FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Iser

Real-Time Light Transport in Analytically Integrable Participating Media

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Oskár Elek, Ph.D.
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2017

I would like to thank my supervisor, Mgr. Oskár Elek, Ph.D., for all the valuable advice he offered when I was working on this thesis. Furthermore, I thank my dear family and friends for their support in my studies.

Title: Real-Time Light Transport in Analytically Integrable Participating Media

Author: Tomáš Iser

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Oskár Elek, Ph.D., Department of Software and Computer Science Education

Abstract: The focus of this thesis is the real-time rendering of participating media, such as fog. This is an important problem, because such media significantly influence the appearance of the rendered scene. It is also a challenging one, because its physically correct solution involves a costly simulation of a very large number of light-particle interactions, especially when considering multiple scattering. The existing real-time approaches are mostly based on empirical or single-scattering approximations, or only consider homogeneous media. This work briefly examines the existing solutions and then presents an improved method for real-time multiple scattering in quasi-heterogeneous media. We use analytically integrable density functions and efficient MIP map filtering with several techniques to minimize the inherent visual artifacts. The solution has been implemented and evaluated in a combined CPU/GPU prototype application. The resulting highly-parallel method achieves good visual fidelity and has a stable computation time of only a few milliseconds per frame.

Keywords: real-time rendering, light transport, participating media

Contents

Introduction

In computer graphics, one of our major topics is generating physically plausible images. We usually refer to this process as photorealistic rendering. It can be used in architectural or design visualizations, simulators, video games, or movies. We often strive to create imagery that resembles reality as we perceive it.

To achieve photorealistic results, we want to understand and simulate how light interacts with the matter around us. When rendering simple scenes, we can sometimes simplify the physical processes by assuming that light travels in a vacuum. But once we want to render more advanced optical phenomena, such as light scattering, we can no longer rely on these simplifications.

Light scattering is a process where the otherwise straight trajectory of the light can be deviated, which happens when light travels through air, water, milk, and other non-metallic participating media [Elek et al., 2013]. Numerous optical phenomena, such as the blurring of objects and lights in a foggy weather, can be explained by the scattering.

This thesis aims at rendering scenes where light has to travel through a participating medium, because the scene, including the camera, is surrounded by the medium. We propose and implement a method that is viable for *real-time rendering* of the participating effects. That means that our algorithm should be able to run at very high frequencies.

Video games typically aim at hundreds of rendered *frames per second* (FPS), but thorough this thesis, our formal requirement is at least 25 Hz [Akenine-Möller et al., 2008], i.e., rendering of one frame should take at most 40 ms. As we will see in Chapter 5, our implementation reaches even higher frequencies and proves that our method can indeed be used in real-time and interactive software.

As we are limited to real-time rendering, our ambition is not to simulate all optical phenomena in all existing media. In this thesis, we only assume multiple scattering and absorption effects in scenes with a single analytically integrable medium. Despite the proposed limitation, there remains a non-trivial motivation.

Motivation

Light scattering can be observed in various real life situations: during foggy weather, sandstorms, when looking at a mist above a lake, or even when swimming *in* the lake itself. When light scattering occurs, we may notice that objects and light sources around us are blurred and our vision gets highly limited. Sometimes, color shifts can also occur because of medium absorption. Water, for example, absorbs blue light less than other colors [Braun and Smirnov, 1993], which explains why objects submerged in water appear bluish.

Whenever we need to visualize an environment with a participating medium—such as the foggy streets in Figure 1—we need an algorithm to simulate the light transport. In case of non-real-time rendering, such as in movies, there exist precise but usually slow methods. Efficient Monte Carlo methods are described, for example, by Jarosz [2008], but the rendering may take several minutes, hours, or even days. Furthermore, even a slight adjustment of the medium parameters can significantly change the rendering time.

In case we want our visualization to run in real time, such as in video games or simulations, we need very efficient approximations. Current real-time software typically uses empirical or single-scattering approximations (Chapter 2) [Mitchell, 2007, Persson, 2012, Wronski, 2014] that cannot correctly blur the scenes.

Correct blurring of light requires taking multiple scattering into account. Multiple-scattering solutions (Chapter 3) can handle fast rendering of homogeneous media [Elek et al., 2013], but methods for non-homogeneous media may still take hundreds of milliseconds even for small resolutions Shinya et al. [2016].

This thesis aims at proposing and fully implementing a real-time method for multiple-scattering effects even in complex scenes with *quasi-heterogeneous media* whose density functions can be analytically integrated.

Use cases

Let us present a list of possible use cases of real-time rendering of participating media. The list is in no way exhaustive but provides several examples of software applications that could benefit from an efficient approximation of light scattering.

Driving simulators Driving in different weather conditions can be very demanding and dangerous. Rain, snow, ice, or fog may affect how a car behaves on a road and how limited our visibility is when driving. Simulating these conditions is therefore desired by many driving and racing simulators. Rendering light scattering in real time is necessary to correctly visualize what a driver can see during a foggy weather.

Military simulators Military simulation is a very broad term. Depending on what we need to visualize, we may benefit from real-time participating media rendering. Simulating underwater environments, for example, can be useful for navy and submarine simulators. Visualizing heavy fogs, sandstorms, or blizzards may be used when simulating navigation in battlefields.

Video games Video games often strive for visual attractiveness. A participating medium can dramatically change how a scene looks like and may alter the depth perception. Even a simple scene with a few objects and light sources may look differently and more interesting when a fog is present (consider Figure 1 that depicts a rather simple scene). Depending on the video game

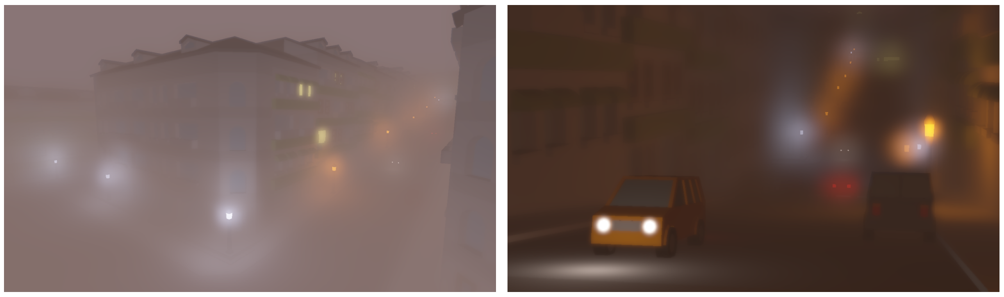


Figure 1: Foggy streets rendered with our method with different parameters.

environment, we may want to render fog, smoke, snow blizzards, or even non-realistic media such as in science-fiction video games.

Goals and structure

This thesis aims to reach the following objectives.

1. **Background** We briefly examine the physical and mathematical background necessary for the correct understanding of light transport. The mostly physical background is presented in Chapter 1. After reading the information, it should be clear how light behaves in participating media. We also present an introduction to image filtering as it will be necessary further in the thesis.

2. **Related works** Before introducing our own method, we briefly review some of the current approaches for real-time rendering of the phenomenon. For this purpose, we introduce two chapters. In Chapter 2, we explain the existing solutions that are either empirical or based on single-scattering approximations. This mainly enables us to understand how modern video games handle the rendering of participating media. In Chapter 3, we have a look at more precise solutions based on multiple scattering.

3. **Proposed method** Based on our examination of the related work, we choose one of the methods as our baseline. Then we propose an improved method with support for quasi-heterogeneous analytically integrable media (such as a fog with a density exponential with regards to altitude). Our new method also support scenes with intensely emissive materials, e.g., night scenes with various light sources such as lanterns and car lamps. After explaining our approach, we analyze the advantages and limitations of our method, also in the context of the competing approaches. Chapter 4 is completely devoted to our proposed method.

4. **Implementation** We prepare a 3D scene containing the mentioned conditions and then fully implement our method in a 3D demo application. The application allows free navigation around the 3D scene and a sufficient freedom in modifying the parameters of the participating medium. We verify that our method is indeed capable of being executed in real time. The explanation of our implementation and analysis of its performance are presented in Chapter 5.

Expectations from the reader

In this thesis, we expect the reader to be already familiar with at least the basic concepts of programming and hardware accelerated real-time rendering. A sufficient introduction into the GPU computing pipelines and available APIs, such as OpenGL or DirectX, would be beyond the scope of the thesis.

We also expect the reader to understand the standard mathematical operations, especially in linear algebra and calculus. On the other hand, detailed knowledge of the mathematics and physics behind light transport and image filtering is not required because the relevant bits will be reviewed in Chapter 1.

1. Physical and mathematical background

When rendering virtual scenes, it is important to understand the physics and mathematics behind the process. Rendering is based on generating an image that should represent what a virtual camera would see when placed in the scene.

The camera is basically a light sensor, so we need to calculate the power of the light that reaches the camera from certain directions. We assume that light is emitted from light sources, then it interacts with elements in the scene, and finally it reaches the camera. Based on this idea, we can construct the *rendering equation*, which mathematically describes the light transport.

In Section 1.1, we very briefly derive how the rendering equation looks like for light transport in a vacuum. It was originally presented by Kajiya [1986] and more interesting information can be found in his article.

Then, we show how the light transport changes when assuming that light interactions occur not only at surfaces but also in the surrounding medium. We briefly explain how light can interact with the participating media (Section 1.2). Based on this knowledge, we derive the *volume rendering equation* (VRE, Section 1.3) as described, among others, by Elek [2016].

After explaining the light transport, a brief introduction to image filtering is presented in Section 1.4. The basics of image manipulation, especially with regards to distribution functions, will be required further in the thesis, primarily in Chapters 3 and 4.

At this point, I would like to clarify the mathematical notation that is used in the remainder of the thesis. The slanted font denotes scalar quantities (e.g., irradiance E), bold font denotes vectors (e.g., position $\mathbf{x}$, direction $\boldsymbol{\omega}$), and sans font denotes discretized functions (e.g., image L). Units of quantities are enclosed in brackets (e.g., $[\mathrm{W} \cdot \mathrm{m}^{-2}]$). The exponential function is always denoted by $\exp(x)$ instead of e^x, because we work with very big exponents and their font would be too small. All other symbols are always explained in the relevant parts of the thesis.

1.1 Light transport in vacuum

Before taking participating media into account, let us explain the light transport in a vacuum. In this section, we derive the rendering equation while assuming that all light interactions happen only when light hits surfaces of objects.

1.1.1 Assumptions

As described by Jarosz [2008] and Elek [2016], computer graphics typically rely on geometric optics (also called ray optics). Even though there also exist wave optics, electromagnetic optics, and quantum optics, they typically provide too low-level descriptions of optical phenomena. In geometric optics, we assume that light travels in straight lines at an infinite speed and that light can only be emitted, reflected, and transmitted.

Because of these simplifications, it is not possible to correctly explain certain effects, such as diffraction and interference [Elek, 2016]. Fortunately, as we will see in the following sections, light transport in participating media can be simulated by geometric optics to the necessary degree.

1.1.2 Radiometric quantities

The rendering equation is based on radiometric quantities. They enable us to objectively describe electromagnetic radiation, including light. Before introducing the full rendering equation, we should first understand the following quantities: flux, irradiance, and radiance. Figure 1.1 illustrates their intuitive meanings.

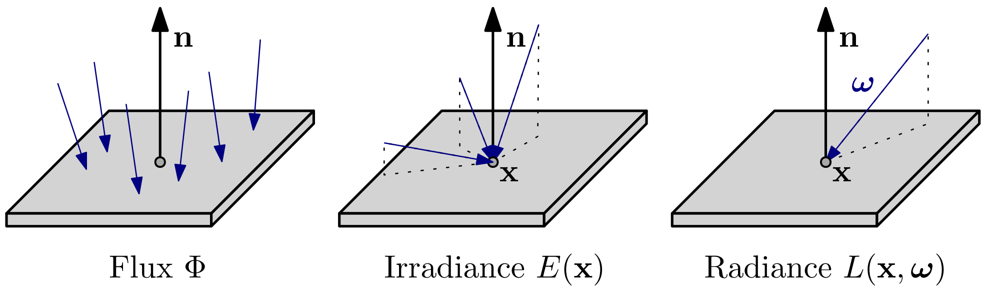

n
n
x
n
ω
x
Flux Φ
Irradiance $E(\mathbf{x})$
Radiance $L(\mathbf{x}, \boldsymbol{\omega})$

Figure 1.1: Illustration of the intuitive meanings of the radiometric quantities. The blue arrows symbolize the arriving light and the vector **n** is the normal of the surface.

Radiant **flux** (also called radiant power) expresses the amount of flowing energy [J] over time [s]. Therefore, the unit of flux is $[\mathrm{J} \cdot \mathrm{s}^{-1}] = [\mathrm{W}]$. In practice, it can be used to express power of light sources. We denote flux by Φ.

To express the amount of radiant power [W] over a certain surface $[\mathrm{m}^2]$, we use **irradiance**. Its unit is $[\mathrm{W} \cdot \mathrm{m}^{-2}]$, and we denote it by E. The irradiance at the surface position $\mathbf{x}$ is denoted by $E(\mathbf{x})$.

The radiometric quantity that we use in the rendering equation is called **radiance**. It expresses the amount of radiant power [W] that a sensor receives from a surface $[\mathrm{m}^2]$ from a certain direction [sr]. Radiance therefore has the unit $[\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}]$. For the position $\mathbf{x}$ and angle $\boldsymbol{\omega}$, we denote it by $L(\mathbf{x}, \boldsymbol{\omega})$.

Relations

We have defined the quantities individually, but they are related to each other as can be understood from Figure 1.1. Irradiance, for example, can be expressed by integrating radiance. The integral needs to be over all directions in the upper hemisphere Ω^+ in respect to the normal vector $\mathbf{n}$ of the surface. Flux can also be expressed by integrating radiance. We need to integrate over the whole surface area A and over all directions. We can therefore write [Elek, 2016]:

$$E(\mathbf{x}) = \int_{\Omega^+} L(\mathbf{x}, \boldsymbol{\omega}) \, \mathrm{d}\boldsymbol{\omega},$$

$$\Phi = \int_A \int_{\Omega^+} L(\mathbf{x}, \boldsymbol{\omega}) \, \mathrm{d}\boldsymbol{\omega} \, \mathrm{d}\mathbf{x}.$$

Please note that the radiance L is used for light in both directions. The *incident radiance* refers to the incoming light and the *exitant radiance* refers to the outgoing light. For the sake of simplicity, we will not use any special notation to differ between the two situations. It should always be clear from the context and from the orientations of the vectors. If $\boldsymbol{\omega}$ points towards a surface, it indicates the incoming direction, and vice versa.

1.1.3 Bi-directional reflectance distribution function

When light travels through a vacuum, it does not interact with anything until it hits a surface. Therefore, the radiance remains constant along straight lines between light sources, surfaces, and cameras. An interaction with a surface can result in light being reflected, refracted, or absorbed. When the light is reflected from an opaque surface, we can describe the reflection by the *bi-directional reflectance distribution function* (BRDF) of the surface material.

Let r denote a certain BRDF. For the surface position $\mathbf{x}$ with incoming light from the direction $\boldsymbol{\omega}'$, the function $r(\mathbf{x}, \boldsymbol{\omega}', \boldsymbol{\omega})$ gives us the ratio of the reflected radiance along direction $\boldsymbol{\omega}$. We can define the BRDF as [Jarosz, 2008]:

$$r(\mathbf{x}, \boldsymbol{\omega}', \boldsymbol{\omega}) = \frac{\mathrm{d}L(\mathbf{x}, \boldsymbol{\omega})}{L(\mathbf{x}, \boldsymbol{\omega}')\langle \mathbf{n}, -\boldsymbol{\omega}' \rangle \,\mathrm{d}\boldsymbol{\omega}'}, \tag{1.1}$$

where $\langle \mathbf{n}, -\boldsymbol{\omega}' \rangle$ denotes the inner product of the normal vector and the direction *towards* the light source (Figure 1.2).

We can now use r to describe the total radiance reflected by the surface from all directions. We need to multiply both sides of Equation 1.1 by the denominator and integrate over all incoming directions $\boldsymbol{\omega}'$ from the upper hemisphere. The total reflected radiance L_r can therefore be computed as:

$$\underbrace{L_\mathrm{r}(\mathbf{x}, \boldsymbol{\omega})}_{\text{reflected}} = \int_{\Omega^+} \underbrace{r(\mathbf{x}, \boldsymbol{\omega}', \boldsymbol{\omega})}_{\text{BRDF}} \underbrace{L(\mathbf{x}, \boldsymbol{\omega}')\langle \mathbf{n}, -\boldsymbol{\omega}' \rangle}_{\text{incoming}} \,\mathrm{d}\boldsymbol{\omega}'. \tag{1.2}$$

As we can see, the BRDF is an important function defining how light interacts with a given surface. There are many different reflectance models for different types of materials. For an overview of common BRDF, refer to the work by Montes and Ureña [2012]. It should be noted that the models can be generally classified into three categories: physical-based, empirical, and experimental.

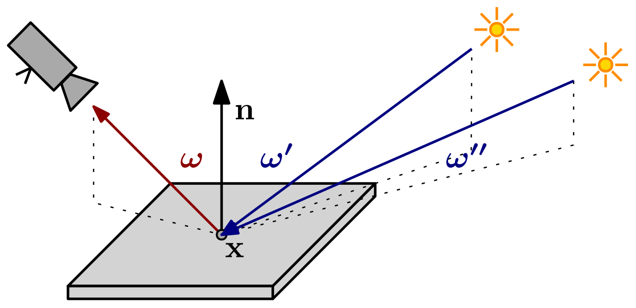


Figure 1.2: The BRDF tells us how light is reflected from an opaque surface.

A very simple empirical reflectance model—still very popular in real-time graphics for its simplicity—is the BRDF by Phong. It can be expressed as:

$$r_{\text{Phong}}(\mathbf{x}, \boldsymbol{\omega}', \boldsymbol{\omega}) = \frac{k_{\text{d}}}{\pi} + k_{\text{s}} \frac{\langle \boldsymbol{\omega}, \boldsymbol{\omega}'_{\text{r}} \rangle^n}{4\pi \langle \mathbf{n}, -\boldsymbol{\omega}' \rangle}, \tag{1.3}$$

where $\boldsymbol{\omega}'_{\text{r}}$ denotes Phong's reflection vector, and k_{d}, k_{s}, n are material parameters. For its simplicity, we use this model in our prototype application (Chapter 5).

1.1.4 Rendering equation in vacuum

Now that we understand the basic concepts, we can present the rendering equation of Kajiya [1986]. It generalizes rendering algorithms by describing the balance in the energy flow between surfaces. The main idea is to express the outgoing radiance L_{s} from a surface at the position $\mathbf{x}$ in the direction $\boldsymbol{\omega}$ as a sum of the emitted radiance L_{e} and the reflected radiance L_{r}:

$$L_{\text{s}}(\mathbf{x}, \boldsymbol{\omega}) = L_{\text{e}}(\mathbf{x}, \boldsymbol{\omega}) + L_{\text{r}}(\mathbf{x}, \boldsymbol{\omega}). \tag{1.4}$$

By substituting Equation 1.2 into Equation 1.4, we get the hemispherical form of Kajiya's rendering equation:

$$\underbrace{L_{\text{s}}(\mathbf{x}, \boldsymbol{\omega})}_{\text{outgoing}} = \underbrace{L_{\text{e}}(\mathbf{x}, \boldsymbol{\omega})}_{\text{emitted}} + \underbrace{\int_{\Omega^+} \underbrace{r(\mathbf{x}, \boldsymbol{\omega}', \boldsymbol{\omega})}_{\text{BRDF}} \underbrace{L(\mathbf{x}, \boldsymbol{\omega}')\langle \mathbf{n}, -\boldsymbol{\omega}' \rangle}_{\text{incoming}} \, \mathrm{d}\boldsymbol{\omega}'}_{\text{total reflected}} . \tag{1.5}$$

The rendering equation then has to be solved by a rendering algorithm. Because of the complexity of the equation, we need a sophisticated numerical integration method. An explanation of some of these would be far beyond the scope of this thesis. Solutions such as *finite element methods*, *Monte Carlo ray tracing methods*, and *irradiance caching* are described in the dissertation of Jarosz [2008].

1.1.5 Real-time applications

When solving the rendering equation in real-time, we are limited by the execution time and cannot rely on precise but slow methods. The equation has to be simplified. One of the possible solutions is based on approximating the indirect light in our scene. We can replace all indirect light by a single constant *ambient term* L_{a}. It enables us to replace the integral in Equation 1.5 by the finite sum of all light sources in our scene:

$$L_{\text{s}}(\mathbf{x}, \boldsymbol{\omega}) = L_{\text{a}} + L_{\text{e}}(\mathbf{x}, \boldsymbol{\omega}) + \sum_{\substack{(L_i, \boldsymbol{\omega}'_i) \\ \text{light sources}}} r(\mathbf{x}, \boldsymbol{\omega}'_i, \boldsymbol{\omega}) L_i(\mathbf{x}, \boldsymbol{\omega}'_i) \langle \mathbf{n}, -\boldsymbol{\omega}'_i \rangle. \tag{1.6}$$

The simplified Equation 1.6 can now be solved in real-time on a GPU. Technically, the light transport can be solved for each pixel individually by simply iterating through all light sources. This is usually done on the GPU in a *fragment shader* (*pixel shader*) that is executed for each pixel. Later in Chapter 5, we implement a lighting shader based exactly on this simplification.

1.2 Interactions in participating media

So far, we have assumed that light is emitted from light sources, then interacts with surfaces in the scene, and finally reaches the camera. It enabled us to derive the rendering equation for light transport in a vacuum. The goal of this thesis, however, is to simulate participating media.

Before introducing the volumetric rendering equation in Section 1.3, let us first have a look at how light interacts with the media. This section is a brief summary and more information can be found in the dissertations by Jarosz [2008, Chapter 4] and Elek [2016, Chapter 2].

1.2.1 Assumptions

When rendering the participating media, we have to make certain assumptions about their properties. We will model a medium as a collection of identical, randomly positioned, and small particles, such as water droplets or dust. The light interactions will be described for individual particles and then extended to the whole medium.

For the position $\mathbf{x}$ in a certain medium, let $\varrho(\mathbf{x})\,[\mathrm{m}^{-3}]$ denote the density of the particles at that location. Each particle is characterized by three properties: absorption cross-section $C_\mathrm{a}\,[\mathrm{m}^2]$, scattering cross-section $C_\mathrm{s}\,[\mathrm{m}^2]$, and a phase function f_p. Both C_a and C_s are wavelength-dependent[1] , which will be implicit throughout the rest of the thesis.

1.2.2 Types of interactions

When describing the interactions between the light and the particles, we understand light as a stream of photons. The photons can arrive to the medium directly from light sources or after being reflected by surfaces in the scene. Furthermore, the medium itself can emit a new photon by converting other forms of energy.

When a photon travelling through the medium hits a particle, the photon itself can be completely absorbed or scattered in a new direction. Therefore, when simulating the media, we consider three interactions: absorption, scattering, and emission (Figure 1.3).

Absorption

When an interaction happens, the photon may be absorbed by the particle it hit. Because energy can never be lost, what actually happens is that the photon energy is transformed into another form. It may cause, for example, a raise of the temperature of the medium.

We are, however, only interested in light transport, hence we consider the energy to be "lost". Macroscopically, the "lost energy" can be observed as the overall intensity of the light decreases.

[1] Wavelength-dependent properties may have different values for different wavelengths of the light. In computer graphics, we typically model the properties for the red, green, and blue channels. To avoid cluttered notation, we will not make any difference between these properties and regular properties.

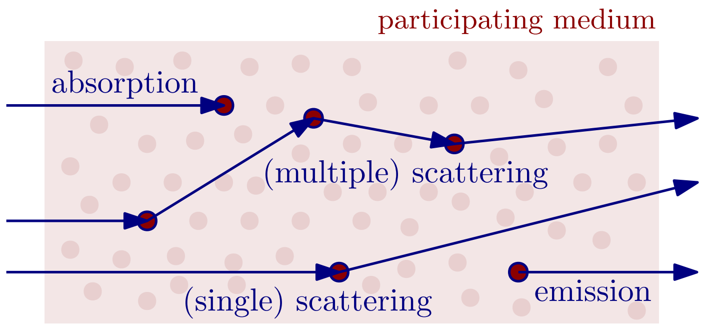


Figure 1.3: Light interactions in a participating medium.

How much the medium absorbs light is determined by the *absorption coefficient* $\sigma_\mathrm{a}\,[\mathrm{m}^{-1}]$. This property depends on the absorption cross-section of the particles and the density of these particles:

$$\sigma_\mathrm{a}(\mathbf{x}) = C_\mathrm{a} \cdot \varrho(\mathbf{x}).$$

Imagine a light beam passing through a medium. Let $\mathbf{x}$ denote a location in the beam and let Δt be a small step in the direction $\boldsymbol{\omega}$ of the beam. Then $\sigma_\mathrm{a}(\mathbf{x} + t\boldsymbol{\omega})\Delta t$ is the number of photons absorbed by the medium.

Analogically, we can compute how long, on the average, a single photon travels until it is absorbed by the medium. The value equals $1/\sigma_\mathrm{a}\,[\mathrm{m}]$.

Scattering

If a photon is not absorbed when it hits a particle, its energy is scattered into a new direction. The direction depends on the phase function of the medium.

For a beam of light passing through a medium (Figure 1.4), we can divide the scattering into two opposite but complementing processes. **Out-scattering** is when photons of the beam are scattered *out* of the path of the beam, reducing its radiance. **In-scattering** describes the opposite effect: when other photons are scattered and converge *into* the path of the beam, increasing its radiance.

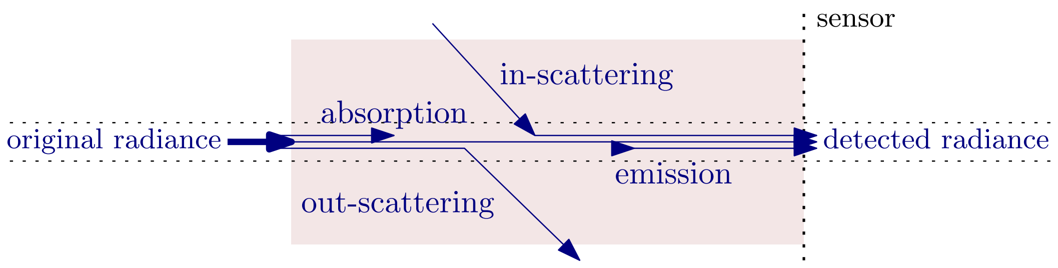


Figure 1.4: Light interactions in the perspective of a light beam.

Furthermore, we can distinguish between *single scattering*, when we assume that light can only scatter once, and *multiple scattering*, when light can scatter multiple times. In real media, multiple scattering occurs, which may cause significant *spatial and angular spreading* (Figure 1.5) [Premože et al., 2004]. This is the reason why objects in fog appear blurred: the light that was reflected from them is smoothly spread around and colors blend together.

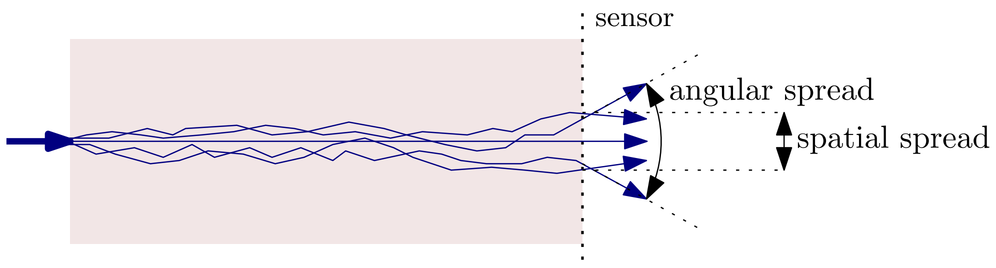


Figure 1.5: Beam spreading caused by multiple scattering.

How much the scattering happens is described by the *scattering coefficient* $\sigma_\mathrm{s}\,[\mathrm{m}^{-1}]$. It depends on the scattering cross-section of the particles and the density of these particles:

$$\sigma_\mathrm{s}(\mathbf{x}) = C_\mathrm{s} \cdot \varrho(\mathbf{x}).$$

Analogically to the absorption, how long a photon travels, on the average, until it is scattered in the medium, equals $1/\sigma_\mathrm{s}\,[\mathrm{m}]$.

