	no	yes	no
4	yes	yes	no
5	no	no	yes
6	yes	no	yes
7	no	yes	yes
8	yes	yes	yes

Table 3.3: Tested configurations.

In configurations where the V-Buffer and In-Scattering calculations are separate, two RGBA16F 3D textures are used to write out the participating media properties. Configurations without checkerboard rendering include the temporal filter in the In-Scattering shader. If checkerboard rendering is enabled, temporal filtering is done in a separate pass. Enabling volumetric shadows causes the In-Scattering shader and the regular (opaque scene) lighting shader to sample the relevant FOM textures. In the following all relevant passes are listed:

- **FOM Directional Lights Depth**
  Generate depth buffers to tighten the depth bounds of directional light FOM.

- **FOM Directional Lights**
  Rasterize particles and volumes into directional light FOM.

- **FOM Local Lights**
  Compute FOM for local lights.

- **Volumetric Fog V-Buffer**
  Voxelize participating media properties into V-Buffer. Runs only if not in merged configuration.

- **Volumetric Fog In-Scattering**
  Read V-Buffer, compute in-scattered lighting and apply temporal filter (if not in checkerboard configuration). Runs only if not in merged configuration.

- **Volumetric Fog Merged**
  Voxelize participating media properties, compute in-scattered lighting and apply temporal filter (if not in checkerboard configuration). Runs only if in merged configuration.

- **Volumetric Fog Filter**
  Fill checkerboard holes and apply temporal filter to In-Scattering Buffer. Runs only if checkerboard rendering is enabled.

- **Volumetric Fog Integrate**
  March through filtered In-Scattering Buffer and compute final result.

- **Volumetric Lighting Depth Downsample**
  Downsample full resolution depth buffer to half resolution with min/max checkerboard downsampling.

- **Long Range Volumetric Lighting Ray Marching**
  Compute long range volumetric lighting.

- **Volumetric Lighting Apply**
  Apply both Volumetric Fog and the long range volumetric lighting to the scene.

3.4.2 Test Systems

The techniques have been tested on two different test systems: A desktop computer and a laptop. Table 3.4 shows the hardware specification of the desktop computer test system.

CPU	Intel i5 4690K @3.5 GHz
GPU	AMD Radeon RX 5700 XT 8 GB
RAM	16 GB
OS	Windows 10

Table 3.4: Hardware specification of the high end AMD desktop test system.

The system shown in table 3.5 is a laptop with a dedicated NVIDIA laptop GPU.

CPU	Intel i5 7300HQ @3.5 GHz
GPU	NVIDIA GeForce GTX 1050 2 GB
RAM	8 GB
OS	Windows 10

Table 3.5: Hardware specification of the laptop test system.

4

Results and Discussion

4.1 Volumetric Lighting

4.1.1 Merged V-Buffer and In-Scattering

Table 4.1 shows the results of voxelizing participating media properties in a separate shader compared to doing it in the In-Scattering shader. The table shows that combining these workloads is a consistent win on both test systems, reducing the computation time by up to 9.1 %.

	RX 5700 XT	GTX 1050
Separate	0.453 (0.0275 + 0.426)	2.734 (0.300 + 2.434)
Merged	0.412	2.582
Difference	-0.041 (-9.1 %)	-0.152 (-5.6 %)

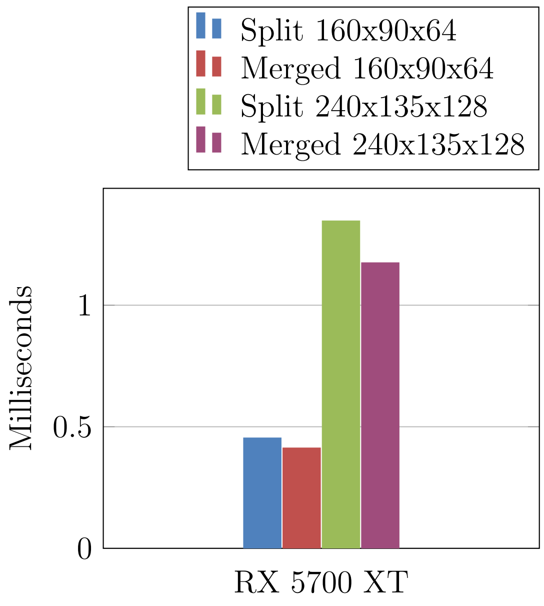
Table 4.1: Separate vs. merged V-Buffer and In-Scattering timings in milliseconds at a volume resolution of 160x90x64.

As table 4.2 demonstrates, this performance improvement also holds at a considerably higher volume resolution of 240x135x128. Interestingly, the RX 5700 XT test system seems to profit more from this improvement as the resolution increases, while the laptop test system does not seem to be affected much by the higher resolution.