Emission

When particles in the medium convert other forms of energy into light, a new photon is emitted from inside the medium. Let us denote the emitted radiance the same way we did in the rendering equation: $L_\mathrm{e}(\mathbf{x}, \boldsymbol{\omega})$.

1.2.3 Extinction coefficient and optical thickness

Extinction describes the event when a photon is either absorbed or scattered. It is described analogically to the absorption and scattering. We define the *extinction coefficient* as $\sigma_\mathrm{t}\,[\mathrm{m}^{-1}] = \sigma_\mathrm{a} + \sigma_\mathrm{s}$.

How long a photon can travel, on the average, until it is absorbed or scattered in the medium, equals $1/\sigma_\mathrm{t}\,[\mathrm{m}]$. This value is called the *mean free path* of a photon in a medium. As we can see, σ_t is the inverse to the mean free path.

The extinction defines both absorption and scattering together, but we may be interested in the probability of a photon being scattered *instead of* absorbed. This probability is called the *scattering albedo*. We denote it by α and it is defined as:

$$\alpha = \frac{\sigma_\mathrm{s}}{\sigma_\mathrm{a} + \sigma_\mathrm{s}} = \frac{\sigma_\mathrm{s}}{\sigma_\mathrm{t}}.$$

By integrating the extinction along a line segment l, we get the variable called **optical thickness** or *optical depth*, denoted by τ:

$$\tau(l) = \int_l \sigma_\mathrm{t}(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{1.7}$$

1.2.4 Beer–Lambert–Bouguer law

In Section 1.1.2, we defined a radiometric quantity called radiance that was later used in the rendering equation for a vacuum (Equation 1.5). We now explain how radiance relates to light interactions in participating media.

As we have already explained, when photons in a light beam travel through a medium, the radiance along the beam may decrease because of the extinction. How exactly the radiance changes is described by the *(beam) transmittance*, denoted by T.

Let l denote the straight optical path between locations $\mathbf{x_0}$ and $\mathbf{x_1}$. We can use the transmittance $T(\mathbf{x_0}, \mathbf{x_1}) = T(l)$ to express the radiance after being reduced by interacting with the medium (Figure 1.6):

$$L(\mathbf{x_1}, \boldsymbol{\omega}) = T(l) \cdot L(\mathbf{x_0}, \boldsymbol{\omega}).$$

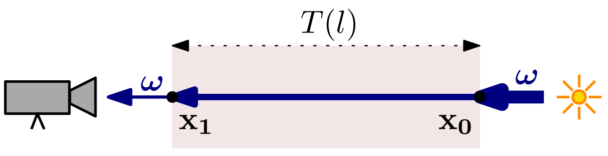


Figure 1.6: Radiance may significantly decrease after passing through a medium.

We can see that for a constant transmittance $T = 1$, we get an equation describing how radiance behaves between surfaces in a vacuum.

It only remains to describe how the transmittance T is related to the parameters we have introduced in the previous subsections. The *Beer–Lambert–Bouguer law* explains exactly what we need by stating how to compute the transmittance with regards to the optical thickness τ:

$$T(l) = \exp\left(-\tau(l)\right). \tag{1.8}$$

1.2.5 Phase function

When light scatters after interacting with particles, the light is distributed in other directions. The *phase function* of the medium describes the angular distribution of the scattering. It can be understood similarly to BRDF (Equation 1.1), but instead of describing how light is reflected from a surface, it tells us how it is scattered from a particle.

We will denote the phase function by f_p and define its parameters analogically to our BRDF. For the position $\mathbf{x}$, incoming direction $\boldsymbol{\omega}'$, and outgoing direction $\boldsymbol{\omega}$, the phase function is $f_\mathrm{p}(\mathbf{x}, \boldsymbol{\omega}', \boldsymbol{\omega})$. Because this thesis assumes that all particles in a medium have the same properties, we can only write $f_\mathrm{p}(\boldsymbol{\omega}', \boldsymbol{\omega})$.

How exactly we model the phase function and what the requirements for such function are can be found in the work by Elek [2016, Section 2.3.2]. In our real-time rendering method, we do not need to understand the details. It may be interesting to note, though, that every phase function is also a proper spherical probability density function.

Scattering asymmetry

We will further assume that all particles in the simulated medium have random, uncorrelated orientations. As described by Elek [2016], we can then express the phase function only as $f_\mathrm{p}(\theta)$, where θ is a scattering angle.

As the function only depends on a single angle, it would be useful if we could define a single quantity describing the shape of the function. For this purpose, we define the *scattering asymmetry factor*, denoted by g. We can express it as the average cosine on the scattering angle θ:

$$g = \int_{\Omega_{4\pi}} f_{\mathrm{p}}(\theta) \cos\theta \,\mathrm{d}\boldsymbol{\omega}', \tag{1.9}$$

where $\Omega_{4\pi}$ is the whole sphere of directions.

Because g is an average cosine, its values are $\in [-1, 1]$. For $g > 0$, scattering in forward directions is favored. For negative values, backscattering is favored.

1.3 Volume rendering equation

Our goal in this section is to derive the equation for radiance in participating media. For this purpose, we use the already derived rendering equation for a vacuum (Equation 1.5) and extend it by assuming the light interactions we described in the previous section (Section 1.2).

1.3.1 Radiative transport equation

As we have already explained, for a fixed light ray passing through a participating medium, there are four types of interactions that can change its radiance. Two of these interactions can increase the radiance, they are *emission* and *in-scattering.* The other two can decrease the radiance, they are *absorption* and *out-scattering.*

Let $\mathbf{x}$ denote a location and $\boldsymbol{\omega}$ the direction of a light ray. We can define the differential change of radiance along $\boldsymbol{\omega}$ at $\mathbf{x}$ as:

$$\begin{aligned}(\boldsymbol{\omega} \cdot \nabla) L(\mathbf{x}, \boldsymbol{\omega}) = {} & \underbrace{L_{\mathrm{e}}(\mathbf{x}, \boldsymbol{\omega})}_{\text{emission}} + \underbrace{\sigma_{\mathrm{s}}(\mathbf{x}) L_{\mathrm{i}}(\mathbf{x}, \boldsymbol{\omega})}_{\text{in-scattering}} && \text{(volume contribution)} \\ & - \underbrace{\sigma_{\mathrm{a}}(\mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega})}_{\text{absorption}} - \underbrace{\sigma_{\mathrm{s}}(\mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega})}_{\text{out-scattering}} && \text{(extinction).}\end{aligned} \tag{1.10}$$

Equation 1.10 is called the *radiative transport equation* (RTE) and was described by Chandrasekhar [1960]. It explains how exactly the radiance is affected by the four types of interactions. Before proceeding, we have to further explain the terms of the equation, especially because we still have not defined what L_{i} stands for.

First of all, we can notice that the absorption and out-scattering together can be expressed as a single term. We have already defined the extinction coefficient (Section 1.2.3) that enables us to write:

$$\sigma_{\mathrm{a}}(\mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega}) + \sigma_{\mathrm{s}}(\mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega}) = \sigma_{\mathrm{t}}(\mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega}). \tag{1.11}$$

The emission and in-scattering together can be expressed by a single term as well. We define the *volume contribution* L_{v} as the sum of the emission and the in-scattering. We can now write:

$$L_{\mathrm{e}}(\mathbf{x}, \boldsymbol{\omega}) + \sigma_{\mathrm{s}}(\mathbf{x}) L_{\mathrm{i}}(\mathbf{x}, \boldsymbol{\omega}) = L_{\mathrm{v}}(\mathbf{x}, \boldsymbol{\omega}). \tag{1.12}$$

Now we have to explain the radiance L_{i}. We want to express the radiance that is scattered *into* the light ray from all other directions. By integrating over the directions while using the phase function f_{p} (Section 1.2.5), we get:

$$L_{\mathrm{i}}(\mathbf{x}, \boldsymbol{\omega}) = \int_{\Omega_{4\pi}} f_{\mathrm{p}}(\mathbf{x}, \boldsymbol{\omega}', \boldsymbol{\omega}) L(\mathbf{x}, \boldsymbol{\omega}') \, \mathrm{d}\boldsymbol{\omega}'. \tag{1.13}$$

Notice how similar Equation 1.13 is to Equation 1.2 with the BRDF. Now, by substituing Equations 1.11 and 1.12 to Equation 1.10, we can express the RTE in a much shorter form:

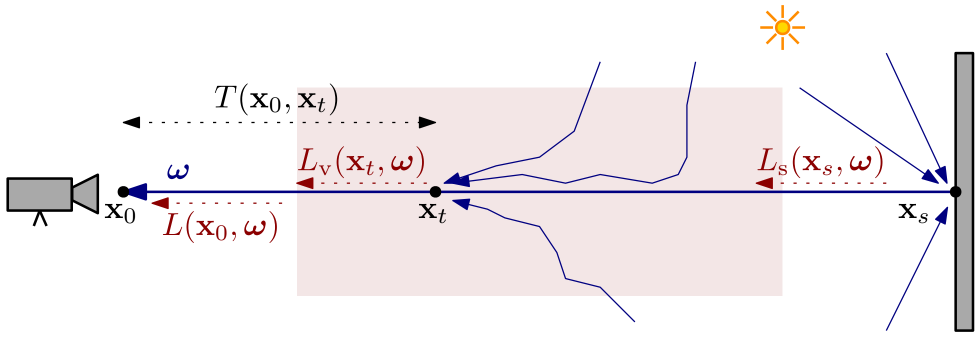
$$(\boldsymbol{\omega} \cdot \nabla) L(\mathbf{x}, \boldsymbol{\omega}) = L_{\mathrm{v}}(\mathbf{x}, \boldsymbol{\omega}) - \sigma_{\mathrm{t}}(\mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega}). \tag{1.14}$$

1.3.2 Rendering equation for participating media

The final goal of this section is to derive the rendering equation for participating media. The original rendering equation for a vacuum (Equation 1.5) defines the surface contribution L_{s}. This contribution has to be reduced (attenuated) according to the transmittance T. On the other hand, the contribution also has to be increased by the volume contribution L_{v}.

The final form of the *volume rendering equation* (VRE) can be obtained by integrating both sides of Equation 1.14 over the optical path between the positions $\mathbf{x}_0$ and $\mathbf{x}_s$ (Figure 1.7) [Elek, 2016]:

$$L(\mathbf{x}_0, \boldsymbol{\omega}) = \underbrace{\int_0^s T(\mathbf{x}_0, \mathbf{x}_t) L_{\mathrm{v}}(\mathbf{x}_t, \boldsymbol{\omega}) \, \mathrm{d}t}_{\text{total volume contribution}} + \underbrace{T(\mathbf{x}_0, \mathbf{x}_s) L_{\mathrm{s}}(\mathbf{x}_s, \boldsymbol{\omega})}_{\text{surface contribution}}. \tag{1.15}$$



$T(\mathbf{x}_0, \mathbf{x}_t)$
$\boldsymbol{\omega}$
$L_{\mathrm{v}}(\mathbf{x}_t, \boldsymbol{\omega})$
$L_{\mathrm{s}}(\mathbf{x}_s, \boldsymbol{\omega})$
$\mathbf{x}_0$
$L(\mathbf{x}_0, \boldsymbol{\omega})$
$\mathbf{x}_t$
$\mathbf{x}_s$

Figure 1.7: Illustration of the intuitive understanding of the VRE.

It now remains to explain how it is possible to solve the equation in real-time applications. We dedicate the rest of this thesis to analyze and implement a solution to this problem. For more details about the methods for non-real-time applications, I will, again, refer you to the work by Jarosz [2008].

1.4 Image filtering

Before introducing a new chapter, let us have a brief look at the basics of image filtering. The filtering is used later in Chapters 3, 4, and 5, but the basic notation we introduce here is also useful for Chapter 2.

1.4.1 Screen-space approaches

Most of the methods described in the following chapters, including our proposed method, are based on *screen-space* approaches. We assume that we can already render our scene in a vacuum using a common rendering approach. Then we *modify the original image* by adding the effects caused by the participating media.

Because we work with the information that are "on the screen", we call these methods "screen-space". They are very popular in real-time rendering on the GPUs [Elek et al., 2013] as they can often be executed for each output pixel individually in parallel. Typically, these approaches tend to be limited as they cannot work with any information that are currently not on the screen, e.g., the objects behind the camera. Fortunately, as we will see later, this limitation is not very critical in certain conditions.

For our purposes, an image is a discrete two-dimensional function $\mathsf{L}\colon \mathbb{N}^2 \to \mathbb{R}^3$. For each *pixel position* $\mathbf{x} \in \mathbb{N}^2$, the function gives us the color of the pixel $\mathsf{L}(\mathbf{x}) \in \mathbb{R}^3$ composed of non-negative real red, green, and blue components. By *high dynamic range* (HDR) images, we understand the images where the ratio between the darkest and the lightest pixel's color can be very high. Therefore, we generally assume that each color channel can have values in $[0, +\infty)$.

1.4.2 Convolution

Many of the techniques for modifying images apply a certain *filter* on the original image. Typically, they are neighborhood-based, which means that the new colors of the pixels are calculated from the initial colors of the neighboring pixels.

Based on this idea, we can use the mathematical operation called *convolution* [Fialka and Čadík, 2006]. The operation is usually denoted by $*$. Let L denote the original image and K the so-called filter kernel. For 2D discrete functions, the convolution can be defined for example as:

$$(\mathsf{L} * \mathsf{K})(\mathbf{x}) = \sum_{\mathbf{x}'} \mathsf{L}(\mathbf{x}')\mathsf{K}(\mathbf{x}' - \mathbf{x}),$$

where the sum iterates over all pixel positions or only the pixels from the neighborhood of $\mathbf{x}$. The filter takes the offsets between the pixels and returns the values describing how important that neighboring pixel's color is for the result.

A simple example is a convolution with a Gaussian kernel, i.e., the discrete kernel based on the continuous Gaussian distribution. It can be used for blurring images and we will come back to it later, especially in Section 3.2.

1.4.3 Point spread function

In computer graphics and physics, we sometimes use the convolution with a special filter called the *point spread function* (PSF) $f_{\mathrm{PSF}}\colon \mathbb{R}^2 \to \mathbb{R}$ [Elek, 2016]. This function describes the spreading of a point signal caused, for example, by an optical system. In our case, we can understand the pixels of an image as point signals, which means that we can convolve an image with a PSF: $\mathsf{L} * f_{\mathrm{PSF}}$ (Figure 1.8). In practice, we can also convolve images with the inverse of a certain PSF, which can be used to fix imperfections of optical systems.

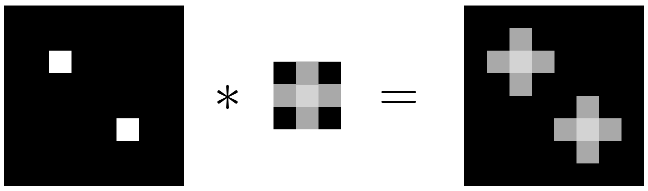


Figure 1.8: Blurring an 8×8 image with a 3×3 point spread function.

1.4.4 Normalization problem

The discrete convolution may suffer from incorrect normalization of the filter. Usually, we want image filtering to conserve energy, i.e., the sum of all colors in the final image should match the sum in the initial image. Especially, we do not want our filter to generate any new energy in the image. Mathematically, we want the sum over all pixel positions in the kernel to equal 1:

$$\sum_{\mathbf{x}} \mathsf{K}(\mathbf{x}) = 1.$$

Unfortunately, if our filtering kernel is based on an originally *continuous* function, such as the Gaussian distribution, then even though the initial distribution was normalized, sampling the function at discrete values will typically not be normalized anymore. Therefore, we cannot simply take any function and use it in the discrete convolution without having to worry about the energy conservation. One of the easiest solutions is to sum the values of the kernel and multiply the whole kernel by the reciprocal value.

1.4.5 Spatially varying kernels

The methods we describe in Chapter 3 are based on image filtering with spatially varying kernels. The definition of convolution assumes that the kernel is the same for all pixels, but we may want each pixel to be distributed differently. For a simple example, imagine that every pixel in the original image has a different PSF (Figure 1.9). The operation for filtering the image would by definition not be a convolution anymore.

We can define new weighting functions $w_{\mathbf{x}'}(\mathbf{x}'-\mathbf{x})$ that would tell us how much $\mathbf{x}'$ contributes to the filtered value of $\mathbf{x}$. We could now rewrite the convolution so that we would get the filtered image L' as:

$$\mathsf{L}'(\mathbf{x}) = \sum_{\mathbf{x}'} w_{\mathbf{x}'}(\mathbf{x}' - \mathbf{x})\mathsf{L}(\mathbf{x}'). \tag{1.16}$$

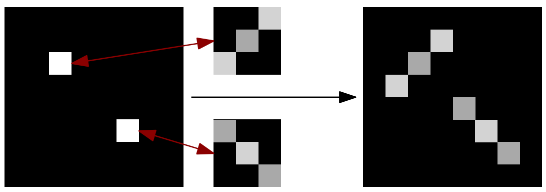


Figure 1.9: Blurring an image with a spatially varying PSF.

1.4.6 Algorithms

For the image size $n \times n$ and kernel size $m \times m$, the time complexity of the convolution algorithms is generally $\mathcal{O}(n^2m^2)$ [Fialka and Čadík, 2006]. Even though it is possible to get lower complexity for separable kernels or using the fast Fourier transform, the ideas are not really important for this thesis. As we have already stated, filtering with spatially varying kernels is not possible with a convolution. Therefore, we need to use alternative algorithms.

Let us now assume that we want to filter an image with spatially varying functions $w_{\mathbf{x}'}$. Furthermore, we assume that these functions are not normalized. Rather, we need to normalize them ourselves during the run of the algorithm. These are the exact conditions that we will be working with later in Chapters 3, 4 and 5, so it makes sense to introduce the algorithms now.

Generally, when assuming these conditions, there are two possible approaches: *splatting* (*distributing*) and *gathering* [Elek et al., 2013, Shinya et al., 2016]. The splatting approach iterates over all pixel positions $\mathbf{x}'$ and adds their distribution to the neighboring pixels. The gathering approach works the other way around: it iterates over all pixel positions $\mathbf{x}$ and adds the distribution of the neighboring pixels to itself. Both approaches are illustrated in Figure 1.10.

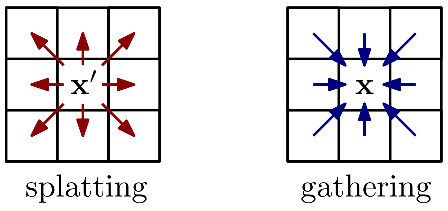


Figure 1.10: Illustration of the splatting and gathering approaches.

Splatting

By splatting, we mean that we take every pixel and we add its contribution to all other pixels, i.e., we "splat" the distributed pixel to the result. This approach is depicted in Algorithm 1.

The time complexity of the algorithm is $\mathcal{O}(n^2m^2)$. The complexity can be theoretically lowered by assuming small neighborhoods, e.g., $\sqrt{n} \times \sqrt{n}$ or $\log n \times \log n$, but it is not generally possible for filters with large support.

Not only is the time complexity high for real-time purposes, the idea of splatting is not very ideal for modern GPUs. Fragment shaders typically allow us to run our algorithms in parallel for each output pixel. It means that we want an algorithm that only writes to a single pixel in each iteration. Obviously, the idea of splatting completely contradicts this requirement.

Gathering

The solution that fits GPUs much better is based on gathering [Elek et al., 2013]. For each pixel, we "gather" the contributions from the neighboring pixels. The idea corresponds to Equation 1.16 and is depicted in Algorithm 2. As we can see, the algorithm only writes to a single pixel in every iteration of the outer loop, so the iterations can run in parallel on the GPU.

Unfortunately, for correct normalization, we need to calculate the $totalWeight$ in an additional inner loop *inside* the inner loop. The time complexity has increased to $\mathcal{O}(n^2m^4)$.

Faster gathering

Because the time complexity of the gathering algorithm is too high, we want to get around the normalization problem. We can remove the inner loop at lines 8–11 and instead compute the weights from pixel locations $\mathbf{x}'$ (Algorithm 3). This way, we get the time complexity $\mathcal{O}(n^2m^2)$.

Unfortunately, this normalization is generally incorrect because we only use one $totalWeight$ for all pixel positions $\mathbf{x}'$. But the weighting functions are spatially varying, which means that their $totalWeight$s may be different.

As explained by Shinya et al. [2016], we can use this approximation to a certain degree by assuming small neighborhoods and similarity of local pixels. Later in Chapter 5, we implement a naive filtering algorithm based on this incorrect normalization. Certain artifacts of our implementation are explained in Section 5.3.7.

Algorithm 1 Splatting in $\mathcal{O}(n^2m^2)$

```
 1: procedure SPLAT($\mathsf{L}$)
 2:     $\mathsf{L}' \leftarrow$ empty image
 3:     for all pixel positions $\mathbf{x}'$ do
 4:         $totalWeight \leftarrow 0$
 5:         $w_{\mathbf{x}'} \leftarrow$ weighting function of $\mathbf{x}'$
 6:         // accumulate the weights at the discrete samples:
 7:         for all pixel positions $\mathbf{x}$ in the neighborhood of $\mathbf{x}'$ do
 8:             $totalWeight \leftarrow totalWeight + w_{\mathbf{x}'}(\mathbf{x}' - \mathbf{x})$
 9:         end for
10:         // distribute according to the normalized weight:
11:         for all pixel positions $\mathbf{x}$ in the neighborhood of $\mathbf{x}'$ do
12:             $weight \leftarrow w_{\mathbf{x}'}(\mathbf{x}' - \mathbf{x})/totalWeight$ // normalization
13:             $\mathsf{L}'(\mathbf{x}) \leftarrow \mathsf{L}'(\mathbf{x}) + weight \cdot \mathsf{L}(\mathbf{x}')$
14:         end for
15:     end for
16:     return $\mathsf{L}'$
17: end procedure
```

Algorithm 2 Correctly normalized gathering in $\mathcal{O}(n^2m^4)$

```
procedure NormalizedGather($\mathsf{L}$)
    $\mathsf{L}' \leftarrow$ empty image
    for all pixel positions $\mathbf{x}$ do
        for all pixel positions $\mathbf{x}'$ in the neighborhood of $\mathbf{x}$ do
            $totalWeight \leftarrow 0$
            $w_{\mathbf{x}'} \leftarrow$ weighting function of $\mathbf{x}'$
            // accumulate the weights at the discrete samples:
            for all pixel positions $\mathbf{x}''$ in the neighborhood of $\mathbf{x}'$ do
                $totalWeight \leftarrow totalWeight + w_{\mathbf{x}'}(\mathbf{x}' - \mathbf{x}'')$
            end for
            // distribute according to the normalized weight:
            $weight \leftarrow w_{\mathbf{x}'}(\mathbf{x}' - \mathbf{x})/totalWeight$ // normalization
            $\mathsf{L}'(\mathbf{x}) \leftarrow \mathsf{L}'(\mathbf{x}) + weight \cdot \mathsf{L}(\mathbf{x}')$
        end for
    end for
    return $\mathsf{L}'$
end procedure
```

Algorithm 3 Incorrectly normalized gathering in $\mathcal{O}(n^2m^2)$

```
procedure FastGather($\mathsf{L}$)
    $\mathsf{L}' \leftarrow$ empty image
    for all pixel positions $\mathbf{x}$ do
        $color \leftarrow \mathbf{0}$
        $totalWeight \leftarrow 0$
        for all pixel positions $\mathbf{x}'$ in the neighborhood of $\mathbf{x}$ do
            $w_{\mathbf{x}'} \leftarrow$ weighting function of $\mathbf{x}'$
            $weight \leftarrow w_{\mathbf{x}'}(\mathbf{x}' - \mathbf{x})$
            $color \leftarrow color + weight \cdot \mathsf{L}(\mathbf{x}')$
            $totalWeight \leftarrow totalWeight + weight$
        end for
        // distribute according to the incorrectly normalized weights:
        $\mathsf{L}'(\mathbf{x}) \leftarrow color/totalWeight$
    end for
    return $\mathsf{L}'$
end procedure
```

2. Empirical and single-scattering methods

After explaining the necessary mathematical and physical background, let us now have a look at several existing real-time solutions to the light transport problem. This step enables us to later present our proposed method (Chapter 4) in the context of the existing approaches.

In this chapter, we describe four existing methods and try to briefly analyze how they work and what their benefits and disadvantages are. As obvious from the name of the chapter, each of the methods is either completely empirical, i.e., not really physically based, or it assumes that light can only scatter once. Furthermore, none of these methods is able to blur all objects in the scene because they only calculate the contribution from designated light sources in the scene. Later in Chapter 3, we introduce screen-space multiple-scattering methods that can blur all pixels.

Despite the limitations of the empirical and single-scattering approximations, I think it is important to include these methods in the thesis because they are very common in real-time applications such as video games. They are very fast and can often be configured by experienced artists to give good visual results.

2.1 Color blending

Color blending is a very old method based on a very simple approximation of the volume rendering equation. As we will see, this method is so useful that it has even been included in OpenGL at least since 1994 [Segal and Akeley, 1994].

2.1.1 Motivation

In real-time 3D computer graphics, the performance of rendering on the GPU heavily depends on geometry complexity and texture resolutions. If our scene contains a lot of detailed objects (meshes) with high-resolution textures, the rendering time can be too high. As mentioned in the introduction, fast rendering is essential for real-time applications.

Because we cannot render too many detailed objects, we have to accomodate techniques such as *level of detail* (LOD). It means that objects that are far from the camera are rendered with lower quality, i.e., lower geometry detail or lower texture resolution. Typically, we also decide not to render meshes that are too far. For this purpose, we can define a *far clipping plane* behind which no geometry is rendered.[1]

It is obvious, though, that once we decide to use LOD or clipping of geometry, the user of our application can notice that some objects are low quality or completely missing. Therefore, we need a way to mask this problem.

[1] It should be noted that clipping planes have other purposes as well, such as suppressing precision problems of floating point numbers in a *z-buffer*.

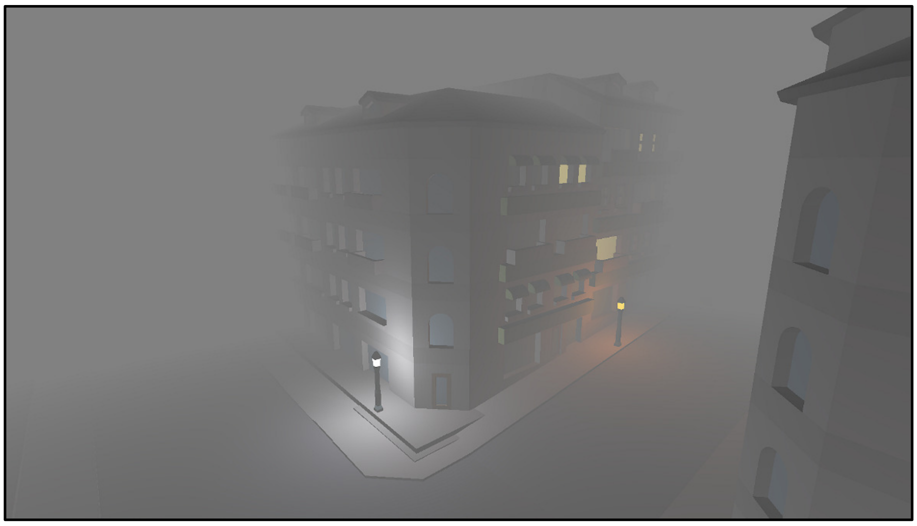


Figure 2.1: Linear color blending fog. As can be seen, the transition is very sharp and the fog has a uniform color independent on the light sources around it.

2.1.2 Idea

One of the very old solutions, which is still used in modern applications, is to smoothly hide distant objects in a fog. Because complex rendering of a fog as a participating medium would be very slow, we instead approximate the fog by simple color blending. This idea was documented, for example, by Segal and Akeley [1994] in the specification of the first version of OpenGL.

The algorithm is based on blending (mixing) the color of a pixel with the color of the fog depending on the distance of the pixel from the camera (Figure 2.1). By "distance of the pixel from the camera", we mean how far the rendered geometry on that pixel is from the camera (Figure 2.2 shows how the 2D pixel position is related to its 3D position). Sometimes, we refer to this distance as the *depth* of the pixel as the information is usually stored in a *depth buffer*, also called *z-buffer*.