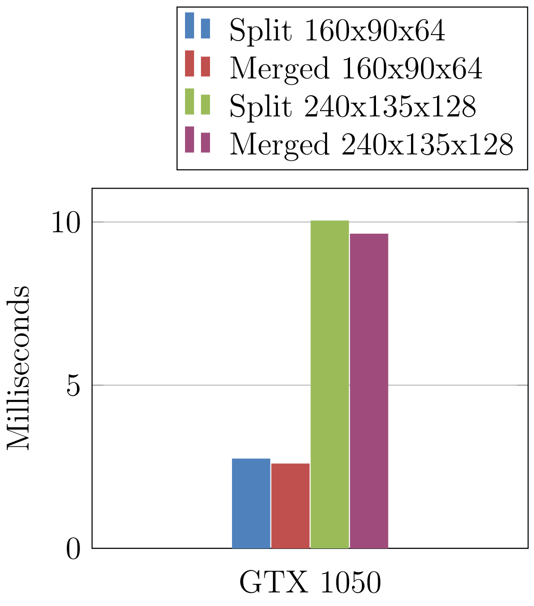
	RX 5700 XT	GTX 1050
Separate	1.346 (0.147 + 1.199)	10.029 (1.249 + 8.780)
Merged	1.174	9.626
Difference	-0.172 (-12.8 %)	-0.403 (-4.0 %)

Table 4.2: Separate vs. merged V-Buffer and In-Scattering timings in milliseconds at a volume resolution of 240x135x128.

These results confirm the findings of Drobot et al. [Dro17b], making this a small but simple and consistent performance improvement. However, while combining voxelization and in-scattering may seem like an obvious optimization, it has a drawback that may make it unsuitable for some applications. Separating these tasks allows for more material variety as the V-Buffer can be written to with different artist authored shaders. This could be done by rendering into the V-Buffer with classic rasterization. Combining both steps limits the system to fixed function materials as all participating media need to be voxelized by the same shader. For the sake of performance, this trade-off was deemed acceptable for this thesis. Figure 4.1 visualizes the performance impact of this optimization. Note that the values in both tables as well as figure 4.1 only include the timings of voxelization and in-scattering and not of the whole effect. Since this performance improvement is so consistent, all results in the following sections were obtained with this optimization enabled.



(a) Desktop Test System



(b) Laptop Test System

Fig. 4.1: Separate vs. merged V-Buffer and In-Scattering timings in milliseconds at different volume resolutions.

4.1.2 Checkerboard Rendering

The results in table 4.3 show that enabling checkerboard rendering gives a considerable performance improvement on both systems, despite moving temporal filtering into its own shader. Especially the weaker laptop test system seem to profit from checkerboard rendering with a reduction in computation time of up to 27.7 %.
Table 4.4 shows that checkerboard rendering scales very well with increasing volume resolution on both test systems, with the laptop test system profiting even more from this optimization. The higher performance improvement on the laptop

	RX 5700 XT	GTX 1050
CB Off	0.412	2.582
CB On	0.325 (0.275 + 0.050)	1.866 (1.638 + 0.228)
Difference	-0.087 (-20.9 %)	-0.716 (-27.7 %)

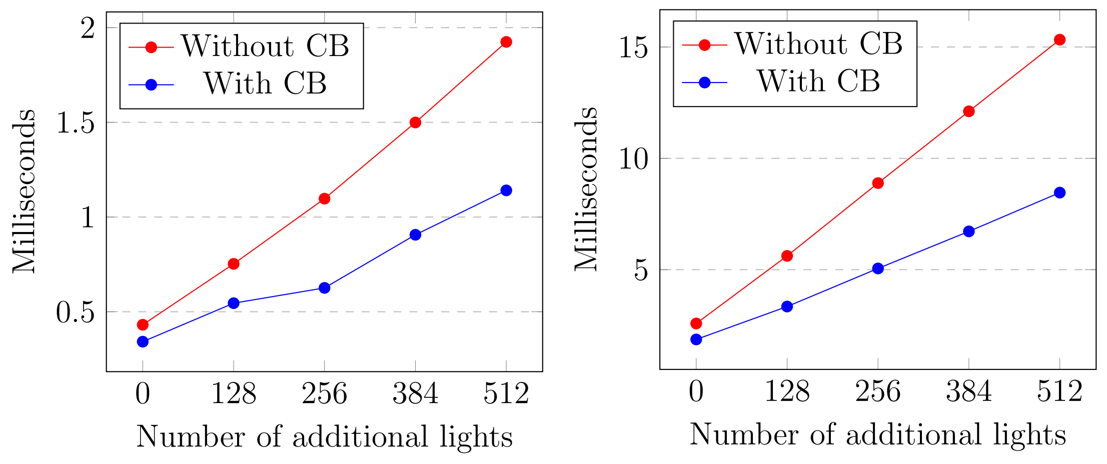
Table 4.3: Timings in milliseconds without and with checkerboard rendering at a volume resolution of 160x90x64.

system is most likely due to a worse bandwidth compared to the modern AMD test system, causing this system to profit more from bandwidth saving optimizations.

	RX 5700 XT	GTX 1050
CB Off	1.174	9.626
CB On	0.935 (0.711 + 0.224)	6.167 (5.176 + 0.991)
Difference	-0.239 (-20.3 %)	-3.459 (-35.9 %)

Table 4.4: Timings in milliseconds without and with checkerboard rendering at a volume resolution of 240x135x128.

Since the performance of filling the holes left by the checkerboard pattern and performing temporal filtering is independent of the number of lights, checkerboard rendering scales favorably with increasing light count, as demonstrated in figure 4.2. The data for this figure was obtained by adding additional unshadowed point lights to the scene.