It is important to note that this algorithm does not rely on any advanced equations and is purely empirical. Its computation is therefore very fast and independent on the number of light sources and the complexity of the rendered geometry.

Color equation

The idea of color blending stems from the observation that in the presence of a fog, distant objects often disappear in a white "cloud". In case of a homogeneous fog, the intensity depends on how far the objects are. We therefore assume that the fog itself has a white or grey color and we blend the color of the pixel with this color [Klawonn, 2008].

Let $\mathsf{L}\colon \mathbb{N}^2 \to \mathbb{R}^3$ represent a two-dimensional image of a certain scene. We assume that there is no participating medium in the image, e.g., the image can be the output of a standard rendering algorithm for a vacuum. We will denote $\mathbf{C} \in \mathbb{R}^3$ the color of the fog. Let $\mathsf{D}\colon \mathbb{N}^2 \to \mathbb{R}_0^+$ denote the depth buffer, i.e., a non-negative function of the depths of the pixels of L.

The new image $\mathsf{L}' \colon \mathbb{N}^2 \to \mathbb{R}^3$ can be computed as:

$$\mathsf{L}' = b(\mathsf{D}) \cdot \mathbf{C} + (1 - b(\mathsf{D})) \cdot \mathsf{L}, \tag{2.1}$$

where b is the blending function defined in the following section.

As we can see, the new color is the result of the linear combination of the old color and the fog color. It may not be immediately obvious, but Equation 2.1 is just a very simple approximation of the VRE (Equation 1.15). The total volume contribution is approximated by $b(\mathsf{D}) \cdot \mathbf{C}$, the attenuated surface contribution corresponds to $(1 - b(\mathsf{D})) \cdot \mathsf{L}$.

Blending functions

The blending function $b \colon \mathbb{R}_0^+ \to [0, 1]$ is an increasing function that tells us how much should a pixel in a certain distance be colored by $\mathbf{C}$.

We typically want objects near the camera to retain their original color, therefore we want $b(0) = 0$. Additionally, objects that are very far away—and therefore could be clipped and not rendered at all—should be completely "hidden" in the fog. Hence, for $d \to \infty$, we want $b(d) \to 1$.

Linear fog

Segal and Akeley [1994] and Klawonn [2008] describe a very simple linear blending function. For distances lower than a certain threshold d_0, the pixels keep their original color. On the other hand, all pixels that are further than d_1 are completely colored by the fog. Between d_0 and d_1, we mix the colors linearly.

The linear blending function for distance d is defined as:

$$b(d) = \begin{cases} 0 & \text{if } d \leq d_0, \\ \frac{d - d_0}{d_1 - d_0} & \text{if } d_0 < d < d_1, \\ 1 & \text{if } d \geq d_1. \end{cases} \tag{2.2}$$

Exponential fog

The problem with the linear fog is that the blending may look very sharp. If we replace Equation 2.2 with an exponential function, we can achieve much smoother results. The specification of OpenGL [Segal and Akeley, 1994] also suggests that a squared exponential can be used. For a custom parameter $c > 0$, we can define the following two functions:

$$b(d) = 1 - \exp(-c \cdot d), \qquad b(d) = 1 - \exp\left(-(c \cdot d)^2\right).$$

Notice the similarity to the Beer–Lambert law (Equation 1.8). For a homogeneous medium, the transmittance exponentially decreases with the distance. Here, b increases, because we use it to blend with the fog color.

Please note that although we use the name "exponential fog", its density does not depend on altitude. The only parameter of the blending function is the depth of the pixel. Later in Chapter 4, we explain a density function that is exponential *with regards to altitude.* These two ideas should not be confused.

2.1.3 Density based blending

The previous solutions are primarily based on the idea that we need to mask the effects of LOD and clipping. The blending functions are designed to not be visually disruptive and to be easily computed during the rendering.

Quílez [2010, 2015] proposes an improvement to this approach. Instead of defining an empirical blending function, he suggests blending the colors according to the physical density of the medium.

Even though the idea of "blending the fog color" is still a huge simplification, let us now assume that we have a participating medium with the density function $\varrho\colon \mathbb{R}^3 \to \mathbb{R}_0^+$ (Section 1.2). We can now define a new blending function b' based on ϱ.

For each pixel of L, there exists a ray from the camera towards the object on that pixel. The new blending function is based on the total density integrated along the ray from the camera. For each pixel position $\mathbf{x}_{2\text{D}} \in \mathbb{N}^2$, let $\mathbf{x} \in \mathbb{R}^3$ be its corresponding location in the 3D space based on the depth of the pixel. We can now define $b'\colon \mathbb{N}^2 \to \mathbb{R}_0^+$ as (Figure 2.2):

$$b'(\mathbf{x}_{2\text{D}}) = \int_{\text{camera}}^{\text{pixel } \mathbf{x}} \varrho(\mathbf{x}')\,\mathrm{d}\mathbf{x}'.$$

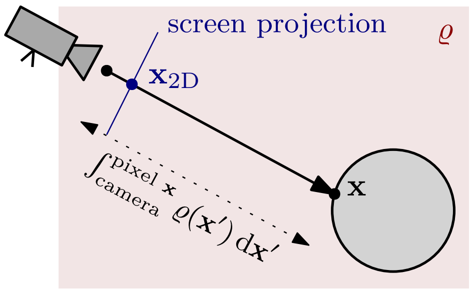

screen projection
ϱ
$\mathbf{x}_{2\text{D}}$
$\mathbf{x}$
$\int_{\text{camera}}^{\text{pixel } \mathbf{x}} \varrho(\mathbf{x}')\,\mathrm{d}\mathbf{x}'$

Figure 2.2: Integrating the "density of a pixel".

Note that we cannot use b' directly in Equation 2.1 because the values are not in $[0, 1]$, but the equation can be normalized by a certain maximum density.

The idea of calculating the medium densities along camera rays is also used in our proposed method (Chapter 4). Later in Section 4.3, we show a few examples of density functions that can be analytically integrated, hence can be used for real-time approaches.

2.1.4 Conclusion

The color blending approach is based on the assumption that we can approximate the volume rendering equation by simple color blending. Such solution, unfortunately, does not take any properties of the actual scene into account, apart from the arbitrary "fog color". Even for a simple scene with a single light bulb in a homogeneous participating medium, it is obvious that this approach is not even able to correctly blur the light in the scene.

The strong benefit of this approach is that it can be computed very easily. The time complexity of the method depends only on the resolution of the input image L. It does not depend on the number of light sources in the scene, nor on how complex the geometry is. With regards to Section 1.4, it is clear that this method can be applied by filtering in screen-space for each pixel in parallel. Furthermore, the presented idea that we can blend the color according to the integrated density can substantially improve the visual results.

2.2 Billboards and particles

In certain video games, the effects of a participating medium are approximated by billboards and particles [Wronski, 2014]. By *billboards* we usually understand two-dimensional semi-transparent textures that are always facing the camera. They can be placed in a scene at a static location. *Particles* are a similar concept, but there are typically many of them, they move, appear, and disappear according to rules set for them. The particles can be rendered by using billboards.

Billboards enable us or an artist to paint a texture for a pre-defined location in our scene. The texture can, for example, display a scattered light beam entering a dark room from a window. The obvious disadvantage is that the billboards are artist dependent. They have to be carefully arranged to look plausible from all angles of the camera. We are not able to use them in dynamic environments, such as when the lighting can change in time.

Particle effects can be used to simulate local effects, e.g., to simulate smoke. Because particles are dynamic, the smoke can move in our scene. Persson [2012] explains how we can calculate the color of the particles depending on the light sources around. Please note that his solution is only based on the diffuse lighting and transparency of the billboards that represent the particles. It can be understood as an empirical approach as it does not really calculate the scattering effects at all.

We are not going to reference to these empirical methods further in this thesis. We only presented the idea because it is used in video games and in certain conditions can be used as an empirical approximation for a participating medium, such as a smoke.

2.3 Crepuscular rays rendering

Crepuscular rays, also known as sunbeams, light shafts, or God rays, are visible rays of sunlight divided by columns of shadows. These phenomena occur when the sunlight is partially occluded by other objects, such as dense clouds, tall buildings, or trees (Figure 2.3). The occlusion causes the light to be visually "divided" into rays of varying intensity. This effect is typical for far and intense light sources, such as the Sun, but it technically occurs for every light in all participating media, only it may be less visible.

2.3.1 Method overview

Mitchell [2007] proposes how the crepuscular rays can be approximated and rendered in real-time in a screen-space shader. His idea is based on simplifying the

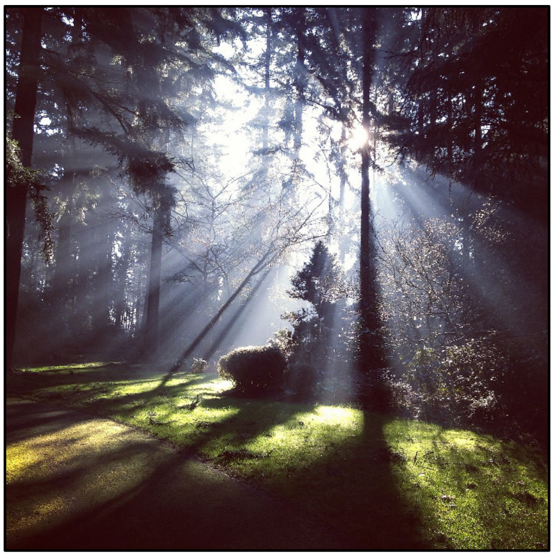


Figure 2.3: Photo of crepuscular rays caused by the sun light being partially occluded by trees. (Photo by Becker [2015], dedicated to the Public Domain)

total volume contribution term in the VRE (Equation 1.15) by approximating how light in-scatters from the sun.

Sun in-scattering

For a camera ray of length s and the angle θ between the sun and the ray, he proposes to express the volume contribution as:

$$L_{\mathrm{in}}(s,\theta) = (1 - \exp(-\sigma_{\mathrm{t}} s)) \cdot \sigma_{\mathrm{t}}^{-1} \cdot E_{\mathrm{sun}} \cdot f_{\mathrm{p}}(\theta),$$

where E_{sun} is the energy of the sun, σ_{t} is the extinction coefficient, and f_{p} is the phase function.

As we can see, the previous equation actually takes the physical parameters of the medium into account. Unfortunately, it only assumes single scattering. It obviously does not handle any light that is scattered multiple times, e.g., from the sun towards a point, then from that point towards the camera ray, and then again along the ray. Also, it ignores all other light sources in the scene as it only takes the in-scattering from the sun into account.

By substituing the previous equation into the VRE, we can express the total radiance along a camera ray. We already know that the original surface radiance L_0 gets attenuated according to the Beer–Lambert law (Section 1.2.4). Therefore, the total radiance along the camera ray can be expressed as (Figure 2.4):

$$L(s,\theta) = \underbrace{L_{\mathrm{in}}(s,\theta)}_{\text{volume contribution}} + \underbrace{\exp(-\sigma_{\mathrm{t}} s)\, L_0}_{\text{surface contribution}} . \tag{2.3}$$

Please note that this equation is clearly a simplification of the original volume rendering equation (Equation 1.15). We merely replaced the integral of the total volume contribution by L_{in}.

Shadows sampling

The radiance computed in Equation 2.3 does not take any occlusions into account. If we want to express that certain rays are less intense—i.e., their radiance is lower—we have to reduce their radiance by a certain factor. Let $D(\boldsymbol{\phi})$ denote the combined opacity of all objects that occlude the sun for the view location $\boldsymbol{\phi}$. Mitchell [2007] proposes that for the total radiance along the view path, we can write:

$$L(s, \theta, \boldsymbol{\phi}) = (1 - D(\boldsymbol{\phi})) \cdot L(s, \theta).$$

In screen-space, in the post-processing step, we cannot exactly calculate the occlusion. However, we are able to estimate it by sampling along a ray to the light source in image space (Figure 2.4). For n samples along the ray, we can write:

$$L(s, \theta, \boldsymbol{\phi}) = \sum_{i=0}^{n} \frac{1}{n} L(s_i, \theta_i).$$

The original method later describes that we have to control the summation by certain parameters: exposure, weight, and decay. We are not going to explain these details in the thesis, because they are only necessary when implementing the method. For more information, please refer to the original article by Mitchell [2007].

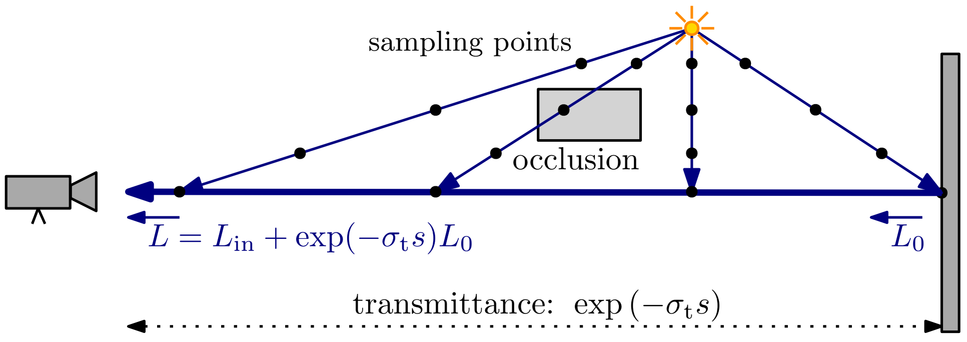

sampling points
occlusion
$\overleftarrow{L} = L_{\text{in}} + \exp(-\sigma_{\text{t}} s) L_0$
$\overleftarrow{L_0}$
transmittance: $\exp(-\sigma_{\text{t}} s)$

Figure 2.4: Single scattering and sampled occlusions of sunbeams.

2.3.2 Conclusion

As we could see, this post-processing method is able to approximate the single scattering and occlusions of sunlight. We presented the method in this thesis to show how the original VRE can be simplified for certain conditions and assumptions. This method, unfortunately, does not handle multiple-scattering effects and is originally designed for a single light source only. The time complexity depends on the image resolution and the number of samples.

2.4 Volumetric fog

The solution we presented in the previous section is based on sampling the in-scattering in the final two-dimensional image, i.e., in screen space. We are now going to briefly explain an algorithm that instead uses a three-dimensional (volumetric) texture. While it is still going to be only a single-scattering solution, it will support multiple light sources. The method will also take occlusion into account, which means that it will also be able to render the crepuscular rays.

2.4.1 Method overview

The algorithm was presented by Wronski [2014] and he decided to use "volumetric fog" as the name of the solution. The name summarizes the idea quite well, because we use volumetric textures to store intermediate results of the light transport calculations.

Basically, we represent the visible scene in front of our camera in a discrete three-dimensional texture (Figure 2.5). The region of space of the visible geometry is typically called the *view(ing) frustrum* or the *camera frustrum*. The volumetric fog algorithm works by calculating the light transport in each *cell* of the texture individually. The original work suggests that we can use volumes of size $160 \times 90 \times 64$, which means that there is almost a million of these cells. The number of cells is equal to the number of pixels in the HD resolution 1280×720, which is what modern GPUs can handle without any problems.

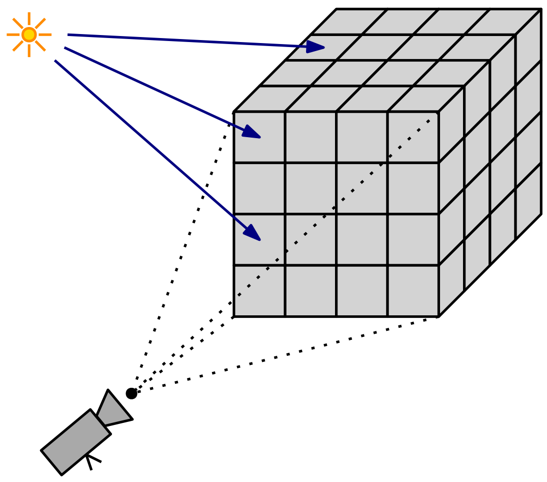


Figure 2.5: The visible scene is represented in a discrete 3D texture. The light transport is calculated for each cell individually.

There are four major steps. First, we calculate the lighting and shadowing for every volume cell. We get a 3D texture A. Then, we estimate the density of the medium, getting a 3D texture B. By combining the information from A and B, we perform a ray marching step while solving the volumetric rendering equation. We will explain this step in more details later. In the final phase, we use the ray marched information to apply the colors to the final 2D texture, which we can later display on the screen. It is important to note that the first two steps are independent on each other. We can fully run them in parallel.

Ray marching

The third step is based on ray marching. We take the already computed 3D textures A and **B**, and we intersect them by the camera rays of the camera projection. For each ray, we *march* through the volume. It means that starting from the camera, we move along the ray by constant steps while accumulating the computed data. We stop once we reach a surface, i.e., we march from 0 to the pixel depth. The result of this step is the calculated density sum and the accumulated in-scattering radiance and out-scattering factor.

Result

The data we get from the ray marching step represent numerically integrated information about the light transport. We can now apply them to the final image L'. Let L denote the original image rendered in a vacuum. During the ray marching step, we calculated the transmittance information T and the in-scattering information S. Both T and S are discrete 2D textures of the same resolution as L and L'. The final color can be computed as:

$$\mathsf{L}' = \mathsf{T} \cdot \mathsf{L} + \mathsf{S}.$$

As we can see, the previous equation is, again, a mere approximation of the VRE (Equation 1.15). The attenuated surface contribution is represented by $\mathsf{T}\cdot\mathsf{L}$, the volume contribution by S. As in the previous method, this solution also does not take multiple-scattering effects into account, because S only represents the single in-scattering from the light sources.

2.4.2 Conclusion

The volumetric fog uses three-dimensional textures to store intermediate results of pre-calculations. By computing the parameters of a medium in the cells of a volumetric texture, we can even render heterogeneous media. Wronski [2014] stated that the method takes only around 1.1 ms to run on Xbox One with a satisfying resolution. That is a very impressive result.

However, the method does not take multiple-scattering effects and global illumination into account. The in-scattering term is only computed from the light sources. The time complexity depends not only on the resolution but also on the amount of the light sources, as we have to calculate the in-scattering for each of them. Approximations of global illumination are possible and are very briefly explained in the original article.

In this thesis, we presented this idea because it is a very fast solution and it produces results that are acceptable for real-time video games. This approach has been successfully used in the commercial video game *Assassin's Creed IV* developed by Ubisoft Montreal.

3. Methods related to multiple scattering

The previously described methods assume that the participating medium has a constant ambient color (Section 2.1), can be approximated by colored textures (Section 2.2), or its color contribution can be computed as single scattering from light sources in the scene (Sections 2.3 and 2.4). These methods cannot blur the illumination reflected from the objects in the scene.

However, the light scattering in real life situations manifests itself mainly by blurring of the illumination [Elek et al., 2013] in the scene, obviously including reflected light. In a real fog, objects around us get visibly blurry, their edges are smudged, the colors are blended together. That happens because in the VRE (Equation 1.15), the volume contribution term is calculated by integrating the radiance from *all* directions in a sphere.

Screen-space approaches presented in this chapter understand every pixel of an image as a light source of its own. It enables us to take the whole illumination into account. Furthermore, we can also approximate multiple scattering of the illumination of the pixels.

This chapter is devoted to a brief examination of two screen-space methods that can approximate the multiple-scattering effects to a certain degree. Our proposed method described later in Chapter 4 is strongly inspired by the ideas presented in this chapter, especially by Sections 3.1 and 3.2.

3.1 Path integral and spatial spreading

Elek et al. [2013] presented a screen-space multiple-scattering method for homogeneous media. It is based on image filtering (Section 1.4) of the input image L that was rendered in a vacuum. Our goal is to derive the final image L' by taking the light transport in a homogeneous participating medium into account.

The key idea is not any different from what we have already worked with. Each pixel in the input image represents the light that reached the camera from the object on the pixel. It essentially corresponds to the radiance that the camera received from the light beam in the direction of the pixel. As we would like to blur the pixels, we try to derive the blurring point spread function (Section 1.4.3) that we can apply to the image L to get the blurred image L'.

3.1.1 Path integral

When measuring the radiance along a light beam passing through a participating medium, we can use the RTE (Equation 1.14) to describe the local behavior, or the VRE (Equation 1.15) to describe the total radiance. We have already described several approaches trying to simplify the original equations. Let us now take a look at a slightly different approach.

As described in several articles, such as by Premože et al. [2004] or Elek et al. [2013], we can reformulate the equations for a light path by using the *Green* *propagator* G:

$$L(\mathbf{x}, \boldsymbol{\omega}) = \int\int G(\mathbf{x}, \mathbf{x}', \boldsymbol{\omega}, \boldsymbol{\omega}') L_0(\mathbf{x}', \boldsymbol{\omega}') \, \mathrm{d}\mathbf{x}' \, \mathrm{d}\boldsymbol{\omega}',$$

where L_0 represents how the initial radiance is distributed in space. For the initially emitted light at the point $\mathbf{x}'$ in the direction $\boldsymbol{\omega}'$, the Green propagator represents the radiance at the point $\mathbf{x}$ in the direction $\boldsymbol{\omega}$.

When light travels between $\mathbf{x}$ and $\mathbf{x}'$ in a participating medium, the photons may take different paths, but still arrive at the same point in space. That is because of the multiple-scattering effects (Section 1.2). We can therefore imagine that the light transport process is a sum of all transfer events occurring at all possible paths the photons can take. Hence, the Green propagator can be expressed by a *path integral* [Premože et al., 2004]:

$$G \sim \int_{\text{all paths}} \exp\left(-\tau(\text{path})\right),$$

where $\tau(\text{path})$ denotes the *effective attenuation* along the path. As we can see, the function we are integrating corresponds to the transmittance for a light path defined by the Beer–Lambert law (Equation 1.8).

3.1.2 Spatial spreading

In Section 1.2, we explained that multiple scattered light is spatially spread (see again Figure 1.5). We would like our real-time method to take spatial spreading into account as it is essential for the light transport. The idea is to take the path integral from the previous section and use it to derive a formula that would express how much the light spreads.

Before deriving such formula, we have to make a very important assumption. We can think of light scattering as a stochastic process with many realizations. Because of this observation and according to the central limit theorem, we can assume that the spatial distribution is Gaussian [Elek et al., 2013] (Figure 3.1).

We will denote the standard deviation of this Gaussian distribution by W, as we already use the common symbol σ to describe medium parameters. The letter W can stand for the word *width*. For a collimated pencil of light between two points with the distance s, Premože et al. [2004] derived the formula for the standard deviation of the Gaussian spatial-spreading distribution:

$$W(s) = \sqrt{\frac{1}{2}\left(\frac{2\sigma_\mathrm{a}}{3s} + \frac{4}{s^3\sigma_\mathrm{s}(1-g)}\right)^{-1}}. \tag{3.1}$$

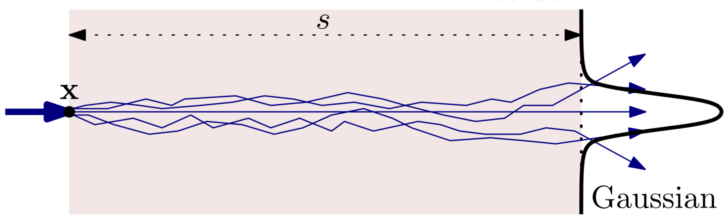


Figure 3.1: The light beam entering at the point $\mathbf{x}$ is spatially spread on the sensor. The spreading can be approximated by the Gaussian distribution.

3.1.3 Filtering

We have already stated that the solution of Elek et al. [2013] is a screen-space method and is based on image filtering. Therefore, we would like to blur each pixel of the original image according to the Gaussian distribution.

Spatially varying blurring

If the whole image had the same standard deviation W, we could compute the new image by a simple convolution with a Gaussian kernel. Unfortunately, as is obvious from Equation 3.1, the blurring width may be different for each pixel according to the distance from the camera. Our filter will have to be spatially varying as described in Section 1.4.5.

For the pixel position $\mathbf{x}'$ with the depth s, let $G_{\mathbf{x}'}$ be its point spread function, i.e., the two-dimensional Gaussian with the standard deviation $W(s)$. We can now use these PSF as the weighting functions according to Equation 1.16:

$$\mathsf{L}'(\mathbf{x}) = \sum_{\mathbf{x}'} G_{\mathbf{x}'}(\mathbf{x}' - \mathbf{x})\mathsf{L}(\mathbf{x}').$$

Because the 2D Gaussian distribution is circularly symmetric, we can redefine our weighting function to only take the distance of the two pixels into account:

$$\mathsf{L}'(\mathbf{x}) = \sum_{\mathbf{x}'} G_{\mathbf{x}'}(\|\mathbf{x}' - \mathbf{x}\|)\mathsf{L}(\mathbf{x}'). \tag{3.2}$$

We can now decide to apply the filtering with an algorithm from Section 1.4.6. The time complexity of these algorithms is, however, very high. Elek et al. [2013] decided to call this solution the *screen-space reference*. Later in this chapter, in Section 3.2, we show a very efficient approximation of the Gaussian blurring that can significantly lower the time complexity.

Preprocessing

If L in Equation 3.2 denoted the original input image, the filtering would not give us correct results. The Gaussian distribution tells us how a pixel scatters in space, but it does not take the light attenuation into account. Notice that by filtering with our Gaussian weighting functions, we are not decreasing the total radiance but only distributing it spatially.

Therefore, we must first *preprocess* the input image L with pixel depths D as described by Elek et al. [2013]. We use the Beer–Lambert law (Equation 1.8) to get the following two images. Let L_{at} represent the *attenuated image*, i.e., the radiance that reaches the camera directly without getting absorbed or scattered. Let L_{sc} represent the radiance that *is scattered* on the way to the camera, but is *not absorbed*. We can write:

$$\begin{aligned} \mathsf{L}_{\mathrm{at}} &= \exp\left(-\sigma_{\mathrm{t}}\mathsf{D}\right) \cdot \mathsf{L}, && (3.3) \\ \mathsf{L}_{\mathrm{sc}} &= \exp\left(-\sigma_{\mathrm{a}}\mathsf{D}\right) \cdot \left(1 - \exp\left(-\sigma_{\mathrm{s}}\mathsf{D}\right)\right) \cdot \mathsf{L}. && (3.4) \end{aligned}$$

Please note that in general, $\mathsf{L}_{\mathrm{at}} + \mathsf{L}_{\mathrm{sc}} \neq \mathsf{L}$, because the absorbed radiance is in neither of the two images. The radiance that is absorbed does not reach the camera at all, hence is attenuated and “lost” in the preprocessing step.

Result

We can now apply the filtering from Equation 3.2 to the image L_{sc}. The final multiple-scattered image L' can be expressed as the sum of the attenuated radiance and the blurred scattered radiance:

$$\mathsf{L}'(\mathbf{x}) = \mathsf{L}_{\text{at}}(\mathbf{x}) + \underbrace{\sum_{\mathbf{x}'} G_{\mathbf{x}'}(\|\mathbf{x}' - \mathbf{x}\|)\mathsf{L}_{\text{sc}}(\mathbf{x}')}_{\text{blurred scattered radiance}} . \tag{3.5}$$

3.1.4 Conclusion

We have presented the background behind the method of Elek et al. [2013], but we are still missing the most important part of the work. Because the time complexity of the standard filtering approaches is too high, it is not really possible to use it in real-time. Therefore, Elek et al. analyzed and implemented much more efficient filtering. We describe it in the next section (Section 3.2).

Before proceeding, let us very briefly summarize the idea. We are able to render the light absorption and light scattering effects by preprocessing and blurring (filtering) an input image and then compositing the results. The approach assumes homogeneous media only. Later in Chapter 4, we present our improved method based on this approach, but we modify it to support even non-homogeneous analytically integrable media.

3.2 Gaussian filtering with MIP maps

Let us now have a look at efficient Gaussian filtering with the concept of *MIP maps*, sometimes spelled *mipmaps*. They are textures of progressively lower resolution, where each subsequent level is a power of two smaller than the previous.

We will denote the original image $\mathsf{L} = \mathsf{L}^{[0]}$ as level 0, then image of half the original resolution $\mathsf{L}^{[1]}$ will be level 1, and so on. The pixels correspond to the ones from other levels as seen in Figure 3.2a. Because GPUs support this concept very well, it can be used for very fast efficient filtering.

3.2.1 Convolution in MIP maps

We now briefly explain how Lee et al. [2009] proposed to Gaussian blur images with MIP maps. They originally presented a method for real-time rendering of *depth of field* (DOF) effects. Real optical systems, including cameras and human eyes, cannot focus on all objects in a scene at once. The DOF represents the distance range where objects appear to be in the focus, i.e., they appear sharp. The objects that are closer or further appear blurred.

Lee et al. [2009] proposed a method for rendering this effect in screen-space, i.e., applying the blurring to the already rendered image where all objects are in the focus. They assume that the blurring can be approximated by filtering the image with a Gaussian kernel, which is exactly what need in Section 3.1.

Let $\mathcal{G}$ represent the 2D Gaussian convolution kernel with the standard deviation of 1 and of a small fixed size, e.g., 4×4 as proposed by Elek et al. [2013]. We

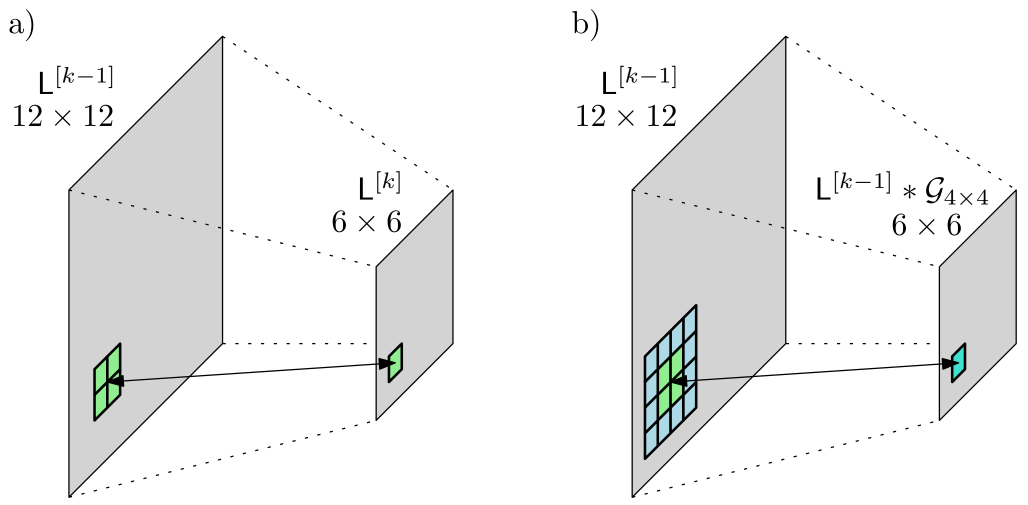


Figure 3.2: In standard MIP maps (left), a single pixel in the level $k > 0$ represents 4 pixels from the previous level $k - 1$. In Gaussian blurred MIP maps (right), a single pixel represents the corresponding blurred neighborhood from the previous level. The size of the neighborhood is determined by the size of the Gaussian kernel (here 4×4, i.e., 16 pixels).

want to use an even sized kernel instead of the typical odd sized ones, because we want the output to have half the resolution of the input (Figure 3.2b).

For $k > 0$, we express the MIP level k as:

$$\mathsf{L}^{[k]} = \mathsf{L}^{[k-1]} * \mathcal{G}, \tag{3.6}$$

where the discrete kernel $\mathcal{G}$ is assumed to be already normalized, which is not a problem to achieve as the kernel is constant with a fixed size. Elek et al. [2013] suggests to use the weights $\{0.13, 0.37, 0.37, 0.13\}$, i.e., the 2D kernel matrix can be obtained by multiplying the weights row-wise and column-wise, so the left-most value would be $0.13 \cdot 0.13 = 0.0169$ and so on.

3.2.2 Standard deviation and MIP levels

It should be obvious from Equation 3.6 that the image is getting more and more blurred with the increasing level k. It essentially corresponds to blurring the original input image with an increasing standard deviation. The relation between the deviation W and the level k can be approximated as [Lee et al., 2009]:

$$W \approx c \cdot 2^{k-1}, \tag{3.7}$$

where c is the *scaling constant* that depends on the size of $\mathcal{G}$.

The exact value should be found by experimenting and comparing the results to the correct non-approximated convolution. Elek et al. [2013] mention that for the size 5×5, we can use $c = 0.56$, and for the size 4×4, we can use $c = 0.8$. Lee et al. [2009] use $c = 1.5$ for the filter size 3×3. As we can see, the scaling constant seems to be inversely proportional to the kernel size.

3.2.3 Fetching the result

Now that we know how the MIP levels correspond to the standard deviation, it remains to explain how to obtain the final filtered image.

Suppose that we want to filter the input image L and obtain the image L' that should be Gaussian blurred with the specific standard deviation W. We can proceed by building the MIP chain $\mathsf{L}^{[0\ldots K]}$ (also called the *pyramid*). Now, according to Equation 3.7, the standard deviation increases exponentially with the MIP level, so the MIP level has to increase logarithmically. Elek et al. [2013] propose to use clamping to approximate the continuous MIP level k:

$$k(W) \approx \operatorname{clamp}\left(\log_2 \frac{W}{c}, 0, K\right) \in [0, K], \tag{3.8}$$

where $\operatorname{clamp}(x, a, b)$ returns a for $x < a$, x for $0 \leq x \leq a$, and b for $x > b$.

Because the level k can be decimal, i.e., between two exact MIP levels, we have to use one-dimensional interpolation when obtaining the level. Furthermore, because the level may have lower resolution than the input image, we have to use two-dimensional interpolation to obtain the specific pixel (Figure 3.3).

The specific interpolation method should be chosen according to the quality we want to achieve. Elek et al. [2013] suggest to use the linear 1D interpolation combined with the bicubic 2D interpolation. Later in Section 4.5.4, we show the visual differences between the interpolation types.

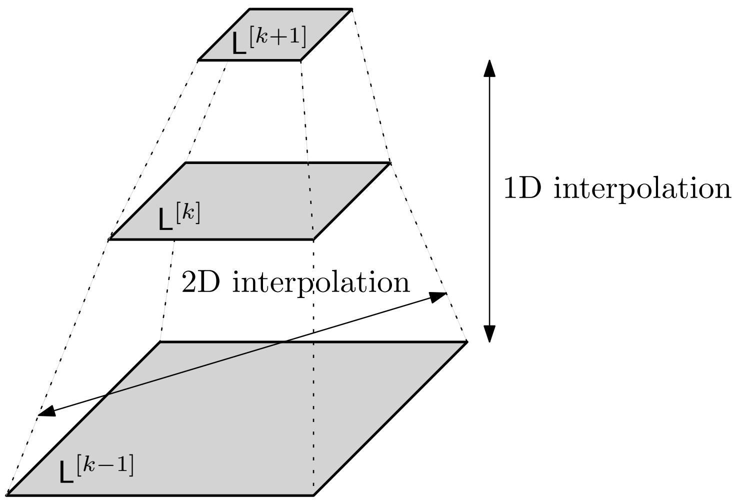


Figure 3.3: When obtaining a pixel from a MIP pyramid, we have to perform 1D interpolation between the levels and 2D interpolation between the pixels.

3.2.4 Spatially varying filtering

If we wanted to filter the whole input image L with a constant standard deviation W, we could obtain the result L' very simply by using Equation 3.8:

$$\mathsf{L}' = \mathsf{L}^{[k(W)]},$$

where the equality notation actually expresses the interpolations.

This is obviously not that simple for a spatially varying standard deviation, which is the case in the method from Section 3.1. The naive approach would be to compute the MIP level for each pixel individually.

For each pixel position $\mathbf{x}$, let $W(\mathbf{x})$ be the standard deviation for the pixel. We could then compose the result as:

$$\mathsf{L}'(\mathbf{x}) = \mathsf{L}^{[k(W(\mathbf{x}))]}. \tag{3.9}$$

Unfortunately, as explained by Elek et al. [2013] and further in Chapter 4, this approach is very naive and does not give very good results in most conditions. The major problem is that we build the MIP pyramid by using convolution as if the blur was spatially invariant. In high MIP levels, pixels from large neighborhoods are blurred together and we cannot separate their information anymore.

In Chapter 4, especially in Section 4.4, we show how Elek et al. [2013] proposed to solve the problem with visual artifacts. Furthermore, we propose additional enhancements in that section.

3.2.5 Conclusion and time complexity

We have just explained how to use the concept of MIP maps to efficiently apply the Gaussian blurring. By using the MIP maps, Elek et al. [2013] managed to significantly improve the performance of their multiple-scattering method. Even though they used an improved version of the filtering to reduce the artifacts, the execution time was much lower than for the naive gathering.

They measured the differences in the execution time for a sample image. For example, the path-tracing reference took 15 hours, the gathering algorithm took 30 seconds, and the enhanced MIP map filtering took 2 milliseconds.

Time complexity

When talking about the improvement of the execution time, we should analyze the time complexity of the MIP map filtering.

We have already explained that the splatting and gathering solutions based on neighborhood filtering have the time complexities $\mathcal{O}(n^2m^2)$ or $\mathcal{O}(n^2m^4)$ (Section 1.4.6). For HDR images, where a single pixel can be blurred into the whole image, we must assume that the neighborhood is the whole image, i.e., $m = n$. In that case, the time complexities are $\mathcal{O}(n^4)$ up to $\mathcal{O}(n^6)$.

Fortunately, the time complexity of the MIP map filtering is much lower. The algorithm is based on Equation 3.6. For each level $k > 0$, we have to perform the convolution with the constant-sized kernel $\mathcal{G}$. That means that for each pixel in the image, we have to iterate through the constant-sized neighborhood and accumulate the weighted contributions.

The biggest image $\mathsf{L}^{[0]}$ has the resolution $n \times n$, all higher levels have lower resolutions. The total number of MIP maps is $\mathcal{O}(\log_2 n)$, because the resolutions decrease exponentially and we cannot work with images of size lower than 1×1. Therefore, the total time complexity of the MIP map filtering is $\mathcal{O}(n^2 \log_2 n)$.

3.3 Narrow beam distributions

Before finally explaining our proposed method, I would like to very briefly introduce another solution. It was presented quite recently by Shinya et al. [2016] as another screen-space based multiple-scattering method. It is also based on filtering images that are originally rendered in a vacuum, but it builds on a slightly different background than the method by Elek et al. [2013].

It should be noted that Shinya et al. claim that their method gives more accurate results than the method by Elek et al. Unfortunately, their solution is not able to run as fast as the efficient MIP map filtering. As we will see in the conclusion, the rendering times mentioned in the original article are a bit too high for our real-time requirements. I still think that this method has a place in this thesis, especially because it can handle even non-homogeneous media.

3.3.1 Beam in a participating medium

Suppose that a narrow light beam with the radiance L_0 is entering a participating medium. The RTE (Section 1.3.1) without emissions can be expressed as:

$$(\boldsymbol{\omega} \cdot \nabla) L(\mathbf{x}, \boldsymbol{\omega}) = -\sigma_\mathrm{t}(\mathbf{x}) L(\mathbf{x}, \boldsymbol{\omega}) + \sigma_\mathrm{s}(\mathbf{x}) \int_{\Omega_{4\pi}} f_\mathrm{p}(\boldsymbol{\omega}, \boldsymbol{\omega}') L(\mathbf{x}, \boldsymbol{\omega}') \,\mathrm{d}\boldsymbol{\omega}', \tag{3.10}$$

where the phase function f_p is assumed to be spatially invariant, hence it does not depend on $\mathbf{x}$. Let us denote the right summand in the equation by L_v (volume contribution) similarly to Chapter 1.

To avoid the need to integrate the spherical integral in the RTE, we would like to approximate it with a planar integral that would be easier to solve. As described in details by Shinya et al. [2016], making the following assumptions enables us to do it. Further in this method, we assume that:

- the radiance L is concentrated near the original direction of the ray,
- the phase function varies smoothly and is axially symmetric around the forward direction,
- all medium particles are of the same type, so the medium cross-sections and phase function are spatially invariant,[1]
- furthermore, the density of the medium can only vary along the z axis, so for $\mathbf{x} = (x, y, z)$, we can write $\sigma_\mathrm{t}(\mathbf{x}) = \sigma_\mathrm{t}(z) = (C_\mathrm{a} + C_\mathrm{s}) \cdot \varrho(z)$.

With the previous assumptions, we can now replace the three-dimensional location $\mathbf{x}$ by a combination of $z \geq 0$ and $\mathbf{r} \in \mathbb{R}^2$ (Figure 3.4). The three-dimensional direction $\boldsymbol{\omega}$ can be replaced by $\mathbf{s} \in \mathbb{R}^2$. This enables us to write:

$$L(\mathbf{x}, \boldsymbol{\omega}) \approx L(z, \mathbf{r}, \mathbf{s}).$$

3.3.2 Screen-space filtering

We have made a significant approximation. We can now transform the original three-dimensional RTE into a two-dimensional problem by substituting the

[1] In this thesis, we already assume that the particles of the participating media are identical. It was mentioned in the assumptions in Section 1.2.

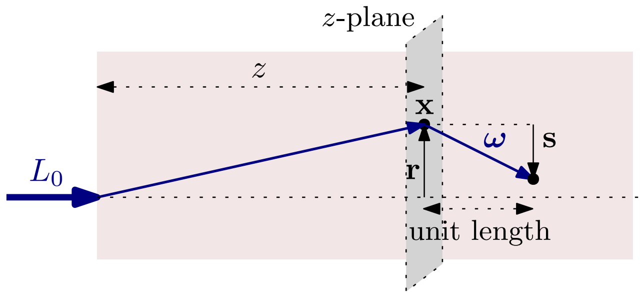


Figure 3.4: The coordinate system used by Shinya et al. [2016].

spherical integral. Since we are filtering images that are 2D, the idea is that we can solve the light transport in screen space.

Suppose that we have an image rendered in a vacuum. Let $I(\mathbf{r}, \mathbf{s})$ denote the image function parametrized in our coordinate system. Shinya et al. [2016] propose that we can understand L_v as the weighting function that tells us the weight of the volume contribution from other coordinates. It enables us to express the filtering for the location $\mathbf{r}$ by a somewhat abstract integral over all locations and directions:

$$\int\int L_\mathrm{v}(\mathbf{r} - \mathbf{r}', \mathbf{s}') I(\mathbf{r}', \mathbf{s}') \, \mathrm{d}\mathbf{r}' \, \mathrm{d}\mathbf{s}'. \tag{3.11}$$

Notice the similarity to Equation 1.16, which is discrete and two-dimensional.

We would like to further simplify the previous equation so that we can efficiently evaluate it in screen space. Calculating L_v would require us to solve a 5D integral, hence Equation 3.11 represents a 9D integral, which is still too complex.

3.3.3 Heuristic distribution

Fortunately, by using the Fourier transform, approximating the phase function by a Gaussian one, and calculating the moments of L_v, Shinya et al. [2016] managed to derive a heuristic distribution function $\Psi(z, \mathbf{r})$. The function tells us the weight of the contribution for the depth z at the vector $\mathbf{r}$ (Figure 3.4). Because the distribution only depends on the length of $\mathbf{r}$, we can write $\Psi(z, r)$, where $r = \|\mathbf{r}\|$.

In order to filter the image, we have to calculate the distribution function $\Psi_\mathbf{x}$ for every pixel position $\mathbf{x}$. Calculating the distribution function corresponds to evaluating the moments of L_v. In heterogeneous media, the moments may be different for each pixel $\mathbf{x}$, so the distributions have to be calculated separately. Shinya et al. [2016] approximate a heterogeneous medium by only calculating the moments in several sample planes along the z axis (Figure 3.4).

Without going into much details that are described in the original work, we can now filter in screen space according to our Equation 1.16. The weighting function $w_{\mathbf{x}'}$ can be easily derived from $\Psi_{\mathbf{x}'}$. The filtering can be done in shaders and run on the GPU for each pixel in parallel with the splatting and gathering algorithms (Section 1.4.6).

3.3.4 Conclusion

The approximation we have just described is able to take multiple scattering into account. Similarly to the solution by Elek et al. [2013], it is capable of blurring objects in the scene as it performs filtering of the whole image. Because the filtering is based on splatting and gathering, the time complexity obviously depends on the image resolution as described in Section 1.4.6.

The original article states that their implementation was able to filter a single frame in about 100–110 ms for the image size 640×480 on a computer with GeForce GTX 690. It is significantly better than path tracing that took several days on the same computer. Unfortunately, as we already explained in the introduction, we want real-time methods to take at most 40 ms to render a single frame with a fairly high resolution. Video games typically want to achieve even lower rendering times.

4. Our method

In the previous chapters, we have introduced the volumetric light transport problem and examined multiple different solutions to it. They were based on empirical, single-scattering, and multiple-scattering approaches.

As we could see, none of the introduced solutions is capable of sufficiently fast real-time rendering of multiple-scattering effects in non-homogeneous media. Let us now introduce our filtering-based method that tries to combine the great ideas of some of the existing approaches and further improve them.

In Section 4.1, we review what our method can achieve and what are the motivations, then briefly explain the overall process. In Sections 4.2, 4.3 and 4.4, we describe in great details how and why the whole process of our method works. The last section (Section 4.5) is devoted to explaining the advantages and limitations of our approach, also in the context of the competing solutions.

4.1 Overview

Before describing our approach in great details, let us now have a look at a short overview of what exactly our method achieves.

4.1.1 Goals and motivations

Some the requirements for our method have already been mentioned in Introduction but without sufficient details. Let us now formalize what our approach achieves while explaining the motivation behind the achievements.

Real-time GPU rendering

The very important goal of our method is the ability to use it for real-time rendering. Without fulfilling this goal, it would not be possible to use the method in the use cases that we defined in Introduction.

Later in Chapter 5, we prove that our method on itself can process a single frame in only a few milliseconds. Together with the full rendering process in our prototype application, we reach frequencies higher than 25 FPS in the high definition (HD) resolution 1280×720. It means that our method runs faster than is necessary by the formal requirement by [Akenine-Möller et al., 2008] in a resolution that can be algorithmically upscaled if need by [Li et al., 2014].

As our method handles real-time rendering, it is reasonable that the algorithms are designed for being executed on GPUs. We implement our solution in fragment shaders later in Chapter 5 and test it on a fairly modern GPU GeForce GTX 760M.

Multiple scattering

From Chapter 2, it is obvious that real-time single-scattering solutions already exist and are even used in commercial video games, which are one of the possible use cases of our method. Our approach, however, simulates much more complex multiple-scattering effects.

Multiple scattering heavily contributes to the blurring effects caused by the spatial spreading of light (Section 1.2). Simulation of the correct blurring effect is essential as it is the main manifestation of the light scattering [Elek et al., 2013]. If we decided to ignore multiple scattering, we would not be able to render the correct effect.

Screen-space solution

It is also important to be able to process scenes that may contain an unlimited number of light sources and intensely emissive surfaces, i.e., objects defined to emit light from arbitrary geometrical shapes, such as neon tubes, lanterns, and glowing windows. Our method can blur even complex emissive geometry without increasing the rendering time. This enables us to work with complicated scenes at the same speed as with simple scenes.

We have achieved these results by using a screen-space approach. Thanks to this approach, the time complexity depends only on the display resolution. As described by Elek et al. [2013], we understand every pixel in the input image as a light source on its own. However, for simplicity, we disregard the fact that the light can be attenuated or scattered even before reflecting of the surfaces.

Non-homogeneous media

Another important achievement is the ability to render light scattering even in media that are not homogeneous. That basically means that the particle density $\varrho(\mathbf{x})$ may change with regards to the position $\mathbf{x}$ in space. Certain methods, such as the one described in Section 3.1, only assume media with constant densities.

This assumption can be quite restrictive with regards to what scenes we are able to render. According to the *barometric formula*, the pressure p of an ideal gas exponentially decreases with the altitude $z \geq 0$ [Berberan-Santos et al., 1997]:

$$p(z) = p(0) \cdot \exp\left(-\frac{mgz}{kT}\right), \tag{4.1}$$

where $p(0)$ is the pressure at $z = 0$, m is the molecular mass of the gas, T its temperature, and $g, k > 0$ are constants.

We can derive a similar exponential equation for $\varrho(z)$ as well, because the density is directly related to the pressure. As we can see from the equation, higher molecular mass results in faster decrease with regards to height, i.e., the more the gas weighs, the more it is concentrated at the ground level.

Even though we can approximate fog and other media to be homogeneous—as done by Elek et al. [2013]—we can get visually more interesting results with heterogeneous media.

Analytically integrable media

In order to render the scattering in a medium, we must be able to handle its density in real-time. For example, we need to calculate the transmittance (Equation 1.8), which indirectly depends on the density along a light path. Therefore, our method has to efficiently integrate the density function between two points.

The approach we have chosen is to use analytical integrations of the density functions. Unfortunately, it is not possible for any arbitrary density function. Fortunately, as already mentioned in Section 2.1.3, the densities of certain media—in this thesis referred to as *quasi-heterogeneous media*—can be analytically integrated. These media include, for example, a medium whose density is exponential with regards to altitude, which was our main goal because of the barometric formula. We can also analytically integrate other functions, such as a sphere. The specific equations are derived later in Section 4.3.

4.1.2 Outline of our approach

Because of our requirements, we decided to approach similarly to Elek et al. [2013] (Sections 3.1 and 3.2) with certain improvements. Unlike the original method, we propose a way to handle non-homogeneous media (Sections 4.2.3 and 4.3) and we introduce a new filtering step that we call the pixel separation (Section 4.4.4).

First of all, we suppose that we are already able to render a scene in a vacuum using a standard real-time rendering approach. Let $\mathsf{L}\colon \mathbb{N}^2 \to \mathbb{R}^3$ denote the input rendered image. The world-space distances from the pixels to the camera will be denoted by $\mathsf{D}\colon \mathbb{N}^2 \to \mathbb{R}$. The notation corresponds to the previous chapters.

Preprocessing

Based on the density function ϱ, camera, and distances D, we integrate the densities along the camera rays and store them in $\mathsf{P}\colon \mathbb{N}^2 \to \mathbb{R}$. Based on the input L and the integrated densities P, we then compute the following two images.

The image $\mathsf{L}_{\mathrm{at}}\colon \mathbb{N}^2 \to \mathbb{R}^3$ contains the attenuated radiance that reaches the camera without any interactions in the medium. The image $\mathsf{L}_{\mathrm{sc}}\colon \mathbb{N}^2 \to \mathbb{R}^3$ contains the radiance that is scattered but not absorbed on the way to the camera.

The next step is to compute the spreading widths of the pixels' points spread functions. Before doing that, we rescale the distances D with regards to P to get the rescaled distances $\mathsf{D}'\colon \mathbb{N}^2 \to \mathbb{R}$. With D', we then compute the spreading widths $\mathsf{W} = W(\mathsf{D}')$. The rescaling step has been introduced in our approach to support non-homogeneous media.

Filtering

During the filtering step, we build the Gaussian MIP chain $\mathsf{L}_{\mathrm{sc}}^{[0\ldots K]}$. To reduce artifacts of the filtering approach, we use three important concepts. The first one is based on luminance weighting for which we build the MIP chain $\mathsf{W}^{[0\ldots K]}$. The second one is based on depth blurring, so we build the MIP chain $\mathsf{D}'^{[0\ldots K]}$. The last step is based on separation of bright pixels, when we separate pixels $\mathsf{L}_{\mathrm{ssc}}\colon \mathbb{N}^2 \to \mathbb{R}^3$ from L_{sc} and build another MIP chain $\mathsf{L}_{\mathrm{ssc}}^{[0\ldots K]}$.

Composition

Based on the MIP chains, we have to compose the final image $\mathsf{L}'\colon \mathbb{N}^2 \to \mathbb{R}^3$. For this purpose, we use interpolation for fetching the correct pixels from the corresponding levels of the MIP pyramids. Furthermore, during the composition, we also add an arbitrary emissive term for enhanced visual results.

4.2 Preprocessing

The main goal of the preprocessing step is to prepare the values that are later used during filtering and compositing (Section 4.4). Our preprocessing approach is more complicated than the preprocessing for homogeneous media introduced by Elek et al. [2013] (Section 3.1.3). That is because for non-homogeneous media, we must handle densities and spreading widths in a different way.

4.2.1 Density

According to the RTE (Section 1.3.1), the light travelling from a pixel through a participating medium is attenuated with regards to the extinction coefficient. For a line segment l between the pixel and the camera, the attenuation corresponds to decreasing the radiance with the transmittance T (Section 1.2.4).

To compute T by using the Beer–Lambert law (Equation 1.8), we first have to know the optical thickness $\tau(l)$ along l. We can get τ by integrating the extinction coefficient (Equation 1.7) and that requires us to integrate the density ϱ:

$$\tau(l) = \int_l \sigma_\mathrm{t}(\mathbf{x})\,\mathrm{d}\mathbf{x} = \int_l (C_\mathrm{a} + C_\mathrm{s})\varrho(\mathbf{x})\,\mathrm{d}\mathbf{x} = C_\mathrm{t}\int_l \varrho(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{4.2}$$

Let us now assume that we already know how to integrate the density of our medium. Examples are later derived in Section 4.3. As the letter P is the uppercase of ϱ, let $\mathsf{P}\colon \mathbb{N}^2 \to \mathbb{R}$ denote the integrated densities along the line segments between the pixels and the camera. We integrate from the camera position in the 3D space towards the pixel's position in the 3D space (as in Section 2.1.3):

$$\mathsf{P}(\mathbf{x}) = \int_{\text{camera}}^{\text{pixel }\mathbf{x}} \varrho(\mathbf{x}')\,\mathrm{d}\mathbf{x}'. \tag{4.3}$$

4.2.2 Attenuated and scattered radiance

Similarly to Equations 3.3 and 3.4 introduced by Elek et al. [2013], we can express the attenuated and scattered radiance. This time, however, we cannot use the distances D directly in case the medium is not homogeneous. We have to combine Equation 4.2 with the Beer–Lambert law.

The attenuated image L_at can be computed as:

$$\mathsf{L}_\mathrm{at} = \exp\left(-C_\mathrm{t} \cdot \mathsf{P}\right) \cdot \mathsf{L}. \tag{4.4}$$

The scattered but not absorbed L_sc can be computed as:

$$\mathsf{L}_\mathrm{sc} = \exp\left(-C_\mathrm{a} \cdot \mathsf{P}\right) \cdot \left(1 - \exp\left(-C_\mathrm{s} \cdot \mathsf{P}\right)\right) \cdot \mathsf{L}. \tag{4.5}$$

4.2.3 Spatial spreading and distance rescaling

Now that we know what part of the energy is attenuated and what part is scattered, the idea is to blur the scattered radiance L_sc. We already know that the blurring width $W(D)$ for the length D can be approximated with Equation 3.1.

The equation, unfortunately, assumes the medium parameters to be spatially invariant along the light path. But we want to use the equation even in media that are not homogeneous. The original solution by Premože et al. [2004] is based on sampling the light path at multiple sample points, but the rendering takes several minutes. Instead, we propose to use the original equation for $W(D)$, but we rescale the distances D according to the optical thickness.

Rescaling idea

When light travels along a line segment with the length D, the spreading width $W(D)$ depends on the scattering interactions occurring along the segment (Sections 1.2 and 3.1.2). By looking at the light scattering as a stochastic process with many realizations, we can assume that $W(D)$ does not depend on where exactly the particles are accumulated along the line segment. We are only interested in how many particles in total could have interacted with the light.

We can use the following simple observation. For the line segment with the length D, if the light passes through a lot of particles, it spreads more than if it passes through less particles. And that is even though the length D is still the same. Therefore, we propose to approximate heterogeneous media by homogeneous ones, but we rescale the distances D with regards to the total optical thickness along the segments.