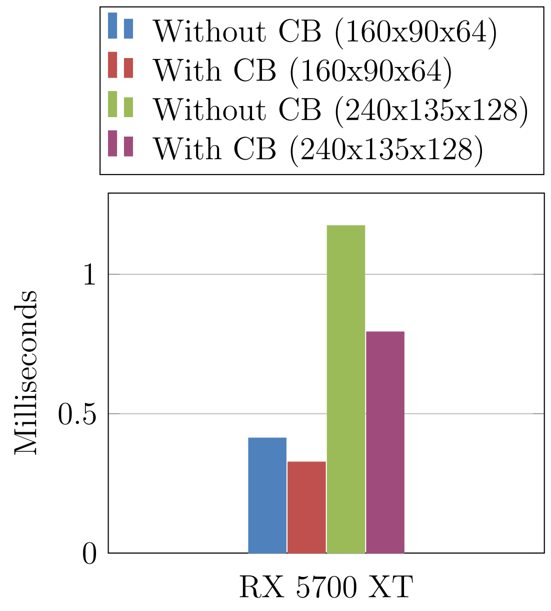


(a) Desktop Test System

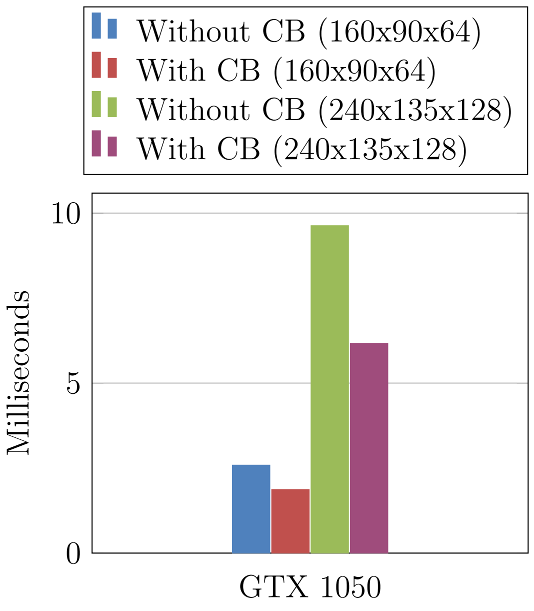
(b) Laptop Test System

Fig. 4.2: Timings in milliseconds for computing in-scattered light and performing temporal filtering without and with checkerboard rendering at a volume resolution of 160x90x64.

The performance improvement of checkerboard rendering is visualized in figure 4.3.



(a) Desktop Test System



(b) Laptop Test System

Fig. 4.3: Timings in milliseconds without and with checkerboard rendering at different volume resolutions.

Figure 4.4 and figure 4.5 show the same scene without and with checkerboard rendering at a volume resolution of 160x90x64. Visually, both results are indistinguishable. Hypothetical artifacts would be even less visible at higher volume resolutions.
These results confirm that checkerboard rendering is a valuable optimization, improving performance considerably without introducing visible artifacts.

4.1.3 Volumetric Fog

Table 4.5 shows the timings of all Volumetric Fog passes at a volume resolution of 160x90x64. The algorithm is fast enough to be feasible even on weaker hardware such as the laptop version of the GTX 1050. Applying the effect to the scene is relatively expensive, especially on the weaker system. However, this is most likely related to the memory bandwidth cost of performing a full screen pass and can be mitigated by applying the effect in the lighting shader.
Figure 4.6 shows the results of rendering Volumetric Fog at a volume resolution of 160x90x64. Effects such as the floating white light in the right corner are not affected by the resolution. However, light shafts caused by high frequency shadows such as the ones cast by the vines around the pillar in the center of the figure appear a bit blurry. Despite this blurriness, the effect is still stable under motion due to the temporal filtering.

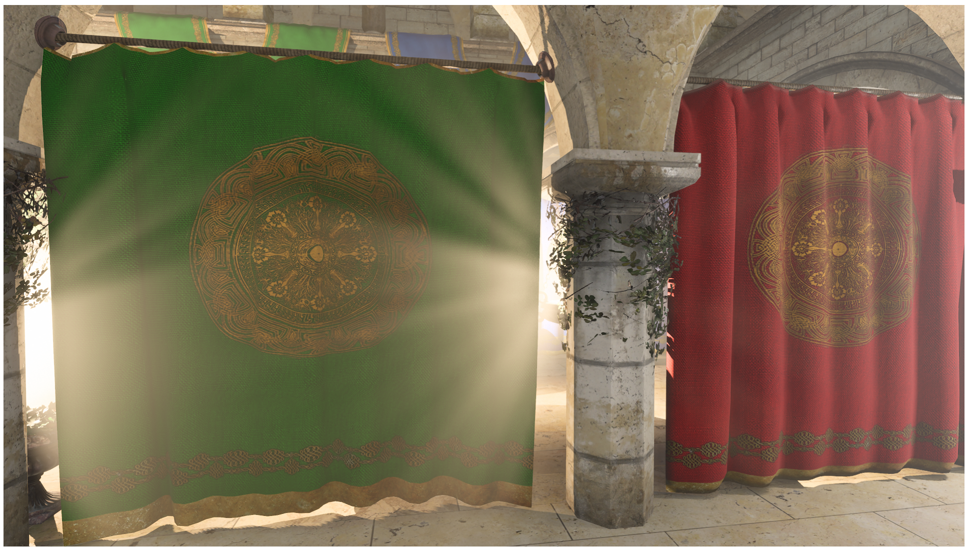


Fig. 4.4: Rendering with a volume resolution of 160x90x64 without checkerboard rendering.