Naive mathematical derivation

For the line segment l of the length D, we can express the optical thickness $\tau(l)$ by using Equation 4.2. In a homogeneous medium with parameters $\sigma_{\mathrm{a}} + \sigma_{\mathrm{s}} = \sigma_{\mathrm{t}}$, we can write:

$$\tau(l) = \int_l \sigma_{\mathrm{t}}(\mathbf{x})\,\mathrm{d}\mathbf{x} = \sigma_{\mathrm{t}} \cdot D. \tag{4.6}$$

For a heterogeneous medium with parameters $C_{\mathrm{a}} + C_{\mathrm{s}} = C_{\mathrm{t}}$ and a density function $\varrho(\mathbf{x})$, the optical thickness is:

$$\tau(l) = C_{\mathrm{t}} \cdot \int_l \varrho(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{4.7}$$

We are working with an assumption that we can approximate an arbitrary medium by a homogeneous one by working with a rescaled length D'. From the equality between Equations 4.6 and 4.7, we get:

$$\sigma_{\mathrm{t}} \cdot D' = C_{\mathrm{t}} \cdot \int_l \varrho(\mathbf{x})\,\mathrm{d}\mathbf{x}. \tag{4.8}$$

Suppose that we know the relation between the constant density ϱ' of the homogeneous medium and the spatially varying $\varrho(\mathbf{x})$ of the other medium, so we can write:

$$\varrho(\mathbf{x}) = \varrho' \cdot \varrho''(\mathbf{x}),$$

where $\varrho''(\mathbf{x})$ is a spatially varying ratio between the densities of the arbitrary and the homogeneous medium. For the homogeneous medium itself, the ratio would obviously be constant for all locations in space: $\varrho''(\mathbf{x}) = 1$.

The optical thickness can now be expressed using the density function as:

$$\tau(l) = C_{\mathrm{t}} \int_l \varrho(\mathbf{x})\,\mathrm{d}\mathbf{x} = C_{\mathrm{t}} \cdot \varrho' \cdot \int_l \varrho''(\mathbf{x})\,\mathrm{d}\mathbf{x}.$$

Now because in homogeneous media, $\sigma_\mathrm{t} = C_\mathrm{t} \cdot \varrho'$ from the definition, we can solve Equation 4.8 for D' as:

$$\begin{aligned}
\sigma_\mathrm{t} \cdot D' &= C_\mathrm{t} \int_l \varrho(\mathbf{x}) \,\mathrm{d}\mathbf{x}, \\
\sigma_\mathrm{t} \cdot D' &= C_\mathrm{t} \cdot \varrho' \cdot \int_l \varrho''(\mathbf{x}) \,\mathrm{d}\mathbf{x}, \\
D' &= \int_l \varrho''(\mathbf{x}) \,\mathrm{d}\mathbf{x}. \qquad (4.9)
\end{aligned}$$

This equation naively enables us to approximate an arbitrary medium by a homogeneous one. We should verify that the equation holds true at least in case we compare the same homogeneous media. In that case, the length of the line segment l is D and the density ratio is obviously $\varrho''(\mathbf{x}) = 1$ for all $\mathbf{x} \in \mathbb{R}^3$ as we compare the exactly same media. Solving Equation 4.9 gives us:

$$D' = \int_l \varrho''(\mathbf{x}) \,\mathrm{d}\mathbf{x} = \int_l 1 \,\mathrm{d}\mathbf{x} = D, \qquad (4.10)$$

which obviously holds true as the length remains the same.

4.2.4 Spreading in screen-space

The problem with rescaling the distances is that it requires us to know the ratio function $\varrho''(\mathbf{x})$. When simulating an arbitrary medium, it is not clear how it can relate to a certain homogeneous medium.

When experimenting with various approaches, we found out that very good visual results can be obtained by simply assuming that the ratio function can be approximated by the density function: $\varrho'' \approx \varrho$. This approach assumes that the original homogeneous medium has a constant density $\varrho' = 1$, therefore $\sigma_\mathrm{t} = C_\mathrm{t}$.

This is a very important observation, because it significantly simplifies our method. We can use the original Equation 3.1 for the spatial spreading width by using the rescaled distances (Equation 4.9):

$$W(D') = W\left(\int_l \varrho''(\mathbf{x}) \,\mathrm{d}\mathbf{x}\right) \approx W\left(\int_l \varrho(\mathbf{x}) \,\mathrm{d}\mathbf{x}\right). \qquad (4.11)$$

Let $\mathsf{D}' \colon \mathbb{N}^2 \to \mathbb{R}$ denote the rescaled distances D and let $\mathsf{W} \colon \mathbb{N}^2 \to \mathbb{R}$ denote the spreading widths of the pixels in L_sc. With regards to Elek et al. [2013], we will also call W the *spread space*. We can simply write $\mathsf{W} = W(\mathsf{D}')$, but we may also notice that Equation 4.11 tells us $\mathsf{D}' \approx \mathsf{P}$:

$$\mathsf{W} = W(\mathsf{D}') \approx W(\mathsf{P}).$$

As we can see, the spreading widths depend on the densities between the pixels and the camera. The higher the density along the camera ray, the more the pixel spreads. For homogeneous media, our equation is in agreement with the original theory of Premože et al. [2004], because in homogeneous media, $\mathsf{D}' = \mathsf{D}$ (Equation 4.10).

In heterogeneous media, our approximation may give us certain errors. For example, if the density varies a lot and the variation is perpendicular to the line segment, the multiple-scattered rays can behave differently above and under the segment. In that case, we would not be able to assume the 2D Gaussian distribution anymore, because the interactions would not be evenly distributed.

Please note that in the rest of this thesis, we will write D' and P separately even though our observation enables us to write $\mathsf{D}' \approx \mathsf{P}$. Despite this approximation, we think it is important to understand the difference between the exactly integrated densities P and our naively approximated rescaled distances D'.

4.3 Analytical integrations of densities

In the previous section, we assumed that we can compute the integrated densities P in real-time. Let us now have a look at a few density functions that can be analytically integrated, i.e., the integrals can be exactly expressed by a formula that we can directly compute in a negligible time.

We will now work in the 3D world-space coordinate system as illustrated in Figure 4.1. We assume the camera to be located at the position $\mathbf{c} \in \mathbb{R}^3$. We will integrate the densities P for each pixel individually, i.e., we will calculate $\mathsf{P}(\mathbf{x}_{2\text{D}})$. Let $\mathbf{x}_{2\text{D}} \in \mathbb{N}^2$ denote the pixel's position on the screen and $\mathbf{x} \in \mathbb{R}^3$ denote the corresponding pixel's position in the 3D space. The normalized direction from the camera towards that pixel is $\boldsymbol{\omega} \in \mathbb{R}^3$. The distance between the pixel and the camera is $\|\mathbf{x} - \mathbf{c}\| = D \in \mathsf{D}$.

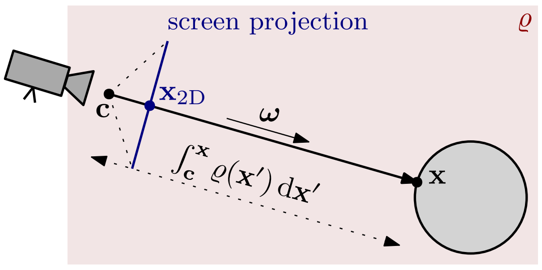


Figure 4.1: Integrating the density between the camera and the pixel.

4.3.1 Homogeneous medium

The simplest medium has a constant particle density $\varrho(\mathbf{x}) = \varrho'$ for all points in space. Let us call $\varrho' \geq 0$ the *constant factor* of the density function. We can integrate the density along a line segment between the camera and the pixel as:

$$\mathsf{P}(\mathbf{x}_{2\text{D}}) = \int_{\mathbf{c}}^{\mathbf{x}} \varrho(\mathbf{x}')\,\mathrm{d}\mathbf{x}' = \varrho' \int_{\mathbf{c}}^{\mathbf{x}} 1\,\mathrm{d}\mathbf{x}' = \varrho' \cdot D.$$

4.3.2 Exponential medium

By the *exponential medium* we understand a medium where the particle density can be modeled by an exponential function. The idea is closely related to the barometric formula (Equation 4.1). If the particle density changes with regards to altitude, the integration can be done as shown by Quílez [2010].

Let us now derive a more general exponential medium (Figure 4.2). We suppose that the density can exponentially decrease in *any* normalized direction $\mathbf{n} \in \mathbb{R}^3$. Furthermore, the density can be offset by any value $\mathbf{o} \in \mathbb{R}^3$.

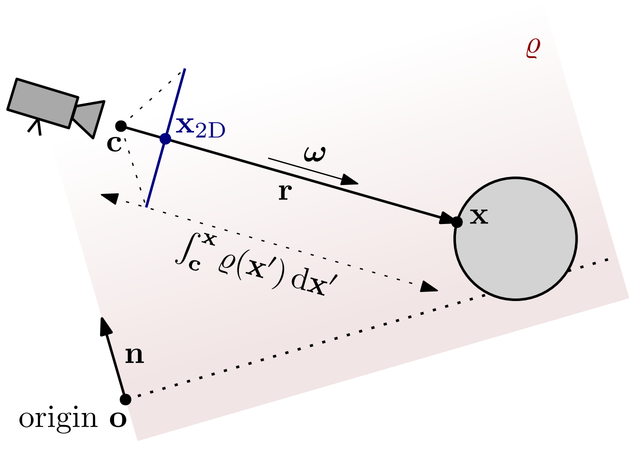

ϱ
$\mathbf{x}_{\mathrm{2D}}$
$\mathbf{c}$
$\boldsymbol{\omega}$
$\mathbf{r}$
$\mathbf{x}$
$\int_{\mathbf{c}}^{\mathbf{x}} \varrho(\mathbf{x}')\,\mathrm{d}\mathbf{x}'$
$\mathbf{n}$
origin $\mathbf{o}$

Figure 4.2: Integrating the density in an exponential medium.

The simple altitude-based model assumes that the density $\varrho(\mathbf{x}')$ only depends on the altitude $h(\mathbf{x}')$ of the location $\mathbf{x}'$. For a constant density factor $\varrho' \geq 0$ and an exponential parameter $b > 0$, we get:

$$\varrho(\mathbf{x}') = \varrho' \cdot \exp\left(-b \cdot h(\mathbf{x})\right). \tag{4.12}$$

In the general case, though, when the exponential is offset by $\mathbf{o}$ and its direction is $\mathbf{n}$, we have to update the equation:

$$\varrho(\mathbf{x}') = \varrho' \cdot \exp\left(-b \cdot \langle \mathbf{x}' - \mathbf{o}, \mathbf{n} \rangle\right), \tag{4.13}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product. Notice that Equation 4.12 is a special case where $\mathbf{o} = (0, 0, 0)$, $\mathbf{n} = (0, 1, 0)$, and $h(\mathbf{x}') = \langle \mathbf{x}' - (0, 0, 0), (0, 1, 0) \rangle$.

In order to integrate the density, we express the ray $\mathbf{r}$ between the camera and the pixel parametrically. For the parameter t, the camera ray $\mathbf{r}$ is defined as:

$$\mathbf{r}(t) = \mathbf{c} + t \cdot \boldsymbol{\omega},$$

where in our case, we assume $0 \leq t \leq \|\mathbf{x} - \mathbf{c}\| = D$.

The density integral then becomes:

$$\begin{aligned}
\mathsf{P}(\mathbf{x}_{\mathrm{2D}}) &= \int_{\mathbf{c}}^{\mathbf{x}} \varrho(\mathbf{x}')\,\mathrm{d}\mathbf{x}' \\
&= \int_0^D \varrho\left(\mathbf{r}(t)\right)\mathrm{d}t \\
&= \int_0^D \varrho(\mathbf{c} + t \cdot \boldsymbol{\omega})\,\mathrm{d}t \\
&= \varrho' \int_0^D \exp\left(-b \cdot \langle \mathbf{c} + t \cdot \boldsymbol{\omega} - \mathbf{o}, \mathbf{n} \rangle\right)\mathrm{d}t \\
&= \varrho' \int_0^D \exp\left(-b \cdot \left(\langle \mathbf{c}, \mathbf{n} \rangle + t\langle \boldsymbol{\omega}, \mathbf{n} \rangle - \langle \mathbf{o}, \mathbf{n} \rangle\right)\right)\mathrm{d}t,
\end{aligned} \tag{4.14}$$

where the last equality holds true because of the bilinearity of the inner product.

As the terms $\langle \mathbf{c}, \mathbf{n} \rangle$, $\langle \boldsymbol{\omega}, \mathbf{n} \rangle$, and $\langle \mathbf{o}, \mathbf{n} \rangle$ are constant for a certain pixel, the definite integral can be expressed by simply integrating the exponential function:

$$\mathsf{P}(\mathbf{x}_{\mathrm{2D}}) = \cdots = \varrho' \cdot \left[-\frac{\exp\left(-b \cdot \left(\langle \mathbf{c}, \mathbf{n} \rangle + t\langle \boldsymbol{\omega}, \mathbf{n} \rangle - \langle \mathbf{o}, \mathbf{n} \rangle\right)\right)}{b \cdot \langle \boldsymbol{\omega}, \mathbf{n} \rangle} \right]_0^D.$$

In case $\langle \boldsymbol{\omega}, \mathbf{n} \rangle = 0$, we cannot divide, so we solve Equation 4.14 directly. Otherwise, we compute the definite integral at the bounds D and 0, subtract the values, and apply the bilinearity again, which gives us the following formula:

$$\mathsf{P}(\mathbf{x}_{2\mathrm{D}}) = \cdots = \frac{\varrho'}{b \cdot \langle \boldsymbol{\omega}, \mathbf{n} \rangle} \cdot \exp\left(-b \cdot \langle \mathbf{c} - \mathbf{o}, \mathbf{n} \rangle\right) \cdot \left(1 - \exp\left(-b \cdot D \cdot \langle \boldsymbol{\omega}, \mathbf{n} \rangle\right)\right),$$

which can now be used to compute the integrated densities P.

4.3.3 Spherical medium

By the *spherical medium* we understand a medium where the density is accumulated in a sphere (Figure 4.3). Let $R > 0$ denote the radius of the sphere. Let us derive the equations for a situation where the density quadratically decreases outwards the center $\mathbf{o} \in \mathbb{R}^3$ of the sphere. Analytical integration of this density function has been derived, for example, by Quílez [2015].

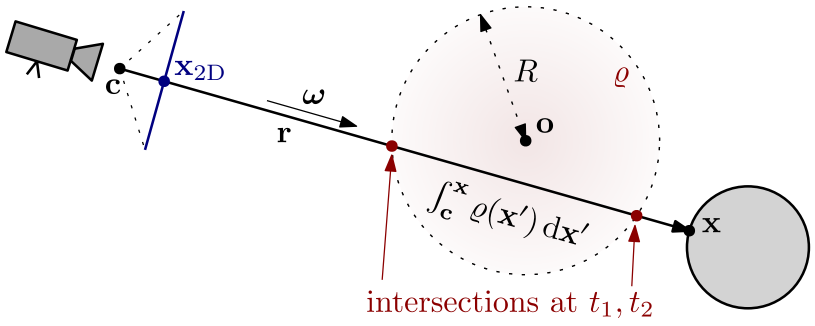
The density $\varrho(\mathbf{x}')$ depends on our arbitrary factor $\varrho' \geq 0$ and the distance from the center. It is important to consider the points outside of the sphere: the density there is zero. We can therefore express the density as:

$$\varrho(\mathbf{x}') = \begin{cases} \varrho' \cdot \left(1 - \frac{\|\mathbf{x}' - \mathbf{o}\|^2}{R^2}\right) & \text{if } \|\mathbf{x}' - \mathbf{o}\| \leq R, \\ 0 & \text{otherwise.} \end{cases} \tag{4.15}$$

Identically to the exponential medium, we use the parametric equation:

$$\mathbf{r}(t) = \mathbf{c} + t \cdot \boldsymbol{\omega}, \qquad 0 \leq t \leq D. \tag{4.16}$$

The density function (Equation 4.15) is expressed differently for two cases. In order to integrate the density, we have to first compute where the camera ray intersects with our sphere. This will be the bounds of the definite integral.



$\mathbf{x}_{2\mathrm{D}}$
$\mathbf{c}$
$\boldsymbol{\omega}$
$\mathbf{r}$
R
ϱ
$\mathbf{o}$
$\int_{\mathbf{c}}^{\mathbf{x}} \varrho(\mathbf{x}')\, d\mathbf{x}'$
$\mathbf{x}$
intersections at t_1, t_2

Figure 4.3: Integrating the density of a spherical medium.

Finding the intersections

All points *on* the sphere have the same distance R from the center $\mathbf{o}$. To obtain the intersections of the ray and the sphere, we can solve the following equation for the parameter t; $0 \leq t \leq D$:

$$\begin{aligned} \|\mathbf{r}(t) - \mathbf{o}\| &= R, \\ \|\mathbf{c} + t \cdot \boldsymbol{\omega} - \mathbf{o}\| &= R, \\ \|t \cdot \boldsymbol{\omega} + (\mathbf{c} - \mathbf{o})\| &= R, \end{aligned}$$

by squaring both sides and expanding the Euclidean distance, we get:

$$\|t \cdot \boldsymbol{\omega}\|^2 + \|\mathbf{c} - \mathbf{o}\|^2 + 2\langle t \cdot \boldsymbol{\omega}, \mathbf{c} - \mathbf{o}\rangle = R^2,$$

then because $\boldsymbol{\omega}$ is normalized, we have $\|\boldsymbol{\omega}\| = 1$, and by using the bilinearity of the inner product, we get:

$$t^2 + \|\mathbf{c} - \mathbf{o}\|^2 + 2t\langle \boldsymbol{\omega}, \mathbf{c} - \mathbf{o}\rangle = R^2, \tag{4.17}$$

which after rearranging the terms gives us:

$$t^2 + 2t \underbrace{\langle \boldsymbol{\omega}, \mathbf{c} - \mathbf{o}\rangle}_{B} + \underbrace{\|\mathbf{c} - \mathbf{o}\|^2 - R^2}_{C} = 0, \tag{4.18}$$

and that is a standard quadratic equation for t.

We can solve Equation 4.18 with the standard procedures for solving quadratic equations. We obtain the values t_1, t_2, which are the parameters of the ray where it intersects with the sphere:

$$t_{1,2} = -B \pm \sqrt{B^2 - C}.$$

Obviously, if $B^2 - C < 0$, the equation does not have a real solution because the ray does not intersect the sphere. Then because $0 \leq t \leq D$, we define the values $t_1 \leq t_2$ as follows:

$$\begin{aligned} t_1 &= \max\left\{0, -B - \sqrt{B^2 - C}\right\}, \\ t_2 &= \min\left\{D, -B + \sqrt{B^2 - C}\right\}. \end{aligned}$$

Integrating the density

Now that we have the bounds t_1, t_2, we can integrate the density function (Equation 4.15). We use the equation of the ray (Equation 4.16) within the bounds:

$$\begin{aligned} \mathsf{P}(\mathbf{x}_{\text{2D}}) &= \int_{\mathbf{c}}^{\mathbf{x}} \varrho(\mathbf{x}')\,\mathrm{d}\mathbf{x}' \\ &= \int_{t_1}^{t_2} \varrho\left(\mathbf{r}(t)\right)\mathrm{d}t \\ &= \int_{t_1}^{t_2} \varrho(\mathbf{c} + t \cdot \boldsymbol{\omega})\,\mathrm{d}t \\ &= \varrho' \int_{t_1}^{t_2} \left(1 - \frac{\|\mathbf{c} + t \cdot \boldsymbol{\omega} - \mathbf{o}\|^2}{R^2}\right)\mathrm{d}t \\ &= \varrho' \int_{t_1}^{t_2} \frac{R^2 - \|\mathbf{c} + t \cdot \boldsymbol{\omega} - \mathbf{o}\|^2}{R^2}\,\mathrm{d}t. \end{aligned}$$

We now expand the Euclidean distance again similarly to the steps that gave us Equation 4.17. By using the values B, C from the previous part, we get:

$$\begin{aligned}
\mathsf{P}(\mathbf{x}_{2\mathrm{D}}) = \dots &= \varrho' \int_{t_1}^{t_2} \frac{R^2 - t^2 - \|\mathbf{c} - \mathbf{o}\|^2 - 2t\langle \boldsymbol{\omega}, \mathbf{c} - \mathbf{o} \rangle}{R^2} \,\mathrm{d}t \\
&= -\varrho' \cdot R^{-2} \cdot \int_{t_1}^{t_2} \left(t^2 + 2Bt + C\right) \mathrm{d}t \\
&= -\varrho' \cdot R^{-2} \cdot \left[\frac{t^3}{3} + Bt^2 + Ct\right]_{t_1}^{t_2} \\
&= -\varrho' \cdot R^{-2} \cdot \left(\frac{t_2^3 - t_1^3}{3} + B\left(t_2^2 - t_1^2\right) + C\left(t_2 - t_1\right)\right).
\end{aligned}$$

Quílez [2015] suggests to multiply the result by $\frac{4}{3}R$ to normalize the density distribution. We can include this normalization factor into our density factor ϱ'.

4.3.4 Summary

We have successfully derived the equations for calculating P in three types of media. Figure 4.4 shows how they may look in our demo scene. It is possible to simulate very diverse conditions using just these three functions, because they offer a lot of parameters. Spherical media can be, for example, used for local effects such as smoke. Rotated exponential media can simulate gust fronts (visible boundaries of wind flow) in sandstorms.

Our list of analytically integrable media is obviously not exhaustive. We could try to derive the equations for more density functions, but that is certainly not the goal of this thesis. When trying to integrate a new medium, it is useful to remember that we can parametrize the camera ray $\mathbf{r}(t)$. This approach has already helped us in the two heterogeneous cases.

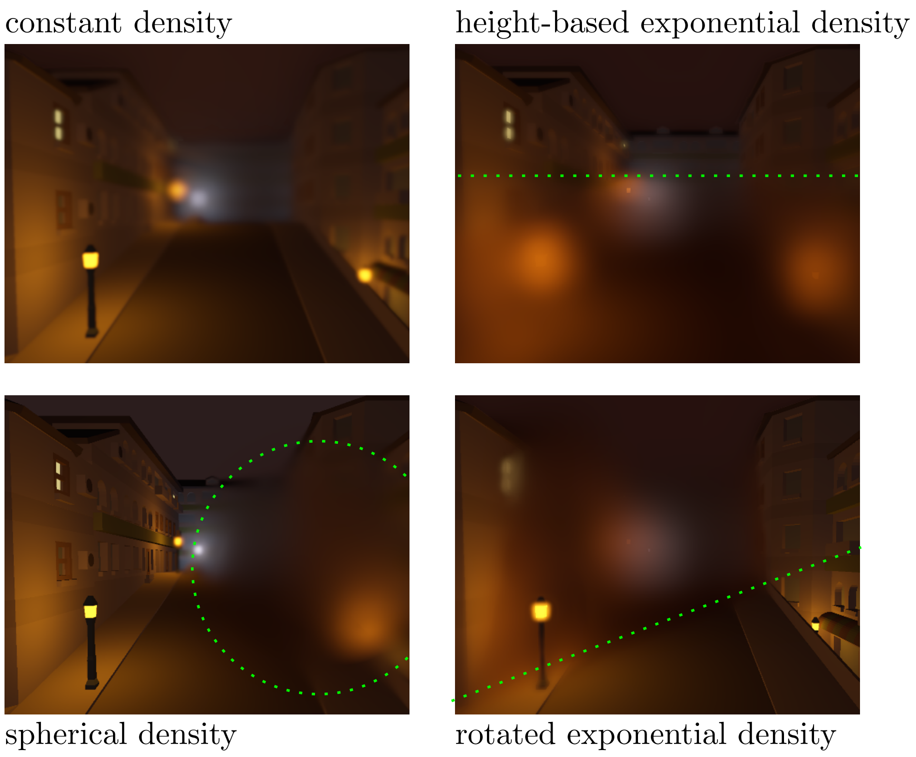


Figure 4.4: Renders of different density functions with highlighted boundaries.

4.4 Filtering and compositing

The goal of the filtering step is to blur the scattered radiance L_{sc} according to the spreading widths W of the pixels. Let us now explain in details how we solve the problem in our method to reduce the inherent visual artifacts. At the end of this section, we then explain how to compose the final output image L' together with the emissive term.

4.4.1 Naive approaches

Naive gathering approach

We assume that the blurring of the pixels is Gaussian. It means that every individual pixel should be blurred according to a certain Gaussian PSF. In Section 3.1.3, we already explained the approach to the filtering by using the splatting and gathering algorithms. Unfortunately, their time complexity is too high, so we refer to these approaches as *naive*.

Because we implement our method on a GPU, we prefer gathering to splatting as explained in Section 1.4.6. We use this gathering approach for comparison with our more efficient filtering, similarly to Elek et al. [2013]. Therefore, we can understand the gathering approach as our reference to a certain degree.

Naive MIP map approach

Because we assume Gaussian blurring, we can certainly use the efficient MIP map filtering described in Section 3.2. It requires us to build a simple Gaussian MIP chain $\mathsf{L}_{\text{sc}}^{[0...K]}$ according to Equation 3.6 (Figure 4.6a). The final blurred image L'_{sc} is then composed by using the spreading widths W in Equation 3.9.

As we can see in Figure 4.5, the simple MIP map approach suffers from a lot of visual artifacts. Therefore, we refer to this approach as *naive* as well. We devote the following subsections to identifying the artifacts and proposing improvements. Our goal is to reduce at least some of the artifacts by using more clever filtering.

4.4.2 Luminance weighting

Illumination leaking

Imagine a very bright object with a low standard deviation W, e.g., the yellow lantern located closer to the camera in Figure 4.5b. If there is a dark object in the background and its standard deviation W is much higher, the luminance of the bright pixels *leaks* into their neighborhood.

It occurs because when building the MIP chain, the luminance of the bright object is propagated into high MIP levels, even though the standard deviation of the object is small. When fetching the colors for the dark background, the bright luminance incorrectly leaks into the dark area.

This problem is called the *illumination leaking* [Elek et al., 2013] and can be seen in Figure 4.5bc as the huge yellow blur of the lantern. Notice that the blur in the gathering solution is much more isolated (Figure 4.5f).

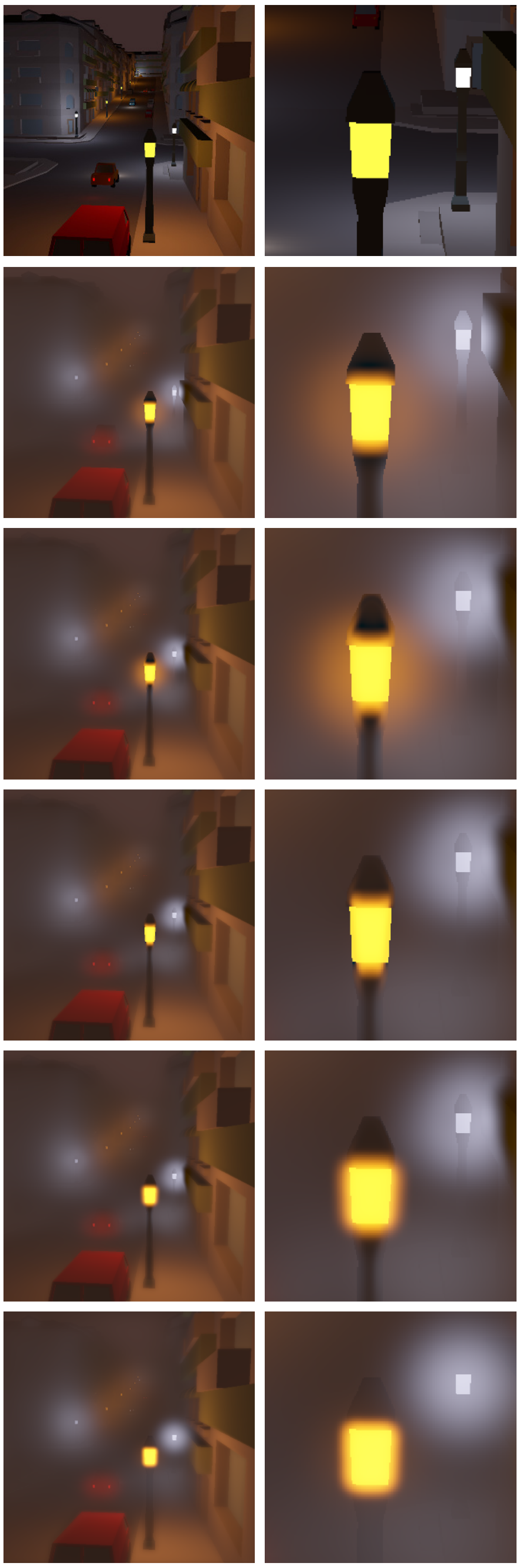


Figure 4.5: Comparison of filtering the input image (a) with different approaches. The naive MIP map filtering (b) suffers from very sharp discontinuities. By blurring the depth (c), we can eliminate them. The luminance weighting (d) prevents incorrect light leaking. The pixel separation (e) correctly blurs bright pixels close to the camera. The gathering algorithm (f) can be understood as a reference, but it is too slow for our real-time use.

Masking idea

In order to suppress the illumination leaking, we certainly need to stop the pixels with low W from propagating to unnecessarily high MIP levels (Figure 4.6b). The idea presented by Elek et al. [2013] is based exactly on this approach.

When building the MIP chain, we introduce a new auxiliary mask M. Let $\mathcal{G}$ denote our discrete Gaussian filter as introduced in Section 3.2. The MIP chain for $k > 0$ is built as:

$$\mathsf{L}_{\text{sc}}^{[k]} = \left(\mathsf{M}^{[k]} \cdot \mathsf{L}_{\text{sc}}^{[k-1]}\right) * \mathcal{G}. \tag{4.19}$$

The mask M can be defined for the level k as:

$$\mathsf{M}^{[k]} = \text{smoothstep}\left(T, (1+\varepsilon)\cdot T, \mathsf{W}^{[k-1]}\right), \tag{4.20}$$

where T and ε control the masking threshold distance and width.

The function smoothstep(e_1, e_2, x) is a smooth interpolation function, where $e_{1,2}$ are left and right edges, respectively. The function returns 0 for $x \le e_1$ and 1 for $x \ge e_2$. For $e_1 < x < e_2$, the returned value is a smooth interpolation between 0 and 1.

Because the masking should correspond to the relation between W and k, we set $T = c \cdot 2^{k-1}$ (Equation 3.7, Elek et al. [2013]). The width $\varepsilon \ge 0$ should be set according to the scene to get the visually best results. When experimenting with our demo scene, we found that changing ε can be noticed when moving around the scene. Very low values, such as $\varepsilon < 0.5$, caused high flickering of the blurring. We managed to slightly reduce the flickering by using $\varepsilon = 2$.

Spread-space MIP chain

The auxiliary mask M in Equation 4.20 requires us to also build the pyramid $\mathsf{W}^{[0\ldots K]}$, where $\mathsf{W}^{[0]} = \mathsf{W}$. This MIP chain should correspond to the standard deviation W averaged for neighboring pixels. That is because we want to mask the pixels according to W in their neighborhoods.

We weight the pixels according to their luminance, or brightness. Therefore, in this case, we do not perform Gaussian blurring, but we use a weighted average as proposed by Elek et al. [2013]:

$$\mathsf{W}^{[k]} = \frac{\left(\mathsf{Y}^{[k]} \cdot \mathsf{W}^{[k-1]}\right) * \mathcal{U}}{\mathsf{Y}^{[k]} * \mathcal{U}}. \tag{4.21}$$

The average is obtained by a convolution with the uniform distribution $\mathcal{U}$ of the same size as $\mathcal{G}$. Convolution with a uniform distribution like in Equation 4.21 is equivalent to averaging the values.

The average is weighted by the luminance of the pixels, hence we call this step the *luminance weighting*. It means that we prefer the standard deviation of brighter pixels. That is because, as we have already explained, we primarily want to mask the bright pixels. The auxiliary weighting mask Y is therefore obtained by computing the absolute RGB luminances y:

$$\mathsf{Y}^{[k]} = y(\mathsf{L}_{\text{sc}}^{[k-1]}).$$

We have now successfully built the MIP pyramid $\mathsf{L}_{\text{sc}}^{[0\ldots K]}$ and we are ready to fetch the filtered values.

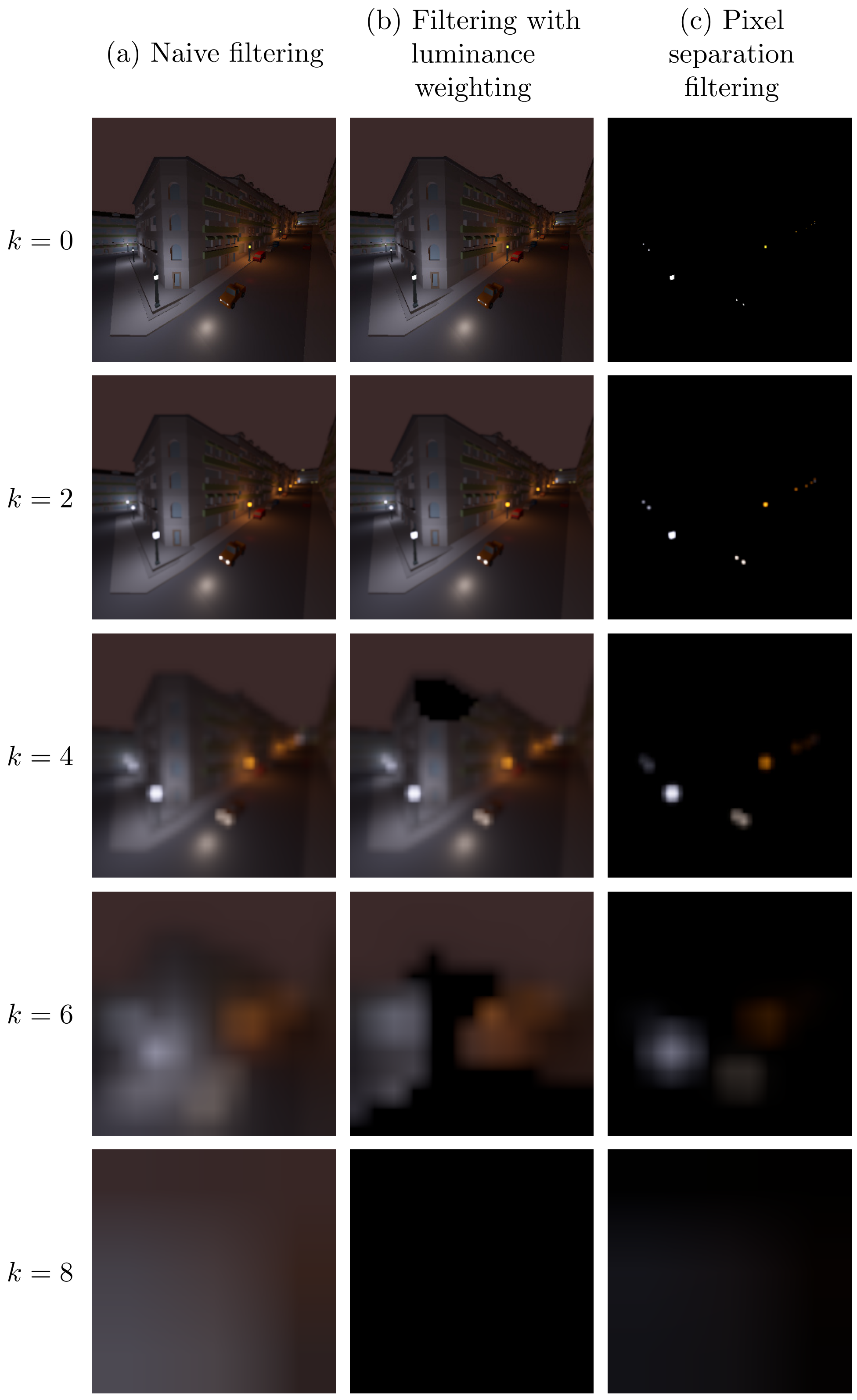

(a) Naive filtering
(b) Filtering with luminance weighting
(c) Pixel separation filtering
$k = 0$
$k = 2$
$k = 4$
$k = 6$
$k = 8$

Figure 4.6: MIP chains of the filtering step at different levels k. The naive filtering (a) blurs all pixels with the same weight. Luminance weighting (b) stops propagation of the pixels whose spreading width is too low. The pixel separation (c) blurs bright pixels separately while weighting their depths.

4.4.3 Fetching and depth blurring

We now need to compute the correctly blurred image L'_{sc}. As we have already explained, the naive algorithm simply fetches the values according to Equation 3.9 with the rescaled depths $d' \in \mathsf{D}'$. The corresponding level ℓ for each pixel can be therefore expressed similarly to Equation 3.8 as:

$$\ell(d') = \operatorname{clamp}\left(\log_2 \frac{W(d')}{c}, 0, K\right). \tag{4.22}$$

Depth blurring

Unfortunately, as can be seen in Figure 4.5b and as explained by Elek et al. [2013], the result computed this way may suffer from noticeable discontinuities. Typically, the depths D and D' can be very different even for directly neighboring pixels because of the geometry in our scene.

In this case, we can eliminate the discontinuities by blurring the depths. Therefore, we apply the Gaussian MIP map blurring to our rescaled depths D'. Let us build another MIP chain $\mathsf{D}'^{[0\ldots K]}$, where $\mathsf{D}'^{[0]} = \mathsf{D}'$ and for $k > 0$, we have:

$$\mathsf{D}'^{[k]} = \mathsf{D}'^{[k-1]} * \mathcal{G}.$$

Fetching from blurred depths

The idea is that we select the corresponding level ℓ according to the blurred rescaled depth. Elek et al. [2013] further proposes to do this process twice as the size of the blurring of D' should be proportional to the size of the blurring of L_{sc}. We therefore express the filtered image as:

$$\begin{aligned} \ell' &= \ell\left(\mathsf{D}'^{[\ell(\mathsf{D}')]}\right), \\ \mathsf{L}'_{\text{sc}} &= \mathsf{L}'^{[\ell']}_{\text{sc}}. \end{aligned} \tag{4.23}$$

By using this approach, we achieved the result in Figure 4.5d.

4.4.4 Pixel separation

We have managed to prevent the illumination leaking by using the luminance weighting and to eliminate image discontinuities by using the depth blurring. Unfortunately, by introducing these two techniques together, we have amplified another problem that we now explain.

Bright pixels

In HDR images, certain pixels may be much brighter than the rest of the image. In Figure 4.5d, the yellow lantern is very bright and we have successfully prevented the leaking of its color by using the luminance weighting mask. On the other hand, our prevention of the leaking has caused the lamp to not blur correctly compared to the gathering algorithm. Let us now explain why the error occurred.

Basically, the depth $d'_1 \in \mathsf{D}'$ of the bright lantern is much lower than the depth $d'_2 \in \mathsf{D}'$ of the neighboring dark background. In this case, when $d'_1 \ll d'_2$, the depth blurring step obviously smoothes this large discontinuity. This is the exact reason why we have introduced the blurring step in the first place: to eliminate the discontinuities by blurring. Therefore, in case the dark background is bigger than the bright foreground, the smoothing can completely eliminate d'_1 from high levels of $\mathsf{D}'^{[0\ldots K]}$.

Now when we fetch the background pixels from the MIP chain according to Equation 4.23, we obviously fetch from a high level because the depth d'_2 of the background is very high. Unfortunately, because of the luminance weighting, the high levels do not contain the blur of the bright foreground, because we masked it during the luminance weighting step. The bright pixels in our case have low depth, so their blur does not get propagated to the high levels. It essentially means that the dark area of our image remains dark and the blur of the bright foreground is "cut" at the edge. This is exactly what happened in Figure 4.5d to the yellow lantern.

Separation

As explained, the incorrect blurring occurs for very bright pixels in relatively low depths. One of the possible solutions would be to change our depth blurring algorithm, e.g., by stating that bright pixels are more important.

However, by preferring the depths of bright pixels, their dark neighborhood would be fetched from an incorrect level. Essentially, the blurring width of the bright object would be correct, but we would cause a noticeable discontinuity on the edge of the blur. It does not really seem to be possible to solve our contradictory needs by saying "we should prefer those pixels over the others".

Instead, we introduce a different heuristic technique. Again, we want to solve the incorrect blurring of bright pixels in low depths. Therefore, we decide to completely *separate* these pixels from the original input L (Figure 4.6c).

For this purpose, we introduce a new auxiliary mask M and we split the input L into two different scattering images. Let L_{sc} denote the scattering image without the separated pixels and $\mathsf{L}_{\mathrm{ssc}}$ the scattering with the separated pixels:

$$\begin{aligned} \mathsf{L}_{\mathrm{ssc}} &= \mathrm{e}^{-C_\mathrm{a}\cdot\mathsf{P}} \cdot \left(1 - \mathrm{e}^{-C_\mathrm{s}\cdot\mathsf{P}}\right) \cdot \mathsf{M} \cdot \mathsf{L}, \\ \mathsf{L}_{\mathrm{sc}} &= \underbrace{\mathrm{e}^{-C_\mathrm{a}\cdot\mathsf{P}} \cdot \left(1 - \mathrm{e}^{-C_\mathrm{s}\cdot\mathsf{P}}\right)}_{\text{original weight of } \mathsf{L}_{\mathrm{sc}}} \cdot (\mathsf{1} - \mathsf{M}) \cdot \mathsf{L}. \end{aligned} \tag{4.24}$$

The image $\mathsf{1}$ denotes an image with all pixels set to 1. Notice that the sum $\mathsf{L}_{\mathrm{ssc}} + \mathsf{L}_{\mathrm{sc}}$ is equal to the original L_{sc} from Equation 4.5. This means that we do not lose any information by introducing this step.

With regards to our intentions, we base the mask M on smoothly selecting the pixels with *high* absolute RGB luminance $y \in \mathsf{Y}^{[0]}$ and *low* depth $d' \in \mathsf{D}'^{[0]}$:

$$\mathsf{M} = \underbrace{\left(\mathrm{smoothstep}\left(T_y, T_y + \varepsilon_y, \mathsf{Y}^{[0]}\right)\right)}_{\text{high luminance}} \cdot \underbrace{\left(\mathsf{1} - \mathrm{smoothstep}\left(T_{d'} - \varepsilon_{d'}, T_{d'}, \mathsf{D}'^{[0]}\right)\right)}_{\text{low depth}}, \tag{4.25}$$

where the parameters define the thresholds. The minimum luminance we want to separate is denoted by T_y, the threshold width is ε_y. The maximum depth we want to separate is denoted by $T_{d'}$, the threshold width is $\varepsilon_{d'}$. These values have to be carefully set based on experiments for individual scenes and media combinations. Whether they give correct results can be verified by comparing to the gathering algorithm. The values we used in our demo scene are commented in Section 4.5.4.

Separation MIP chain

The image L_{sc} is filtered using the same luminance weighting technique we described in Section 4.4.2. Our pixel separation does not have any impact on the L_{sc} filtering as we only masked out the close bright pixels that we want to filter separately.

However, the MIP chain $\mathsf{L}_{\text{ssc}}^{[0...K]}$ is built differently, because the image is mostly black and only contains a few separated pixels. If we used the same filtering technique, we would block the luminance from propagating to higher MIP levels, which is certainly not what we want. Therefore, we use a modified approach.

Because most of the pixels in L_{ssc} are black, we are not interested in their depths at all. We would like to prioritize the depth of the bright separated pixels. Hence, we construct the new chain $\mathsf{D}'^{[0...K]}_{\text{ssc}}$ based on luminance weighted averaging:

$$\mathsf{D}'^{[k]}_{\text{ssc}} = \frac{\left(\mathsf{Y}^{[k]}_{\text{ssc}} \cdot \mathsf{D}'^{[k-1]}_{\text{ssc}}\right) * \mathcal{U}}{\mathsf{Y}^{[k]}_{\text{ssc}} * \mathcal{U}}, \tag{4.26}$$

where $\mathsf{D}'^{[0]}_{\text{ssc}} = \mathsf{D}'$ and the luminances $\mathsf{Y}^{[0...K]}_{\text{ssc}} = y(\mathsf{L}^{[0...K]}_{\text{ssc}})$. It is exactly the same principle we used in Equation 4.21 for building $\mathsf{W}^{[0...K]}$.

It remains to explain how to build $\mathsf{L}^{[0...K]}_{\text{ssc}}$. As we have already mentioned, we do not want to use luminance weighting, because we obviously need our separated pixels to propagate. Unlike in Equation 4.19, we only need to build the Gaussian MIP chain without any further masking:

$$\mathsf{L}^{[k]}_{\text{ssc}} = \mathsf{L}^{[k-1]}_{\text{ssc}} * \mathcal{G}, \tag{4.27}$$

where $\mathsf{L}^{[0]}_{\text{ssc}} = \mathsf{L}_{\text{ssc}}$. This is not any different from Equation 3.6.

Result

We now build the blurred image L'_{ssc}. The process is similar to building L'_{sc} (Equation 4.23), except we need to take special care of choosing the correct level. Our depth chain $\mathsf{D}'^{[0...K]}_{\text{ssc}}$ is built based on luminance weighted averaging and it is not exactly clear which level contains the correct depths.

Unfortunately, this heavily depends on how many pixels were actually separated in our scene. We have to introduce another arbitrary parameter k_{ssc} ($0 \le k_{\text{ssc}} \le K$) based on experimenting. For our demo scene, we achieved satisfying results with higher values close to K, e.g., $k_{\text{ssc}} = 0.7 \cdot K$. The fetching equation is:

$$\begin{aligned} \ell' &= \ell\left(\mathsf{D}'^{[k_{\text{ssc}}]}_{\text{ssc}}\right), \\ \mathsf{L}'_{\text{ssc}} &= \mathsf{L}^{[\ell']}_{\text{ssc}}. \end{aligned} \tag{4.28}$$

Figure 4.5e shows how we can correctly blur the bright yellow lantern by separating its color in L_{ssc}.

4.4.5 Final compositing

The final step of our method is to generate the correctly attenuated and scattered image L'. The result is based on three images that we have computed during the previous steps.

The attenuated radiance L_{at} corresponds to the radiance that reaches the camera without any interactions. The blurred radiances $\mathsf{L}'_{\mathrm{sc}}$ and $\mathsf{L}'_{\mathrm{ssc}}$ together represent the radiance that is scattered on the way to the camera. The radiance that is absorbed is not included in any of the three images.

Therefore, the final output is obtained by simply adding the results together:

$$\mathsf{L}' = \mathsf{L}_{\mathrm{at}} + \mathsf{L}'_{\mathrm{sc}} + \mathsf{L}'_{\mathrm{ssc}}.$$

4.4.6 Emissive term

When compositing the output image, we can also add an approximated emissive term to the final result (Figure 4.9). Let us suppose that the participating medium emits the radiance L_{e} towards the camera from each point of the medium. Therefore, for each camera ray, we can integrate the emitted radiance from every point along the ray while attenuating it according to the Beer–Lambert law.

Our assumption that L_{e} is constant is obviously not a good approximation for heterogeneous media where the particle density varies from point to point. Also, in order to integrate the attenuated emission along the camera ray, we would have to integrate the optical thickness between the camera and each emission point individually to correctly compute the transmittance. However, that would result in an integral inside an exponential function inside another integral. The complexity of the equation for heterogeneous media would be too high.

To simplify our situation, we can use our rescaled distances D' (Section 4.2.3). When introducing the rescaling idea, we mentioned that the distances D' enable us to approximate heterogeneous media by homogeneous ones. In case of the emission, we assume that the radiance L_{e} is constant along the rescaled distances and the attenuation is computed by using our approximation $\mathsf{D}' \approx \mathsf{P}$.

We integrate the emissions as if we were working with a homogeneous medium. Let L_{em} denote the additional emissive image. For the camera rays of the lengths $\mathsf{D}' \approx \mathsf{P}$, we integrate the emissions occurring along the ray and we attenuate them as they travel towards the camera:

$$\mathsf{L}_{\mathrm{em}} = L_{\mathrm{e}} \int_0^{\mathsf{D}'} \underbrace{\exp\left(-C_{\mathrm{t}} \cdot d'\right)}_{\text{attenuation}} \mathrm{d}d' = L_{\mathrm{e}} \cdot \frac{1 - \exp\left(-C_{\mathrm{t}} \cdot \mathsf{D}'\right)}{C_{\mathrm{t}}}. \tag{4.29}$$

Then we can add this emissive term to the final output:

$$\mathsf{L}' = \mathsf{L}_{\mathrm{at}} + \mathsf{L}'_{\mathrm{sc}} + \mathsf{L}'_{\mathrm{ssc}} + \mathsf{L}_{\mathrm{em}}. \tag{4.30}$$

4.5 Results and their limitations

After describing how to obtain the output images, let us now have a look at a few examples. Figure 4.7 shows two sets of input and output images filtered using our method. Both output images are screenshots from our real-time application.

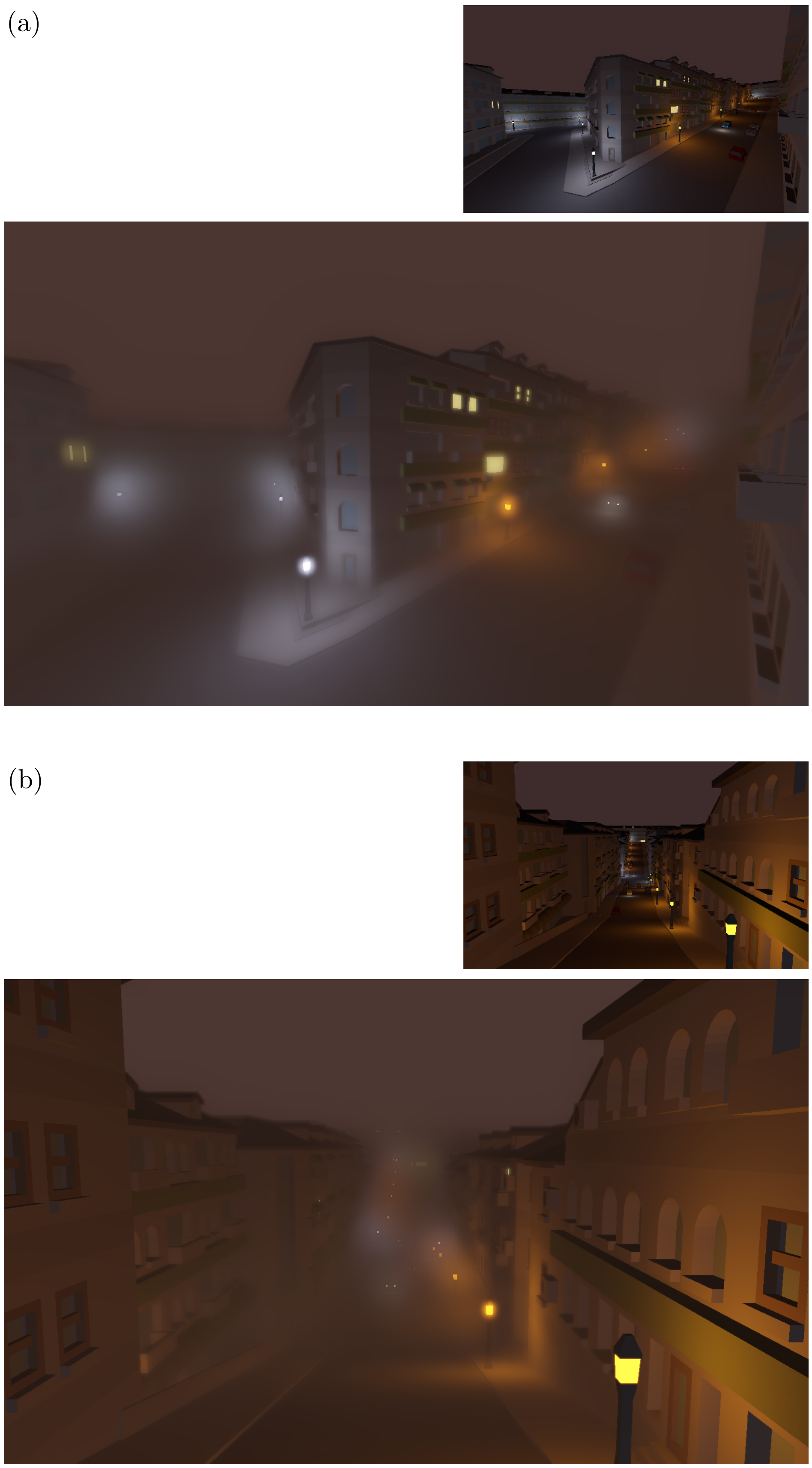


Figure 4.7: Examples of the typical outputs of our method. Above each output image there is its corresponding input.

4.5.1 Advantages over competing methods

The general goals of our method were already introduced in Section 4.1.1. Let us now briefly summarize what exactly the advantages of our method are over the competing solutions.

Blurring the whole scene

As we can see, our method is capable of blurring the geometry in our scene, which is a major improvement over all empirical and single-scattering methods from Chapter 2.

We can also handle light scattering from an arbitrary number of light sources and our time complexity still remains the same, that is $\mathcal{O}(n^2 \log_2 n)$ for the resolution $n \times n$ as explained in Section 3.2.5. This is an advantage over the solutions that can only handle a single source (Section 2.3) or have to compute the contribution from each source individually (Section 2.4).

Various non-homogeneous media

Both images in Figure 4.7 show non-homogeneous media with exponential densities. When compared to the homogeneous results of Elek et al. [2013], we can see major visual differences in our approach. It is especially interesting how the roofs of the buildings are blurred much less than the road, which alters the depth perception of the scene: it looks like the buildings are much taller than they actually are. This effect can be very useful for video games and other visualizations where the artists often want to influence how viewers perceive the scene.

Furthermore, various parameters of the density functions can be changed. Rotated exponential media can be used to simulate sandstorms, which are usually very dense inside the sand cloud but the density outside can be marginal. Spherical media are useful for local effects. Fire, for example, could be simulated by placing an intense animated light source inside the spherical medium.

Intense light sources

By introducing our new pixel separation step, we have also significantly improved the quality of the filtering of HDR images with very intense lights. This is important for rendering night scenes, where the light sources can be arbitrary intense and placed at various locations in the scene.

Performance

Finally, the performance of our method seems to be much better than the performance of the competing multiple-scattering method by Shinya et al. [2016] (Section 3.3). We can process a single HD image in a few milliseconds. More details about our performance are mentioned later in Section 5.4.

After briefly explaining the advantages over the competing methods, we would now like to thoroughly describe the inherent limitations. This method and the competing methods are based on a lot of approximations that we have explained in the previous sections and chapters. So far, we have not yet described what exactly these approximations mean in the terms of the quality of the results.

4.5.2 Limitations caused by incomplete inputs

Our proposed method is based on filtering input images. All information that the method has available have to be stored in the inputs. As our only inputs are the rendered image L, the depths of the pixels D, and parameters of the camera and the medium, the method does not know any other information about the scene.

Visibility and flickering

The limitation that is common for all screen-space approaches, in our case primarily for our method and the methods described in Chapter 3, is that the objects that we want to process have to be directly visible to the camera. As we can see in Figure 4.8, when a certain object disappears from the input image, we cannot correctly blur it, because we are missing the information in the pixels.

This may cause flickering of light sources when we use our method for rendering interactive scenes where the objects and the camera can move. It is also noticeable for distant light sources that only take a small portion of the screen, e.g., only 1 pixel that can disappear and appear again when moving the camera. In that case, the blur from the single pixel may be huge, so flickering occurs.

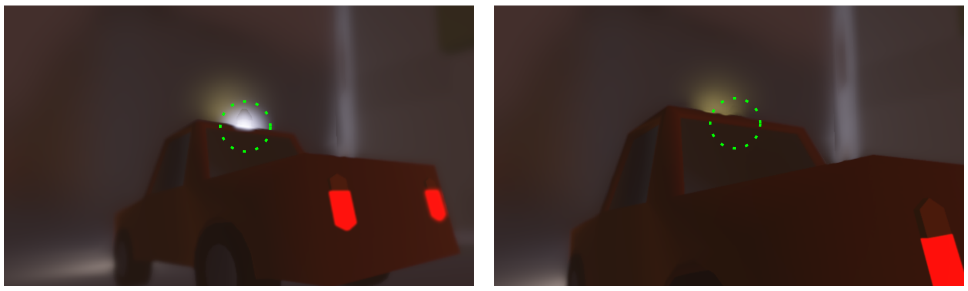


Figure 4.8: Objects that are not directly visible on the screen do not get blurred. The left picture shows a white glow around a lantern. The glow has disappeared after we moved the camera a little bit (right picture).

Light interactions

Another simplification is that we only simulate the light interactions that occur between the rendered pixels and the camera. In order to correctly simulate the light transport, we would also have to attenuate and scatter the radiance even before it reaches the objects and reflects towards the camera.