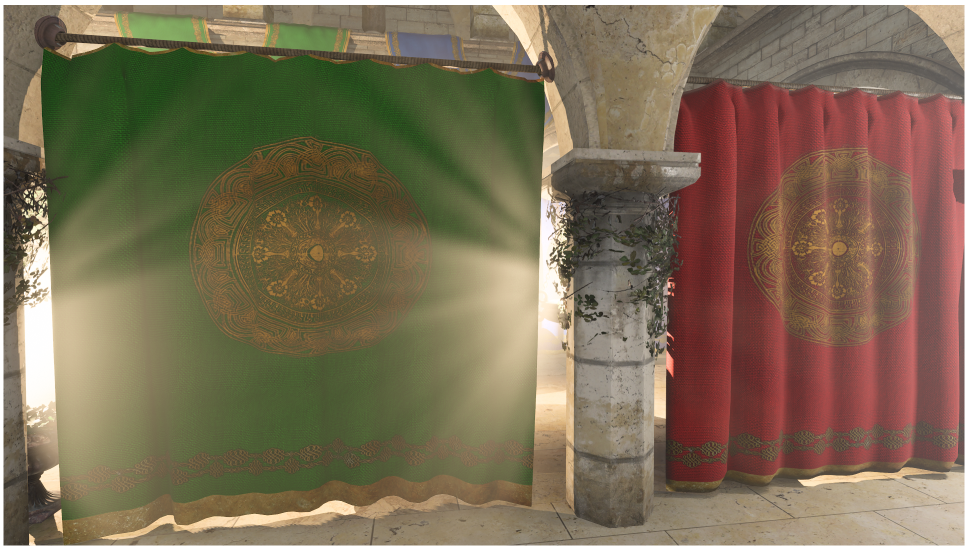


Fig. 4.5: Rendering with a volume resolution of 160x90x64 with checkerboard rendering.

	RX 5700 XT	GTX 1050
In-Scatter	0.291	1.638
Filter	0.051	0.228
Integrate	0.059	0.169
Apply	0.223	1.003
Total	0.624	3.038

Table 4.5: Performance timings in milliseconds of all Volumetric Fog passes at a volume resolution of 160x90x64.

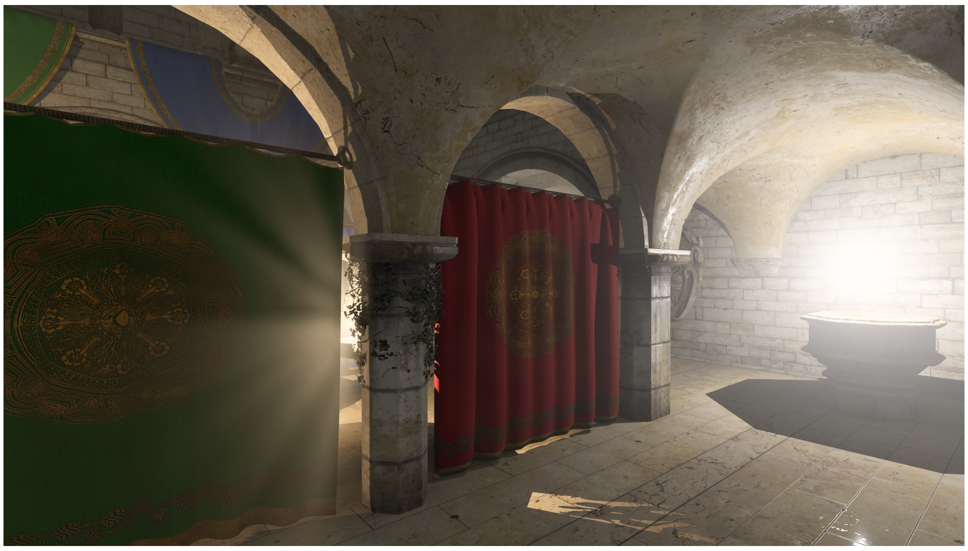


Fig. 4.6: Rendering Volumetric Fog at a volume resolution of 160x90x64.

The timings shown in table 4.6 were obtained at a volume resolution of 240x135x128. They demonstrate that such a resolution is justifiable on modern hardware such as the RX 5700 XT. The other test system exhibits unacceptable performance at this resolution.

	RX 5700 XT	GTX 1050
In-Scatter	0.752	5.176
Filter	0.224	0.991
Integrate	0.191	0.723
Apply	0.224	1.020
Total	1.391	7.910

Table 4.6: Performance timings in milliseconds of all Volumetric Fog passes at a volume resolution of 240x135x128.

Figure 4.7 shows the same scene as figure 4.6 but rendered at a resolution of 240x135x128. While the floating white light looks identical to the lower resolution result, the light shafts in the center of the figure are much more detailed. Although this effect is still relatively fast at 1.39 ms, the increased resolution does come with a performance penalty that might not be worth it for some applications, even on modern hardware. For such applications, rendering Volumetric Fog at a lower resolution can still achieve good results.
Visually, the implemented Volumetric Fog algorithm delivers satisfying results. Temporal filtering and TAA manage to hide almost all undersampling artifacts, while suffering only from slight ghosting. As the comparison between the two resolutions showed, the effect also gives good results at lower resolutions, especially in the absence of shadow maps with high frequency detail. Ghosting and leaking

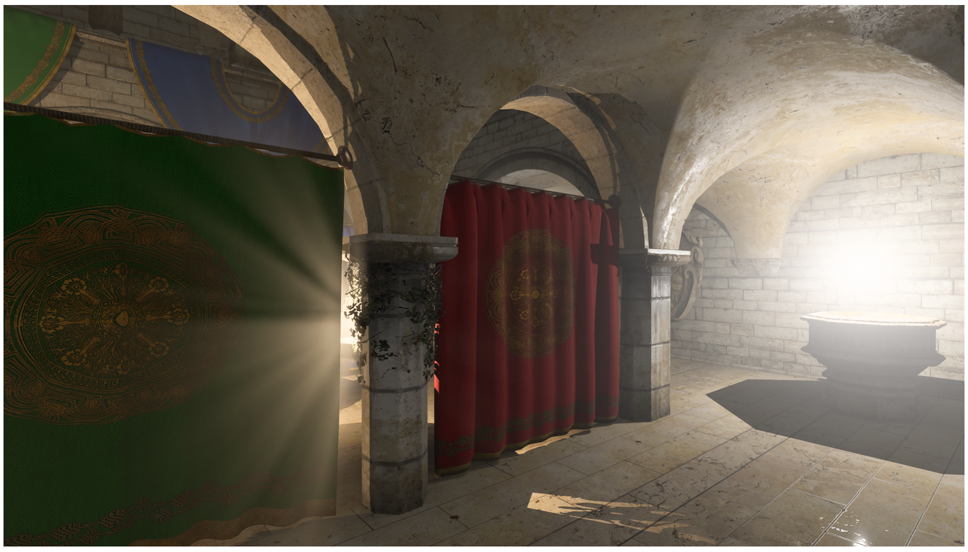


Fig. 4.7: Rendering Volumetric Fog at a volume resolution of 240x135x128.

remain the primary artifacts of the temporal filtering stage. If all filtering could be moved to screen space, these artifacts would likely be gone.

4.1.4 Long Range Volumetric Lighting

Table 4.7 shows the performance timings in milliseconds of the long range volumetric lighting test scene. Long range ray marching is too slow for the tested laptop GTX 1050. This fallback would either need to be disabled on such hardware or it would need to be further optimized by running at quarter resolution instead of half resolution. While the performance might be acceptable on the RX 5700 XT system, it would also profit from an optimized ray marching pass.

	RX 5700 XT	GTX 1050
In-Scatter	0.088	0.530
Filter	0.051	0.239
Integrate	0.057	0.175
Depth Downsample	0.0001	0.074
Ray Marching	0.316	4.42
Apply	0.193	0.828
Total	0.705	6.266

Table 4.7: Performance timings of the long range volumetric lighting test scene in milliseconds at a volume resolution of 160x90x64.

As figure 3.23 shows, long range volumetric lighting would likely profit from some form of volumetric shadowing. Since the directional light volumetric shadows are limited to the range of the cascaded shadow maps, most of the ray marching samples are not covered by a FOM cascade. At the cost of some performance, volumetric shadows could be realized by performing secondary ray marches towards the light source.

4.2 Volumetric Shadows

Table 4.8 shows the timings of generating the FOM textures and the overhead of sampling them in the relevant volumetric lighting passes. Note that all FOM textures for all lights are recomputed every frame. For the test scene, this includes nine local lights and one directional light with four cascades. As an optimization, time slicing could be implemented in the future. That way only a subset of all active lights would have their FOM textures recomputed each frame.
Rendering the depth buffers for the directional light seems to be equally fast on both systems. The reason for this is most likely that the workload is so light that the GPU is undersaturated. Curiously, the laptop system seems to be faster at rendering the directional light FOM textures. On the other hand, computing FOM textures for local lights is a lot more expensive on that system. On the desktop test system, both passes are equally as expensive. This might be a hint that the GTX 1050 would profit from performing local light FOM generation with rasterization too. Since local lights take global participating media into account, they need to march the whole ray, not just a subsection intersecting a local participating medium. As a consequence, the shader might run longer and require more bandwidth as all relevant media need to be sampled at each iteration. Since the RX 5700 XT features a much higher bandwidth than the other GPU, it suffers less from these additional memory accesses. Sampling the FOM textures in the Volumetric Fog In-Scattering shader incurs only a minor performance degradation. This is even more so the case for the long range ray marching, where the overhead is so small that it is effectively non-existent. The reason for this low overhead is likely that the rays cover such a long distance that only few samples are inside the range of the directional light shadow cascades. Overall, generating FOM for directional lights is so inexpensive that it is feasible on all tested systems. FOM for local lights are more expensive, but still justifiable on modern hardware. It might be worth it to disregard volumetric shadows cast by global participating media and also use rasterization to generate local light FOM textures.

	RX 5700 XT	GTX 1050
Dir. Lights Depth	0.022	0.025
Dir. Lights	0.122	0.047
Local Lights	0.129	2.262
In-Scattering Overhead	0.016	0.128
Ray Marching Overhead	0.000	0.000
Total	0.289	2.462

Table 4.8: Timings in milliseconds for generating the FOM textures and sampling them for volumetric lighting.

Table 4.9 shows the overhead of sampling the FOM textures in the Volumetric Fog In-Scattering shader at a resolution of 240x135x128. Since the costs of generating the textures and sampling them in the ray marching shader are independent of the Volumetric Fog volume resolution, these timings are omitted from the table. However, they are still included in the sum in the second row.