In that case, our input image L would already have to be rendered with considering the participating medium. This is not the case of our method nor the methods from Chapter 3 as we generally assume standardly rendered inputs.

Ambient

Furthermore, the overall atmosphere of the scenes may be influenced by the total environmental illumination caused by all light sources together. For example, if all major light sources in our scene are yellow, the whole participating medium may also appear to have a yellowish color with regards to the scattering and attenuation parameters.

As described by Elek et al. [2013], we can simulate the ambient color by adding an emissive term to the simulated medium. Figure 4.9 shows the same scene rendered with different emissive terms according to Section 4.4.6. We have also tried changing the color of the sky and together with the emissions, we can substantially change the overall atmosphere.

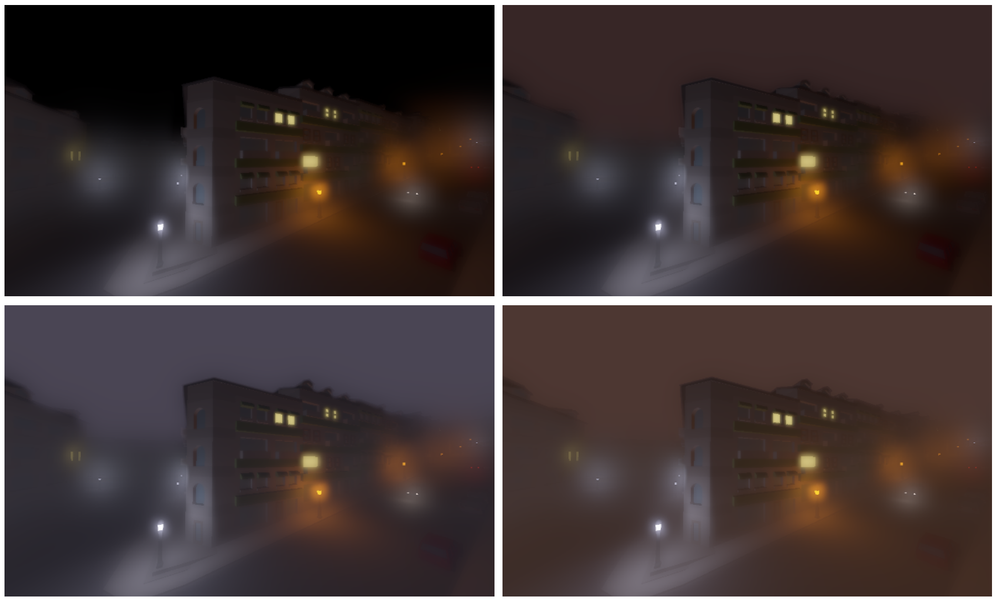


Figure 4.9: The same scene rendered with different sky colors and emissive terms. The upper images are rendered with a participating medium without emissions. The bottom images have slightly blue and yellow emissive terms respectively.

4.5.3 General simplifications of our approach

Occlusions

Similarly to the methods of Elek et al. [2013] and Shinya et al. [2016], our method ignores volumetric occlusions that happen on the way towards the camera after the radiance is reflected from surfaces. We cannot correctly render volumetric phenomena such as crepuscular rays (Section 2.3) and volumetric caustics.

If we wanted to take occlusions into account, we would have to introduce additional steps to the algorithm. In case of a limited number of light sources, we could sample the occlusions in screen-space similarly to Section 2.3. However, the correct solution would require us to sample the occlusions for each pixel individually. Again, by solving the problem in screen-space only, we would be limited to the information that are available in the input images.

Media simplifications

Since Section 1.2, we have assumed that all interactions happen in a single participating medium with identical evenly distributed particles. None of the methods presented in this thesis take mixed materials, such as milk in water, into account. Furthermore, the Gaussian PSF approximation by Premože et al. [2004] assumes spatially invariant parameters of the media.

We have introduced a very important approximation later in Section 4.2.3 by rescaling the distances and assuming that the heterogeneous media can be approximated by homogeneous media. In reality, the situation is obviously not that simple, because the multiple-scattering process depends on where exactly the particles are accumulated.

Imagine a medium with zero density below a light ray and high density above the light ray. Our approximation assumes the PSF to be circular, but it is obviously incorrect in this example, because no scattering can occur in zero particle density below the light ray.

Furthermore, similarly to all solutions presented in this thesis, we completely ignore phenomena such as refractions and dispersions [Elek, 2016]. Other phenomena caused by very small particles, e.g., Rayleigh scattering that explains why the sky appears to be blue [Shinya et al., 2016, Elek, 2016], are also not taken into account. More sophisticated methods would be required.

4.5.4 Gathering approach and MIP filtering

Comparison

Finally, we have introduced a huge approximation by the efficient MIP map filtering. We had to propose several heuristic improvements including luminance weighting, depth blurring, and pixel separation. How the pyramidal filtering looks compared to the gathering approach has already been shown (Figure 4.5).

But we should also note that the MIP map filtering may give visually more appealing results than the gathering approach. Compare, for example, the magnified images in Figure 4.5ef. The gathering approach blurred the white lantern with a perfect circular Gaussian PSF. The MIP map approach, on the other hand, preserved the edge of the building because it was closer to the camera. In this case, the edge of the building acts as an occlusion of the light and the MIP map result may look more realistic.

The major difference between the pyramidal filtering and the gathering approach is the following. The gathering approach assumes that all pixels are spread according to *their* PSF. The pyramidal approach approximates the spreading of other pixels by the depth of the sampled pixel. That is obviously incorrect, hence we had to introduce so many heuristic improvements to hide the artifacts.

Arbitrary parameters

In order to use our method, it is also necessary to correctly configure certain parameters. The scaling parameter c in our case was set to 0.86 as we got fairly close results compared to the gathering approach. The luminance weighting threshold width ε was mostly set to 2 to reduce flickering caused by high differences between MIP levels. Elek et al. [2013] mention that the value depends on the scene.

The pixel separation parameters T_y and ε_y were in our case set to 3. They generally depend on how intense the emissive materials are in the scene. The depth separation parameters $T_{d'}$ and $\varepsilon_{d'}$ depend on the parameters of the medium and the dimensions of the scene. In case of our demo scene and noticeably dense heterogeneous media, good visual results can be obtained by setting both parameters to 200 or 250. Finally, k_{ssc} was usually set to $0.7 \cdot K$ as the ideal level.

Visible artifacts caused by MIP filtering

Additionally to the already mentioned limitations, there are also artifacts that are MIP map filtering specific. First problem occurs when rendering *very distant objects* that take a large part of the screen, such as sky. Theoretically, the sky should be rendered in infinity. Obviously, if we depth blurred the infinitely far pixels of the sky, the blurring would smooth all depths to infinity. Therefore, we have to set a maximum depth and maximum density and clamp the values.

Another problem occurs when rendering objects at the *edge of the screen.* When building the MIP pyramid, the edge pixels have incomplete information about their neighborhoods and discontinuities may occur. Typically, if a very distant object, such as the already mentioned sky, is located at the edge of the screen, flickering may happen when moving with the camera. The easiest solution to this problem would be to crop the output images, i.e., use inputs with higher resolution than the output resolution.

We also have to comment on the *interpolation and filter sizes* (Figure 4.10). Generally, we should use the bicubic interpolation when fetching the pixels as it provides much smoother results, especially in motion. Unfortunately, even with bicubic interpolation, flickering may occur. With regards to the performance of our implementation (Section 5.4), the 4×4 filter with the linear-bicubic interpolation seems to be to best compromise between speed and quality.

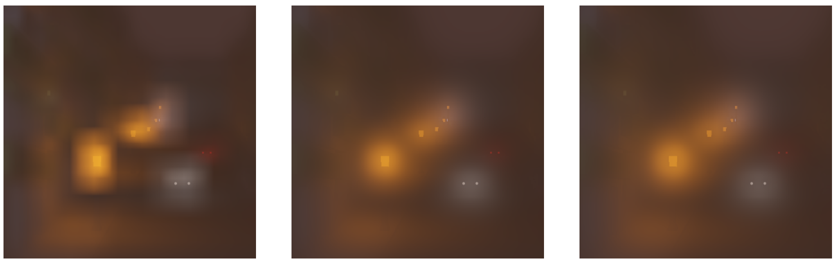

2×2 filter with linear-bilinear interpolation
4×4 filter with linear-bicubic interpolation
6×6 filter with cubic-bicubic interpolation

Figure 4.10: Comparison between different filter sizes and interpolation techniques. The two right pictures look almost identical. In higher resolutions, the 6×6 filter looks slightly smoother. The cubic-bicubic interpolation proved to be almost useless as it requires more fetches with unnoticeable visual improvement.

The last problem that we describe is caused by approximating the blurring widths by the level of the MIP chain. First, the parameters c and ε are only *constant approximations*, second, the MIP filtering can only build *a few meaningful levels* until we reach the resolution 1×1. Furthermore, the higher the level, the lower its resolution, which means less information are available. If the resolution of the input image is too low for the scattering, we cannot even build sufficiently many MIP levels.

Therefore, we cannot correctly render high standard deviations, because their information is stored in high levels that are either not available at all, or their resolution is too low. This problem is very noticeable for very high scattering parameters and can be compensated by increasing the absorption, which attenuates the problematic pixels.

5. Implementation

In the previous chapter, we proposed a solution for simulating light transport in participating media. In order to verify how our proposed method behaves and how our filtering steps improve the results over simpler approaches, we have implemented the method in a prototype demo application.

The implementation is an important part of this thesis, because it demonstrates the real behavior of our method. Furthermore, it proves that our solution is really capable of running real-time with sufficiently high FPS.

Section 5.1 is devoted to a brief overview of what exactly the output of our implementation is and how our demo scene looks like. In Section 5.2, we explain our rendering engine and the used technologies. Later, in Section 5.3, we describe the shaders, i.e., the code that is executed on the GPU. The specific implementation of our filtering is explained in this section as well. Finally, in Section 5.4, we analyze the performance of our application to verify that it is indeed sufficient for real-time purposes.

5.1 Overview

In order to demonstrate our method, we implement a real-time interactive demo application. It is capable of rendering scenes in a vacuum and then applying our filtering steps on the rendered images. During this process, the user is able to freely navigate around the scene, change various parameters of the medium, and configure the behavior of our method.

The user documentation for the demo application is available as Attachment 1 – User reference. The file paths mentioned in this chapter will be mostly relative to the `demo/` directory (see Attachment 2 – CD contents).

5.1.1 Demo scene

The inherent part of our implementation is the demo scene. We built our scene in the open source 3D creation suite Blender (Figure 5.1) as it is a common editor publicly available for free. Our scene depicts a static town street with buildings, vehicles, and lanterns. The scene is composed of a lot of small 3D objects that have been dedicated to the Public Domain by Kenney Group and Jim van Hazendonk.

We decided to demonstrate our light transport simulation at a night scene with many light sources such as lanterns and lights of the motor vehicles. The primary motivation for this decision was that the similar method by Elek et al. [2013] was only demonstrated with scenes with intense ambient light.

Furthermore, certain surfaces in our scene, such as the lanterns or the yellow windows of the buildings, have emissive colors. It means that radiance is emitted directly from their geometry. The motivation for emissive textures stems from the observation that the methods from Chapter 2 do not calculate illumination from these nontrivial light sources. We would therefore like to demonstrate that our solution is capable of blurring even intensely emissive surfaces.

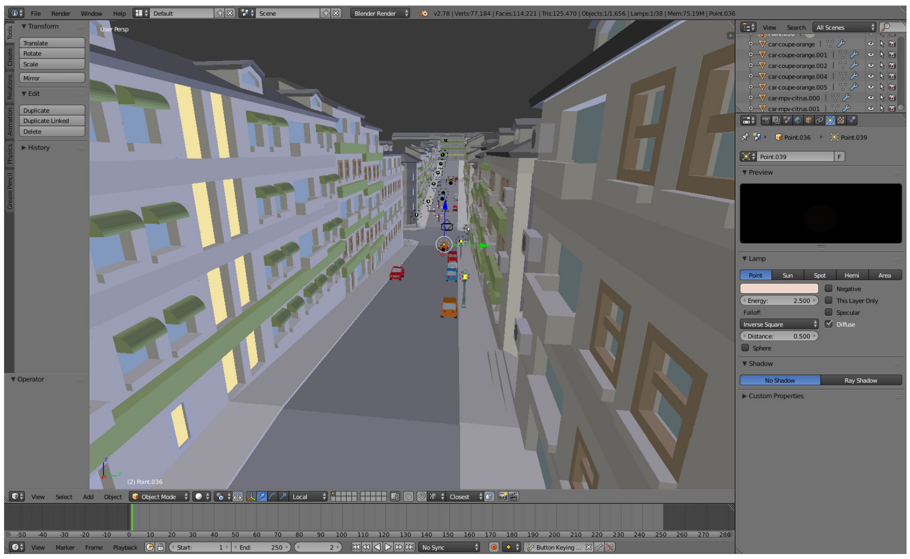
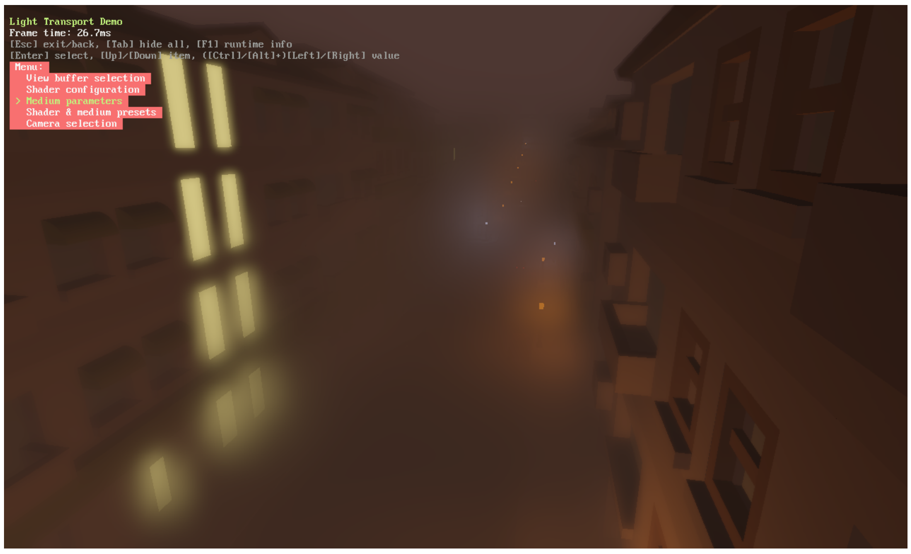


Figure 5.1: Demo scene in Blender. The yellow windows, car lights, and lanterns have emissive colors of their surface.



```
Light Transport Demo
Frame time: 26.7ms
[Esc] exit/back, [Tab] hide all, [F1] runtime info
[Enter] select, [Up]/[Down] item, ([Ctrl]/[Alt]+)[Left]/[Right] value
Menu:
   View buffer selection
   Shader configuration
 > Medium parameters
   Shader & medium presets
   Camera selection
```

Figure 5.2: Demo application displaying the scene and a menu.

5.1.2 Demo application

When implementing the demo, we could have used an existing modern rendering engine and only modify it by adding the additional filtering phase. However, at the time when I started with preparing the demo, I was not satisfied with the existing solutions, especially because many had a restrictive proprietary license.

On the other hand, I found the existing open-source solutions with unrestricted licenses to not have a satisfying documentation. This is especially critical because our method requires the use of advanced custom building of the MIP chains and the access to the MIP levels is not always documented or even available without modifying the original source code.

Therefore, I decided to prepare a completely new demo application (Figure 5.2) that is fully capable of importing the demo scene, rendering it in real-time in a vacuum, and then applying our method. The application also enables to view the different phases of the rendering pipeline and change the majority of the parameters of the light transport algorithm.

To avoid unnecessarily complicated rendering, the demo does not support any advanced features such as the rendering of shadows. With regards to this decision, I have also decided to give the scene a simple “cartoon” look without any detailed textures and without specular reflections.

The decision not to use an existing engine and therefore only support the necessary features should in no way harm the quality of the visualization of our method. The method is implemented in the same way as it would be for more advanced inputs, because it is based on the screen-space approach.

5.2 Application

Let us now have a look at how the demo application is implemented. This section is devoted to explaining the used technologies and the structure of the code that is executed on the central processing unit (CPU). The implementation of the code that runs on the GPU, i.e., the vertex and fragment shaders, is covered later in Section 5.3.

It should be noted that the final demo application has only been compiled for the Windows platform using the `builds/vs/LightTransportDemo.sln` project in Visual Studio 2015[1]. However, all technologies that we have decided to use should be fully compatible with other platforms including mobile phones. The decision not to compile the demo for other platforms was made because it would require additional testing of the application. Furthermore, the latest GPU drivers are usually targeted primarily to the Windows platform.

5.2.1 Technologies

The whole application is programmed in C++ and we use the modern features available in the latest C++11 and C++14 specifications, such as smart pointers, lambda functions, and range-based loops. The decision to use this programming language stems from the availability of cross-platforms compilers that can perform

[1] More details can be found in the electronic attachment, file `README-COMPILE.txt`.

very efficient optimizations of the compiled code. This is especially useful for real-time applications.

Scene loading

In order to load the scene from Blender, we first export it to the digital asset exchange (DAE) format and then load it using the cross-platform C++ library Assimp. This format and the library have been chosen after experimenting with various data formats. Assimp is able to extract the scene data from the DAE, such as the meshes and light sources, without losing any necessary information exported from Blender.

In case the imported scene contains textures, we need to load the image data from a texture file. We use the single file image loader called `stb_image`.

Rendering

The rendering itself is done via calling API functions of the bgfx library. This is a modern cross-platform graphics API library that supports multiple rendering backends including OpenGL and DirectX. This was one of the reasons why I have originally decided to use this library. Unfortunately, for our purposes, we only use the OpenGL backend of the library, because different errors occurred when using the DirectX backend on Windows.

Because of a bug in the bgfx MIP map API functions, we had to modify the source code of the original library. The code was only modified in the OpenGL part of the code, hence we do not use any other backend. The bug has been reported to the author of bgfx, but at the time of programming the application, the bug was still not fixed.

Together with the bgfx library, we also use SDL2. This cross-platform library handles the creation of the main application window. It also detects the mouse movement and the pressed keys when the window is focused. The communication between bgfx and SDL2 is handled transparently for us, the included source codes have been written by the author of bgfx.

When rendering objects in a 3D space, we often need to work with vectors and matrices. For this purpose, we use the glm library that only contains templated C++ header files. It enables us to work with vector and matrix classes such as `glm::vec3` and `glm::mat4`. The values in these classes can then be passed to bgfx as data pointers and bgfx correctly transports them to the GPU.

Saving and loading presets

Finally, in order to save and load various configurations (presets) of our demo application, we use the cross-platform serialization library called cereal. It is also a modern header-only C++ library that can be easily used for storing and loading data directly from and to C++ classes without any significant overhead. We have decided to use the JavaScript Object Notation (JSON) format for storing the data as it is supported by the standard version of cereal. Additionally, in order to find what preset files are stored in a directory, we use a `tinydir.h` header file.

5.2.2 Initialization and main loop

Entry

The main initialization procedure, i.e., the creation of the SDL2 window, is handled by the source codes in the `src/entry/` directory. This code was written as a part of bgfx and we have included it in our application. The `main` function is located in `src/entry/entry_sdl.cpp`.

Our own code is executed by bgfx in a new thread. For this purpose, we have the secondary main function called `_main_` in `LightTransportDemo.cpp`. The `LightTransportDemo` class is the main class of our demo application. It derives from `entry::AppI`, which is the interface that bgfx uses to communicate with our application from the window.

The application interface defines three important methods that we implement. The `init` method initializes our application and loads the resources. The `update` method runs in a loop and is where we render the scene. Finally, when the user decides to close the window, the `shutdown` method is called and we have to correctly unload the resources. The approach is illustrated in Figure 5.3.

Initialization

Let us now have a look at the behavior of `LightTransportDemo::init`. Even though the window has already been created when `init` is called, we still have to initialize the bgfx rendering backend—OpenGL in our case—and declare the format of our vertex data that we will send to the GPU.

The vertex data represent the meshes that we want the GPU to render for us. The vertex formats define how the data streams to the GPU look like, i.e., which bytes represent the position, the normal vector, the colors, etc. In our case,

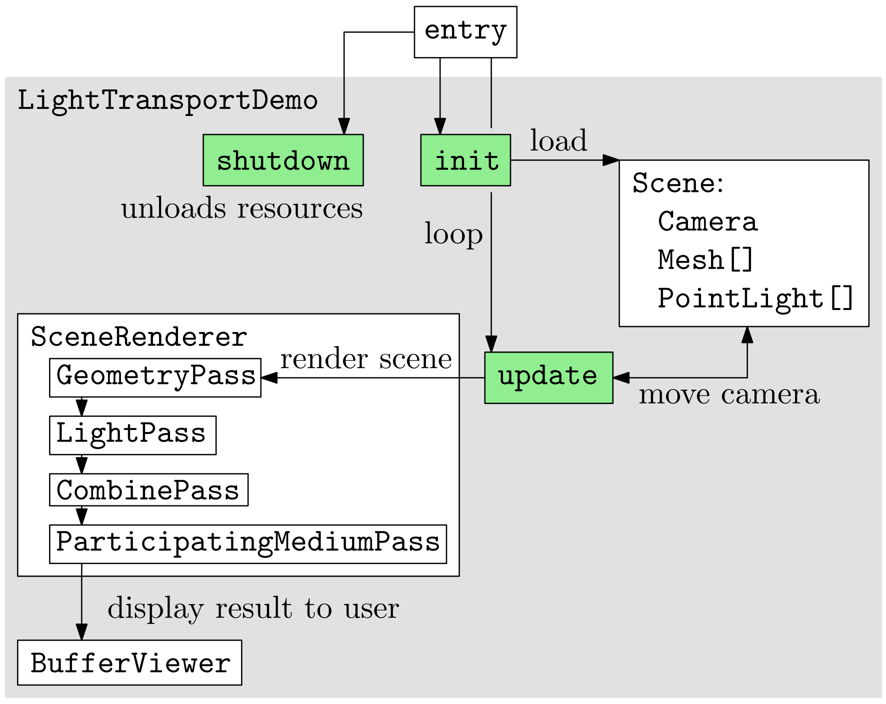


Figure 5.3: Simplified diagram of the application.

PositionNormalColorTextureVertex and PositionTextureVertex are the two formats that we use throughout the application.

After initializing bgfx, we load the scene in the DAE format. For this purpose, we use the class Scene (Section 5.2.3) that uses the API of Assimp to load all necessary data. Then we initialize the SceneRenderer (Section 5.2.4) that handles the creation of GPU buffers and the rendering of the scene in various phases, including the final filtering by our method. We also have to initialize the BufferViewer (Section 5.2.5) that displays the buffers to the screen so the user can see them. Finally, we set up the main menu.

Main loop

The main loop of the application is performed by the update method. Its main goal is to update the camera position and direction according to the state of the mouse and keys, then render the scene according to the new state, display the selected buffer to the user, and finally render the main menu over the screen. For this purpose, we use the classes that we have initialized in init and also other helping classes, such as CameraMovement that handles the mouse and key states to move the camera.

5.2.3 Scene representation

The representation of the scene is handled by the Scene class and additional complementary classes in the src/scene/ directory. For the purposes of our simple rendering engine, we do not need to store a whole graph of the scene objects (also called the *scene graph*). Instead, we represent the meshes, cameras, and point lights linearly in standard C++ vectors.

Each mesh is represented by the Mesh class that primarily stores the diffuse and emissive colors, and textures managed by the TextureLibrary. The class also stores handles to the GPU index buffer and vertex buffer. The necessary vertex data are copied to the GPU in the Mesh::loadFromScene method that loads the mesh from the Assimp representation aiMesh.

The cameras are represented by the Camera classes. They store the parameters of perspectives cameras, especially their positions in world-space coordinates and their directions. The class is also capable of computing the *view matrix* that represents the camera transformation for the GPU shaders. Each camera also has a field of view (FOV), which is implicitly set to 60°.

Each point light is represented by the PointLight class that stores the world-space position, color, and the attenuation coefficients. The attenuation factor of the light sources is determined by the parameters c_1, c_2, c_3 which denote the constant, linear, and quadratic attenuation terms respectively. For the distance d from the light source, the attenuation of the light is computed according to the following formula [Klawonn, 2008]:

$$f_{\text{att}}(d) = \min\left\{\frac{1}{c_1 + c_2 d + c_3 d^2}, 1\right\}. \tag{5.1}$$

Additionally, we also store the configuration of the participating medium in the scene. It is represented by the ParticipatingMedium class, which is merely a storage of various parameters, such as the absorption and scattering cross-sections.

5.2.4 Scene renderer

The rendering of the scene is handled by the `SceneRenderer` class and other classes in the `src/renderer/` directory. The scene renderer is basically a container of the four rendering passes that are illustrated in the diagram in Figure 5.4. Before explaining the passes, let us first have a look at the interface.

The scene renderer and the rendering passes share a similar interface: `init`, `shutdown`, `reset`, and `render`. The `init` methods are primarily intended for acquiring GPU resources, e.g., creating textures, loading the shader programs, and creating shader *uniforms* (parameters of the shader programs). The `shutdown` methods should properly unload the acquired resources, otherwise a memory error may occur when aborting the application.

When a resolution or the state of the bgfx backend changes, we may need to delete our textures and create them again with the current resolution. That is the major goal of the `reset` methods. Finally, the `render` methods are called from the main loop and should basically pass data to the vertex and fragment shaders (Section 5.3) and execute them.

The process we are using for the rendering is sometimes referred to as the *deferred rendering* [Klawonn, 2008], because we defer the actual shading to later phases. In our case, the rendering has three major phases, excluding the participating medium pass from Chapter 4.

Geometry pass

The first phase is the `GeometryPass` that primarily renders the scene meshes according to the camera view. This phase has five major outputs plus an additional non-linear z-buffer for the correct depth testing of the GPU.

The most important output is the albedo buffer that represents the colors of the surfaces without taking any lights into account. The emissive buffer represents the emissive colors of the surfaces, i.e., the emitted radiance. The distance buffer represents the linear distances of the pixels from the camera. The normal buffer are the encoded 3D normal vectors of the surfaces. Finally, the position buffer represents the 3D position vectors of the pixels.

The outputs are rendered by iterating through the meshes of the scene while taking the camera view into account. The vertex data are uploaded to the GPU through bgfx. The data are then passed to the vertex and fragment shaders that are explained later in Section 5.3.

Light pass

The `LightPass` is the second phase of our rendering pipeline. This step and all future steps of the method are *screen-space*, i.e., they do not render any meshes of our scene. Instead, we render a single quadrilateral (rectangle) that completely covers the screen. The rectangle has a texture that we render our outputs to. This is a trick that enables us to use the GPU to perform screen-space rendering. The correct rectangle generation code is in `ScreenSpaceQuad.hpp`.