	RX 5700 XT	GTX 1050
In-Scattering Overhead	0.040	0.195
Total	0.321	1.637

Table 4.9: Timings in milliseconds for sampling the FOM textures at a volume resolution of 240x135x128. "Total" includes all other timings from table 4.8.

Table 4.10 shows the performance overhead of enabling volumetric shadows for the two different resolutions. The same data is visualized in figure 4.8.

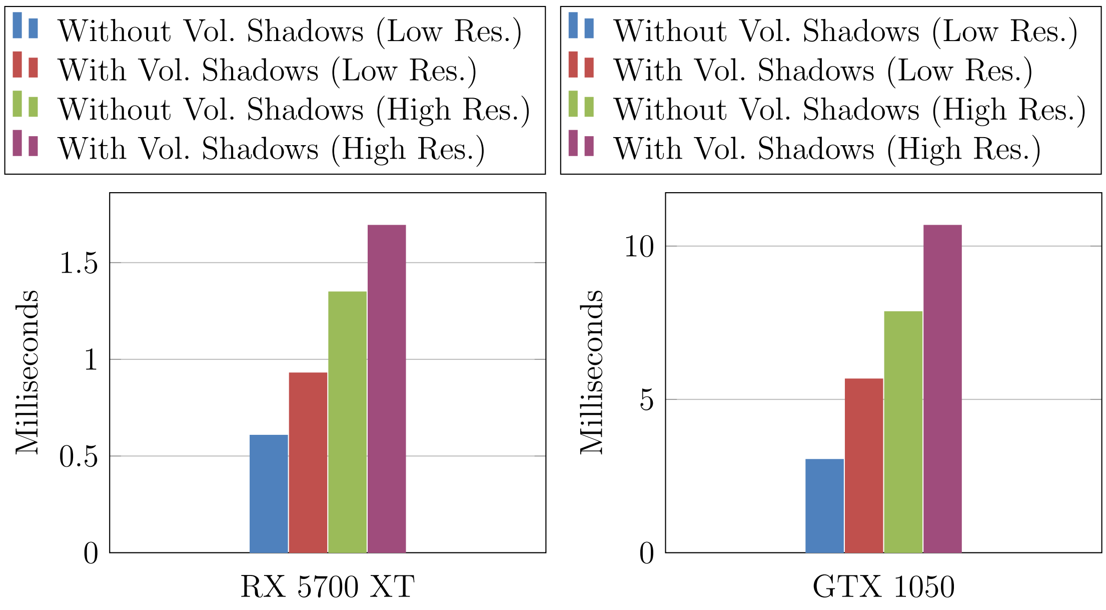
	RX 5700 XT	GTX 1050
Without Volumetric Shadows (Low Res.)	0.607	3.038
With Volumetric Shadows (Low Res.)	0.929	5.663
Without Volumetric Shadows (High Res.)	1.348	7.857
With Volumetric Shadows (High Res.)	1.692	10.673

Table 4.10: Timings in milliseconds for the complete volumetric lighting solution without and with volumetric shadows at different volume resolutions.

With regards to visual quality, the impact of volumetric shadows is very scene dependent. Scenes like the one depicted in figure 3.19 clearly profit from this feature. It appears that volumetric shadows are important in the presence of participating media with high extinction. Furthermore, the quality of billboard particles seems to always improve with volumetric shadows. As a consequence, scenes like the Amazon Lumberyard Bistro shown in figure 1.1 could likely disregard volumetric shadows in favor of some additional performance. This is especially true for weaker hardware, where such a feature is not as easily justifiable. However, for applications with high extinction participating media running on modern hardware, the volumetric shadows technique proposed in this thesis might be a good solution.

4.3 A Unified Solution

The proposed implementation achieves the initial goal of creating a unified solution for both billboard particles and participating media volumes. Billboard particles and other transparent objects can sample the Volumetric Fog result texture, making them integrate correctly with other scene elements. This unification also extends to volumetric shadows, where FOM manages to represent the transmittance curves of both types of participating media representations. Figure 4.9 shows



(a) Desktop Test System (b) Laptop Test System

Fig. 4.8: Timings in milliseconds for the complete volumetric lighting solution without and with volumetric shadows at different volume resolutions.

a scene, where a particle system is placed inside a local participating media volume. Both the particles and the volume are blended seamlessly and correctly cast shadows onto themselves, each other and the scene.

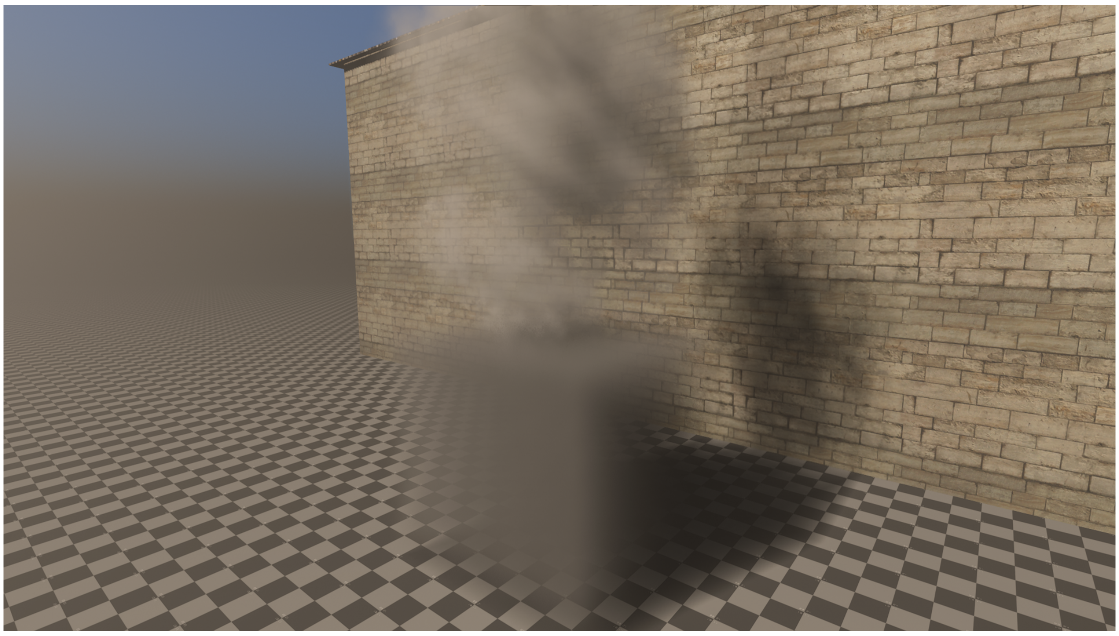


Fig. 4.9: Billboard particles seamlessly integrating with a local participating medium. Both participating media representations cast shadows onto one another and the scene.

5

Conclusion

5.1 Conclusion

This research aimed to detail the implementation of a modern, fast and unified solution for volumetric lighting with volumetric shadows. Adapting the Fourier Opacity Mapping algorithm, a technique originally intended for particles, to also support participating media volumes proved to be a successful strategy for creating a unified volumetric shadows solution for both participating media representations. Furthermore, it was shown that applying the concept of checkerboard rendering to the Volumetric Fog algorithm is a valuable technique for lowering the performance impact of volumetric lighting, making this effect viable on a wider range of hardware. Partially deferring temporal filtering of the volume texture to screen space by taking advantage of TAA demonstrated to be a promising solution to combat ghosting artifacts. These advancements helped to achieve the goals set by this thesis. In conclusion, the approaches detailed in this work allow efficient high quality volumetric lighting effects on modern GPUs, while still delivering good results on weaker hardware.

5.2 Future Work

In future work, it would be interesting to research better solutions to the ghosting artifacts inherent to temporal filtering. In addition, finding a more robust strategy against leaking artifacts would be a valuable contribution. The implementation itself could be improved by optimizing long range volumetric lighting and extending it with atmospheric scattering and cloud rendering.

References

And09. Andersson, Johan: *Parallel Graphics in Frostbite – Current & Future*, 2009.

Bau19. Bauer, Fabian: *Creating the Atmospheric World of Red Dead Redemption 2: A Complete and Integrated Solution*, 2019.

BJ13. Bavoil, Louis and Jon Jansen: *Particle Shadows & Cache-Efficient Post-Processing*, 2013.

CDE$^+$14. Cigolle, Zina H., Sam Donow, Daniel Evangelakos, Michael Mara and Morgan McGuire: *Survey of Efficient Representations for Independent Unit Vectors.* Journal of Computer Graphics Techniques (JCGT), 2014, 2014.

CGK19. Cho, Kyungjoon, Kwanghyeon Go and Daeil Kim: *Practical Dynamic Lighting for Large-Scale game Environments*, 2019.

Cha60. Chandrasekhar, Subrahmanyan: *Radiative Transfer.* Dover Publications, New York, 1960.

DGMF11. Delalandre, Cyril, Pascal Gautron, Jean-Eudes Marvie and Guillaume François: *Transmittance Function Mapping.* page 31, 2011.

Dro17a. Drobot, Michal: *Improved Culling for Tiled and Clustered Rendering*, 2017.

Dro17b. Drobot, Michal: *Rendering of Call of Duty Infinite Warfare*, 2017.

DS15. Delmont, Samuel and Peter Sikachev: *Labs R&D: The Rendering Techniques of Deus EX: Mankind Divided and Rise of the Tomb Raider*, 2015.

El 16. El Mansouri, Jalal: *Rendering 'Rainbow Six — Siege'*, 2016.

EMK$^+$06. Engel, Klaus, Hadwiger Markus, Joe M. Kniss, Christof Rezk-Salama and Daniel Weiskopf: *Real-Time Volume Graphics.* A K Peters Ltd, Wellesley, Mass, 2006.

EPG$^+$12. Esteve, Jose, Jamie Portsmouth, Pascal Gautron, Jean-Colas Prunier, Jean-Eudes Marvie and Cyril Delalandre: *Bringing Transmittance Function Maps to the Screen.* DigiPro '12: Proceedings of the Digital Production Symposium, 2012:73, 2012.

EPK12. Esteve, Jose, Jamie Portsmouth and David Koerner: *Adaptive DCT Compression for High Quality Volume Rendering of Dense Media.* 2012, 2012.

FWKH17. Fong, Julian, Magnus Wrenninge, Christopher Kulla and Ralf Habel: *Production Volume Rendering.* SIGGRAPH 2017 Course, pages 1–79, 2017.

GDM11. Gautron, Pascal, Cyril Delalandre and Jean-Eudes Marvie: *Extinction Transmittance Maps.* SIGGRAPH Asia 2011 Sketches, 2011:1, 2011.

Gla14. Glatzel, Benjamin: *Volumetric Lighting for Many Lights in Lords of the Fallen*, 2014.

HG41. Henyey, L. C. and J. L. Greenstein: *Diffuse radiation in the Galaxy.* The Astrophysical Journal, 93:70, 1941.

Hil15. Hillaire, Sebastien: *Physically-based & Unified Volumetric Rendering*, 2015.

JB10. Jansen, Jon and Louis Bavoil: *Fourier Opacity Mapping.* Proceedings of the 2010 Symposium on Interactive 3D Graphics, page 165, 2010.

JB11. Jansen, Jon and Louis Bavoil: *Fast rendering of opacity-mapped particles using DirectX 11 tessellation and mixed resolutions*, 2011.

Jim16. Jimenez, Jorge: *Filmic SMAA: Sharp Morphological and Temporal Antialiasing*, 2016.

Kar14. Karis, Brian: *High Quality Temporal Supersampling*, 2014.

KN01. Kim, Tae-Yong and Ulrich Neumann: *Opacity Shadow Maps.* Eurographics Rendering Workshop 2001, 2001.

KPHE02. Kniss, Joe, Simon Premoze, Charles Hansen and David Ebert: *Interactive Translucent Volume Rendering and Procedural Modeling.* IEEE Visualization, 2002. VIS 2002., pages 109–116, 2002.

KSS11. Kasyan, Nickolay, Nicolas Schulz and Tiago Sousa: *Secrets of CryENGINE 3 Graphics Technology*, 2011.

KvH84. Kajiya, James T. and Brian P. von Herzen: *Ray Tracing Volume Densities.* SIGGRAPH Comput. Graph., 18(3):165–174, 1984.

LG18. Lagarde, Sebastien and Evgenii Golubev: *The Road Toward Unified Rendering with Unity's High Definition Render Pipeline*, 2018.

Lum17. Lumberyard, Amazon: *Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)*, 2017.

LV00. Lokovic, Tom and Eric Veach: *Deep Shadow Maps.* SIGGRAPH 2000 Proceedings (August 2000), 2000:385–392, 2000.

Mit08. Mitchell, Kenny: *Volumetric Light Scattering as a Post-Process.* In Nguyen, Hubert (editor): *GPU Gems 3*, pages 275–285. Addison-Wesley, 2008.

MSS$^+$10. Meyer, Quirin, Jochen Süssmuth, Gerd Sussner, Marc Stamminger and Günther Greiner: *On Floating-Point Normal Vectors.* Computer Graphics Forum, 29(4):1405–1409, 2010.

OBA12. OLSSON, OLA, MARKUS BILLETER and ULF ASSARSSON: *Clustered Deferred and Forward Shading.* In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 87–96, Goslar, DEU, 2012. Eurographics Association.

Per12. PERSSON, TOBIAS: *Practical Particle Lighting*, 2012.

Per15. PERSSON, EMIL: *Practical Clustered Shading*, 2015.

Pet16. PETERS, CHRISTOPH: *Free blue noise textures*, 2016.

PHJ17. PHARR, MATT, GREG HUMPHREYS and WENZEL JAKOB: *Physically Based Rendering: From Theory to Implementation.* The Morgan Kaufmann series in interactive 3D technology. Morgan Kaufmann, Cambridge, MA, Third edition edition, 2017.

Sal16. SALVI, MARCO: *An Excursion in Temporal Supersampling*, 2016.

SG16. SOUSA, TIAGO and JEAN GEFFROY: *The devil is in the details: idTech 666*, 2016.

SVLL10. SALVI, MARCO, KIRIL VIDIMČE, ANDREW LAURITZEN and AARON LEFOHN: *Adaptive Volumetric Shadow Maps.* Computer Graphics Forum, 29(4):1289–1296, 2010.

SWR13. SOUSA, TIAGO, CARSTEN WENZEL and CHRIS RAINE: *The Rendering Technologies of Crysis 3*, 2013.

TU09. TÓTH, BALÁZS and TAMÁS UMENHOFFER: *Real-time Volumetric Lighting In Participating Media.* EUROGRAPHICS 2009, 2009.

Val14a. VALIENT, MICHAL: *Reflections and Volumetrics of Killzone Shadow Fall*, 2014.

Val14b. VALIENT, MICHAL: *Taking Killzone Shadow Fall Image Quality into the Next Generation*, 2014.

Wen06. WENZEL, CARSTEN: *Real-time Atmospheric Effects in Games*, 2006.

Wen07. WENZEL, CARSTEN: *Real-time Atmospheric Effects in Games Revisited*, 2007.

Wih17. WIHLIDAL, GRAHAM: *4K Checkerboard in Battlefield 1 and Mass Effect Andromeda*, 2017.

Wro14. WROŃSKI, BARTŁOMIEJ: *Volumetric Fog: Unified compute shader based solution to atmospheric scattering*, 2014.

Xu16. XU, KE: *Temporal Antialiasing In Uncharted 4*, 2016.

YK08. YUKSEL, CEM and JOHN KEYSER: *Deep Opacity Maps.* Computer Graphics Forum, 27(2):675–680, 2008.

A

Erklärung der Kandidatin / des Kandidaten

☐ Die Arbeit habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

☐ Die Arbeit wurde als Gruppenarbeit angefertigt. Meine eigene Leistung ist ...

Diesen Teil habe ich selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Namen der Mitverfasser: ...

Datum	Unterschrift der Kandidatin / des Kandidaten