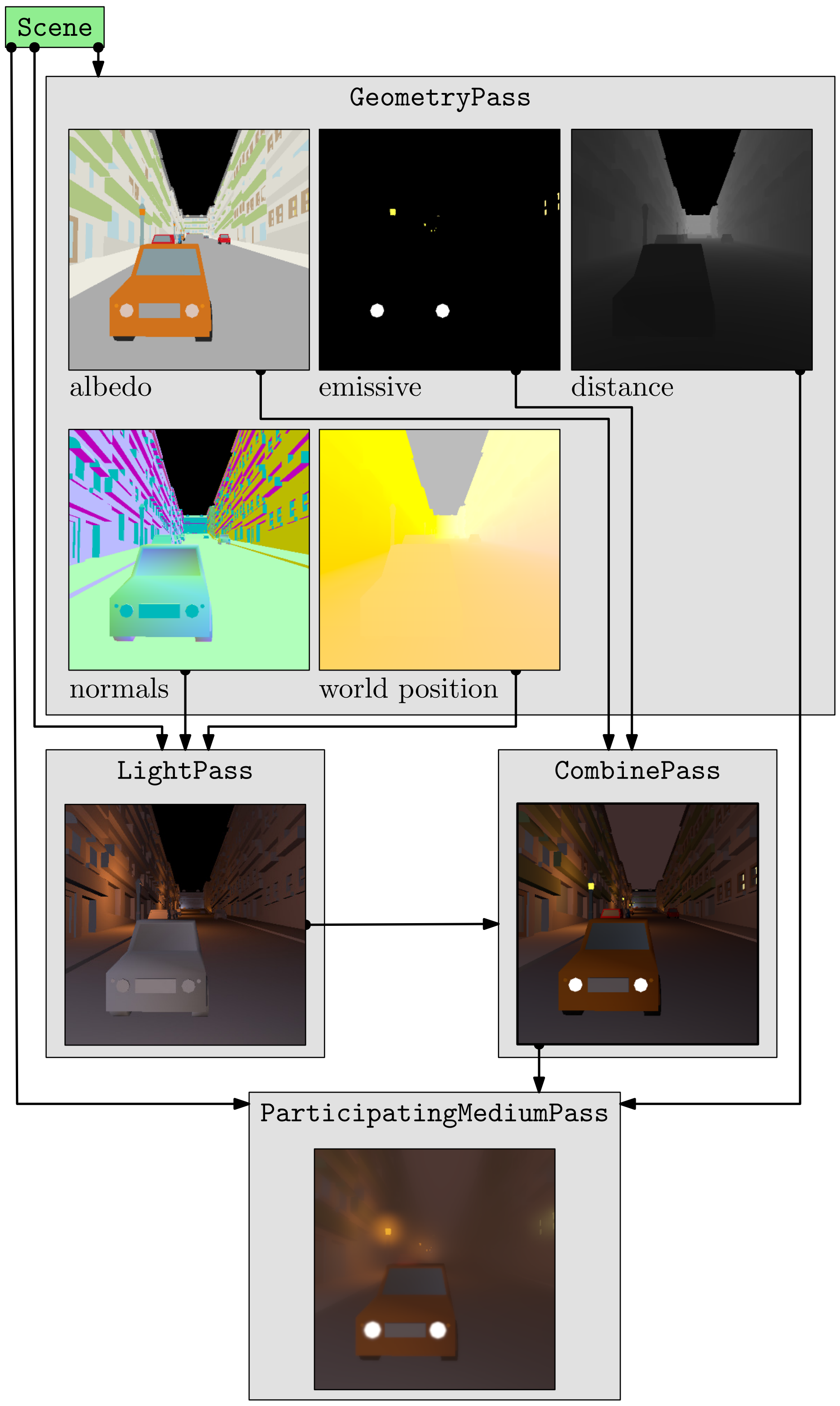


Figure 5.4: Diagram depicting the various stages of the rendering. The arrows illustrate the dependencies on the scene and other images (buffers). The distance, normal, and world position data are encoded into colors for visualization purposes.

The light pass iterates through all light sources in the scene and calculates their contribution one by one for each pixel in the output. In order to correctly calculate the diffuse reflection from geometry surfaces, we need the normals and world positions as inputs of this pass.

It is important to note that the time complexity of this pass is $\mathcal{O}(n^2m)$, where $\mathcal{O}(n^2)$ is the time complexity of the shader itself for the resolution $n \times n$, and m is how many times we have to execute the shader, i.e., how many point light sources we have in the scene. In our case, this step is *not optimized* for rendering a lot of light sources, which would require more sophisticated approach.

Combine pass

The combine pass is only responsible for executing the shader that calculates the final colors of the image rendered in a vacuum according to the rendering equation from the first chapter (Equation 1.6). The light contributions are multiplied with the albedo buffer, that gives us the sum in the equation, then we add the emissive colors that correspond to L_e, and finally we add the constant ambient term L_a.

Participating medium pass

The final rendering pass is the `ParticipatingMediumPass`. The goal of this pass is to take the medium configuration `ParticipatingMedium`, the shader configuration `ParticipatingMediumPassConfig`, and submit these information as uniforms to the shaders. The shaders (Section 5.3) then work in screen-space to preprocess, filter, and compose the images as explained in Chapter 4. The important implementation code runs on the GPU and explained in Section 5.3. The C++ code for the light transport is very similar to all previous passes, except we also need to handle the MIP maps.

For the purpose of correct MIP map handling, we had to modify the original bgfx source code. In our version of bgfx (see the electronic attachment of this thesis), we had to comment the line 5196 in the file `lib/bgfx/src/renderer_gl.cpp`. With this line commented, we prevent OpenGL from building its own MIP chain and overwriting our custom MIP map generation.

It should be noted that the participating medium pass is divided into phases according to Chapter 4. The preprocessing, filtering, and final compositing phases correspond to the methods `renderPrepareStep`, `renderMipmaps`, and `renderResult` respectively. Furthermore, the last two steps are replaced by the `renderGathering` in case we want to render using the gathering approach instead of MIP map filtering. In that case, `renderGathering` is executed only once (see the `render` method) as it is very time consuming even on modern GPUs and can completely freeze the computer (see Attachment 1 – User reference).

5.2.5 Buffer viewer

The `SceneRenderer` is responsible for rendering the outputs to buffers. But the buffers are “hidden” in the GPU memory. In order to display the buffers to the user, we have to render them again on the screen using the `ScreenSpaceQuad`. The `BufferViewer` is responsible for this step.

Our demo application allows the user to display various buffers on the screen—see again Figure 5.4 with images of the buffers. For this purpose, the buffer viewer enables to set the specific input buffer that is rendered on the screen. Furthermore, if the input is a MIP chain, we can set the specific level, and if the input is an HDR image, we can enable tone mapping, i.e., conversion of the HDR input to a standard $[0, 255]$ RGB output. In case the input represents non-image data, such as the distance buffer, we can use a colormap, i.e., render the input as a linear grayscale.

5.2.6 User interface

So far, our application is capable of rendering the scene and displaying the output. Let us now have a look at the user interface, i.e., the classes that handle the camera movement, main menu, and presets saving and loading. The necessary code is located in the `src/ui/` directory.

Camera movement

The `CameraMovement` class handles the mouse and keyboard input from the user. The movement of the position of the camera depends on the pressed keys, the velocity and the time that has passed since the last rendered frame (*delta time*). This corresponds to the standard formula for the traveled distance with regards to speed and time.

When handling the rotation of the camera, we use the position of the mouse on the screen. For that purpose, we have to translate the camera direction vector into vertical and horizontal angles and then back again using the standard trigonometric functions and their inverses. We have to pay close attention to using the correct ranges of the inputs, because otherwise the standard C++ functions could give us invalid results such as `NaN`.

Main menu

The main menu is primarily handled by the `Menu` class that stores the different menu items. The items are assigned to menu categories (`MenuCategory`). The hierarchy of the categories is represented by the stack of open categories. The default view that shows the current FPS and the help text is the implicit open category that should never be removed from the stack.

All menu items are represented by various classes that inherit from the main abstract class `MenuItem`. The method `processInput` is of particular interest, because it defines what happens when the item is selected and the user presses keys on the keyboard.

The `MenuItemSelect` is a selectable menu item. It can be used, for example, for opening new menu categories when the user selects the item. The `MenuItemFloat` represents a floating-point value that can be adjusted by the user by pressing the arrow keys. Finally, the `MenuItemOptions` represents an item where the user can select from a vector of options.

Please note that the menu items are intended to be used with modern C++11 lambda functions. This approach was chosen because it enables us to very quickly define even a complicated menu. For an example of how the menu is constructed, please have a look at the `fillMenuItems` method of the `LightTransportDemo`.

Presets

Because configuring the participating medium and the shader can be very time consuming, the users are allowed to save and load the configurations (presets). The presets are stored as JSON files in the `presets/` directory. The simple `PresetsSerialization` class handles the presets saving and loading using the cereal serialization library and `tinydir.h` for iterating over all files in a directory. Because `tinydir.h` is a C library, the code that we use in this class is a mixture of C++ and standard C approaches.

5.3 Shaders

5.3.1 Overview

In our case, shaders are the programs that are executed on the GPU and create the graphical outputs in Figure 5.4. The `SceneRenderer` class and the rendering passes explained in the previous section are responsible for loading the shaders, passing correct data to them, and setting them for execution. But the actual vertex and pixel calculations are done in the shader programs.

Our application uses two types of shaders, both stored in the `shaders/` directory. The *vertex shaders*, which in our case are very simple, are executed for each vertex of the current geometry and calculate the 3D positions of the vertices on the screen, i.e., the 2D coordinates on the screen and the depth. They are also used to compute other arbitrary outputs that are then passed as inputs to the *fragment shaders*. These shaders are then executed for each pixel (fragment) and have only a single output—the final color of the pixel on the screen.

5.3.2 Notes regarding bgfx

Because we use the bgfx library for our cross-platform multiple-backend rendering, we also have to use a special programming language for writing the shaders. It is very similar to the OpenGL Shading Language (GLSL) but with certain differences[2]. Our shaders with the `.sc` extension are then compiled and optimized for the correct backend, which is OpenGL and GLSL in our case. The shaders can be compiled by running the `shaders/compileAll.bat` script. The compiled shaders are then stored in the `shaders/glsl/` directory.

Please note that this universal shading language we use has certain limitations for compatibility reasons. For example, we are not allowed to define arrays and our uniforms can only be of a few allowed types. This is also the reason why we compact a lot of data into 4D vector uniforms whenever possible.

Furthermore, bgfx has certain predefined uniforms that are accessible from the shaders without the need to specify them in the code. These include for example the `u_modelViewProj` matrix or the `u_viewRect` vector representing

[2] More details, rather outdated, about the differences between OpenGL and bgfx shaders can be found at `https://bkaradzic.github.io/bgfx/tools.html#shader-compiler-shaderc`.

the current screen rectangle. The list of the uniforms can be found in the official API documentation or in the file `lib/bgfx/src/bgfx_shader.sh`. It should also be noted that the *varying* parameters, i.e., the inputs and outputs of the vertex shaders, are defined in the file `shaders/varying.def.sc`.

5.3.3 Vertex shaders

In our case, the vertex shaders are very simple, because the most of our rendering is based on the screen-space approach. For that case, we have a single vertex shader `screenSpaceQuad_vs.sc`, which correctly transforms the screen-space quadrilateral (`ScreenSpaceQuad.hpp`, Section 5.2.4) to the projected screen-space coordinates. The projection to the screen is done by multiplying the local quadrilateral coordinates by the composed model-view-projection matrix. This is a standard approach in computer graphics.

The only exception that does not use the screen-space rendering is the geometry pass. The vertex shader `deferred_geom_vs.sc` primarily needs to correctly calculate the positions of the geometry vertices according to the model-view-projection matrix. Additionally, we also pass other information, such as the normals and world-space positions, as varying outputs to the corresponding fragment shader.

5.3.4 Deferred rendering fragment shaders

We have already explained the outputs of the deferred rendering passes in Section 5.2.4. Let us now have a look at the fragment shaders that are responsible for computing the outputs.

First of all, let us shortly explain the division of our fragment shaders. We can divide them into three categories denoted by prefixes in the file names. The shaders of the deferred rendering passes are denoted by the `deferred_` prefix. The buffer viewer shaders have the `bufferViewer_` prefix. Finally, the participating medium rendering shaders share the `participatingMedium_` prefix.

Geometry shader

The `deferred_geom_fs.sc` shader is very simple as it basically takes the information of the meshes and fills the necessary data into the output buffers. The colors are taken from the uniforms that were passed to the shader. The normal vectors are encoded into 8-bit RGB data using the bgfx `encodeNormalUint` function, which enables the GPU to work with a faster integer buffer. The world positions are merely copied from the vertex shader. The distances are computed simply as the lengths of the vectors between the camera and the pixel positions.

Lighting shader

In order to calculate the lights, we execute the `deferred_light_fs.sc` shader for each light source. For this purpose, the outputs are blended together, i.e., each output of the shader is added to the output of the previous iteration.

The radiance reflected from a surface is computed using the simplified Phong BRDF (Equation 1.3) for each pixel. We only compute the diffuse reflections to maintain the “cartoon” look of the scene, so the original equation is simplified even more as we assume $k_s = 0$. The BRDF is then used in Equation 1.2. The light sources are also attenuated according to Equation 5.1.

Combine shader

The outputs of the geometry shader and lighting shader are combined together in `deferred_combine_fs.sc`. We basically multiply the albedo color by the light contribution, i.e., the constant ambient plus the light buffer. Then we add the emissive colors to the final result.

The only exception are the pixels of the sky. Generally, the sky pixels are not rendered by the geometry shader at all, because the sky has no geometry. Therefore, if the alpha channel of the albedo buffer is zero, the final color is determined by the sky color uniform.

5.3.5 Buffer viewer fragment shaders

The buffer viewer has to display a certain buffer to the screen, i.e., the screen-space quadrilateral. Because the input buffer may be HDR, have multiple MIP levels, or may represent other data, we have three different types of fragment shaders.

In case the input buffer directly represents 8-bit RGBA colors, we simply read the data and pass them to the output directly. If the input is a MIP chain, we use `texture2DLod` to access the correct level. This is performed in `bufferViewer_simpleMapping_fs.sc`.

For HDR inputs, we use a similar approach, but we have to scale the range down to the range $[0, 1]$ in order to correctly display the colors. Note that the $[0, 1]$ range in the shader corresponds to the $[0, 255]$ range in the final output on the screen. In our case, the *tone mapping* is very simple. We compute the final color as $1 - \exp(-\text{hdrColor})$. This is done in `bufferViewer_toneMapping_fs.sc`.

Finally, the input buffer may represent data that are not colors, such as the distance data or encoded normal vectors from the geometry pass. In this case, we use `bufferViewer_colormap_fs.sc` where we rescale the inputs to the $[0, 1]$ according to the input range stored in a corresponding uniform.

5.3.6 Participating medium fragment shaders

Rendering of the light transport in a participating medium is handled by the remaining fragment shaders. The `ParticipatingMediumPass` class is responsible for executing these shaders depending on the current shader configuration.

Preprocessing

The `participatingMedium_prepare_fs.sc` shader is responsible for the preprocessing phase (Section 4.2). This shader is executed for both gathering and MIP filtering approaches. It has three different outputs (buffer attachments) that have to be computed.

The scattering attachment is an HDR 16-bit RGBA texture and represents the L_{sc} image. Because the scattering image only needs the RGB channels, we use the alpha channel to store the rescaled distances $\mathsf{D}' \approx \mathsf{P}$. Similarly, the $\mathsf{L}_{\mathrm{ssc}}$ and the distances $\mathsf{D}'_{\mathrm{ssc}}$ are stored in the separation HDR 16-bit RGBA texture. Finally, we store the standard deviations W in the 16-bit single-channel spread space texture. Note that the L_{at} image is not computed and stored here, because it is only necessary in the compositing step and we do not need to store the data in the GPU memory.

The integrated densities are computed by the `getDensity` function. We first have to get the camera ray of the current pixel by using the inverse matrix of the camera view. Then we use the analytically integrated formulas (Section 4.3) for the different types of density functions. The formulas are calculated by the functions in the separated file `participatingMedium_densityFunctions.sc`.

According to the integrated densities, we can then compute the final values for the buffer attachments. For this purpose, we merely use the equations derived in Section 4.2.

MIP map filtering

The MIP map shader `participatingMedium_mipmap_fs.sc` is executed once for every level $k > 0$, $k < K$ according to Section 4.4. The shader has three output attachments that represent the appropriate levels $\mathsf{L}^{[k]}_{\mathrm{sc}}$, $\mathsf{D}'^{[k]}$ (alpha channel), $\mathsf{L}^{[k]}_{\mathrm{ssc}}$, $\mathsf{D}'^{[k]}_{\mathrm{ssc}}$ (alpha channel), and $\mathsf{W}^{[k]}$. The inputs contain the whole MIP chains, but we only access the previous level $k-1$ in every execution. The necessary variables, including the Gaussian filter size, are stored in uniforms.

The algorithm consists of two filtering functions. The `filterSpreadSpace` function filters $\mathsf{W}^{[k]}$ according to Equation 4.21. The `filterColorAndDepth` is a general implementation that can filter both scattering and distances in both the original and separated images. The filters are implemented as standard convolutions simply by iterating over all pixels in the neighborhood, similarly to the gathering approach.

Final composition

The final output image L' is composed according to Equation 4.30 and the supplementing Equations 4.23 and 4.28 in `participatingMedium_result_fs.sc`. Fetching from specific MIP levels can be done using `texture2DLod`. This shader function performs a linear-bilinear interpolation between MIP levels and the pixels themselves. Because of the GPU hardware, the interpolation happens completely transparently to us and we can assume that it is very fast.

However, as mentioned in Section 3.2.3 and shown in Figure 4.10, bicubic interpolations can give better results. It is especially important when moving around the scene as the cubic spline has a continuous second derivative, so all possible flickering when moving the camera is reduced. For this purpose, our shader also implements the linear-bicubic and cubic-bicubic interpolations.

Unfortunately, as mentioned by Elek et al. [2013], a simple naive bicubic interpolation requires 16 fetches (texture read operations) per each output pixel. The linear-bicubic interpolation would require $2 \cdot 16 = 32$ texture reads. It is therefore not wise to use the naive way in real-time applications.

Fortunately, the bicubic interpolation can be computed in a much faster way. We can use the already mentioned `texture2DLod` function and fetch the pixels at offset positions. The offsets can be computed with regards to the weights of the cubic spline function.

This way, the bicubic interpolation can be performed using only 4 bilinearly interpolated fetches with `texture2DLod`. The linear-bicubic interpolation therefore requires $2 \cdot 4 = 8$ fetches and our cubic-bicubic implementation requires $4 \cdot 4 = 16$ fetches. The implementation in our fragment shader is based on the algorithm presented in the article by Sigg and Hadwiger [2005, Section 20.2]. The interpolation functions are a part of `participatingMedium_common.sc`.

5.3.7 Gathering shader and its artifacts

Instead of executing the MIP map and compositing shaders, we may execute the gathering algorithm in case the user has selected to use it. The algorithm is written in the `participatingMedium_gathering_fs.sc` shader. The final output color is calculated by summing up the attenuated radiance, the gathering filtered scattered radiance, and the emissive term.

Because of the high time complexity of the correct gathering algorithm, we have only implemented the naive incorrectly normalized gathering according to Algorithm 3. Even this “fast” version of the algorithm still takes tens of seconds to render, which requires the user to change the GPU driver settings in Windows to prevent timeouts (see Attachment 1 – User reference).

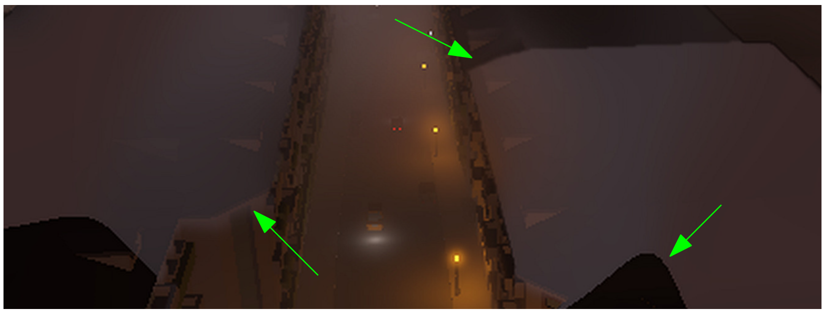


Figure 5.5: Illustration of the artifacts in our implementation of the gathering algorithm. The green arrows point at discontinuities and incorrectly bright areas.

Unfortunately, we have noticed certain artifacts because of the incorrect normalization. Figure 5.5 shows an example of incorrectly calculated contributions. A major error occurs when the contribution to a certain pixel is too low. In that case, the sum of the weights is very low, even though the sum of the colors can be high because of emissive light sources.

When normalizing the color by the incorrectly low total weight, floating point precision errors may occur resulting in bright areas and sharp discontinuities. This is especially critical for HDR inputs with intense emissive light sources, e.g., with our demo scene.

A possible solution is to ignore all pixels whose contribution is too low. Unfortunately, it seems to be very difficult to select a correct threshold not to cause other discontinuities. In our case, we were not able to eliminate the artifacts.

Furthermore, the gathering algorithm requires us to correctly scale the pixel distances. The Gaussian distribution should obviously not depend on the resolution of the screen. However, it is necessary to introduce correction factors to scale the distribution. For example, we need to take the aspect ratio into account in case the screen is not square. In our case, we obtained the best visual results by simply multiplying the distances by the size of the screen-space quadrilateral.

Despite the imperfections, the gathering algorithm is important as it enabled us to compare the blurring of the light sources in Figure 4.5. The artifacts in the implementation do not affect our previously introduced comparisons in any way.

5.4 Performance

The performance of our prototype was thoroughly tested on a fairly modern laptop with 64-bit Windows 10 Pro, Intel Core i7-4700HQ, and NVIDIA GeForce GTX 760M. As our method is primarily targeted to video games and simulations, we can assume that the users will have at least a similar setup, if not better.

The execution times were measured for the HD resolution 1280×720 and depict how long it took to render a single frame. The measured results are 10-second averages to eliminate possible spikes.

Implementation notes – bottlenecks of our prototype

Before commenting on specific numbers, let us note that our demo application is only a prototype. The fragment shaders are not primarily optimized for performance. Our largest fragment shader, after it has been compiled by the bgfx compiler, has 62 kB. It is mainly because of loop and branching transformations made by the compiler. Because of the high configurability of our application, the shaders contain a lot of branching, which is not very optimal for GPUs. Therefore, it can be assumed that the performance could be much better if the shaders were rewritten for a single constant configuration.

Furthermore, because of simplicity, our lighting pass in the prototype application is not optimized for a lot of point light sources. Our demo night scene contains 38 point lights and *all* of them are always rendered for the whole screen resolution. During our experiments on different platforms, we noticed that our lighting pass is especially slow on certain types of GPUs. This step is *not a part of our method* for rendering the participating media and it is a bottleneck of our prototype when rendering the input image L (Section 4.1.2).

To prove that the lighting pass is indeed the bottleneck, we added a startup option to completely disable it. Please see Attachment 1 – User reference.

Results

Rendering the scene without any medium, i.e., generating the input L, took 22.8 ms. It should be noted that by disabling our unoptimized lighting pass, the rendering only took 4.2 ms, which proves the major bottleneck.

The times to process these input images to render the participating effects using our method are detailed in Table 5.1. These times were measured without the pixel separation phase. The additional costs of the pixel separation are detailed in Table 5.2.

As we can see, rendering the participating effects takes 1.8 ms with our fastest configuration and $2.8 + 1.6 = 4.4$ ms with a reasonable configuration with a 4×4

filter, linear-bicubic interpolation, and pixel separation.

The time differences between integrating the homogeneous, exponential, and spherical media were immeasurable. The rendering times remained the same with ±0.2 ms differences for the density functions. That is because the integrations are performed analytically by solving rather simple formulas for each pixel.

Filter size	**Interpolation**		
	Linear-bilinear	Linear-bicubic	Cubic-bicubic
2×2	1.8 ms	1.9 ms	2.0 ms
4×4	2.5 ms	2.8 ms	3.4 ms
6×6	9.0 ms	9.5 ms	10.3 ms

Table 5.1: Times to process a single 1280×720 frame with an exponential medium using our method with various filter sizes and interpolation techniques. The times were measured without the pixel separation step.

Filter size	**Interpolation**		
	Linear-bilinear	Linear-bicubic	Cubic-bicubic
2×2	0.4 ms	0.8 ms	1.6 ms
4×4	1.0 ms	1.6 ms	3.5 ms
6×6	2.6 ms	3.2 ms	4.1 ms

Table 5.2: Additional processing time for the pixel separation step with various filter sizes and interpolation techniques.

Discussion

The results we have achieved with the 4×4 filter and linear-bicubic interpolation are fairly similar to the results of Elek et al. [2013]. Similarly to their conclusion, this configuration seems to be the best compromise between price and quality.

Using the 2×2 filter cannot be recommended, because it is essentially a box filter and does not behave like the Gaussian distribution. On the other hand, the cost of the 6×6 filter is excessive without any significant visual improvements.

Furthermore, using the cubic-bicubic interpolation does not seem to be necessary and almost same results can be achieved with the linear-bicubic interpolation (see again Figure 4.10). On the other hand, the linear-bilinear interpolation suffers from very noticeable artifacts when moving or rotating the camera.

The results prove that our method *can be used for real-time rendering.* Even with our unoptimized prototype, the total rendering time (4×4, linear-bicubic) is 27.2 ms, lower than our formal 40 ms requirement. Results we could obtain when using an optimized commercial AAA video game engine would probably be even much more optimistic.

We would like to note that the naive incorrectly normalized gathering approach took 79 seconds for the same scene, which is absolutely unsuitable for real-time rendering. Path-tracing based methods would probably take several hours or days as shown by Elek et al. [2013] and Shinya et al. [2016].

Conclusion

Summary

In this thesis, we aimed to propose and fully implement a real-time method for solving a difficult problem of computer graphics. Our goal was to render light transport in virtual scenes while considering multiple scattering and other light interactions in participating media. Furthermore, we assumed interactions not only in simple homogeneous media, but also in quasi-heterogeneous media whose density functions can be analytically integrated.

Within this work, we have explained the necessary physical and mathematical background and examined various existing solutions. After verifying that none of the approaches is capable of solving our problem while fulfilling all our requirements, we presented our own solution. For this purpose, we have selected one of the existing methods as our baseline for improvements.

We have managed to propose how the equations in the existing approach can be modified to support non-homogeneous media. Later, we derived and showed how certain density functions can be integrated in real-time and used in our method. The various density functions can simulate not only global, but also local effects. Furthermore, we have enhanced the original filtering by introducing new steps for better visual results.

The whole method has been successfully implemented in our prototype demo application. We have also built a demo scene to demonstrate how our solution behaves in night scenes with intensely emissive light sources. Our implementation proves that our method is capable of running in real time as the processing of a single frame takes only a few milliseconds. It also proves that our method can indeed solve our problem to the necessary degree.

Even though the method is not perfect—and we have thoroughly explained the limitations and inherent visual artifacts in Section 4.5—it satisfyingly simulates the participating media while fulfilling all our requirements. Despite the limitations, which are common for the competing methods as well, our solution provides an alternative that can successfully compete with the existing approaches.

Fulfillment of the goals

Let us now verify that the original goals presented in Introduction have been successfully fulfilled.

1. **Background** In order to present our method, we first had to explain the necessary physical and mathematical background. Chapter 1 was devoted to clarifying how light behaves in a vacuum and how the behavior changes when introducing participating media. We presented the important volume rendering and radiative transfer equations. Finally, we had a look at the mathematical operation called convolution, we explained what a point spread function is, and we showed how to handle spatially varying filtering.

2. **Related works** In Chapters 2 and 3, we examined a substantial volume of related works. We began with explaining how others approximated our problem by empirical or single-scattering approaches. Because these approximations were rather simple, the following chapter was devoted to much more precise multiple-scattering solutions. Within the chapter, we also explained efficient Gaussian MIP map filtering and various physical approaches to the multiple-scattering problem.

3. **Proposed method** Our biggest contribution, the proposed method, was explained in Chapter 4. At the beginning, we formalized our requirements in the context of physics and the related works. As a result, we decided to base our own solution on the real-time multiple-scattering method of Elek et al. [2013]. Then we thoroughly explained our modified approach for multiple-scattering in analytically integrable participating media. We also improved the filtering step for better support of HDR images with intensely emissive materials. Finally, the results and limitations of our method were precisely analyzed, also in the context of the related works.

4. **Implementation** To fully demonstrate our method, we have built a demo scene and implemented a demo application. Not only it allows free navigation around the scene and modifying various parameters of the participating medium, the user of our application can also display different steps of our method and change the configuration of the process fully interactively. Chapter 5 is dedicated to the implementation details. We described the technologies, our C++ source codes, and the vertex and fragment shaders. The last section of the chapter was devoted to the performance of our method, where we proved that our implementation is capable of real-time rendering. The demo application is available as an electronic attachment.

Future work

During my work on this thesis, I have realized that the light transport problem is very broad and difficult, especially when trying to solve it in real-time. In order to solve the problem, we had to make various assumptions and sophisticated approximations. As a result, we certainly did not solve all aspects of light transport and there remains many different paths for possible future exploration. Let us mention at least a few of them.

Path-tracing reference It would be very interesting to compare our method to a path-tracing reference similarly to Elek et al. [2013]. The path-tracing algorithm can take several hours or days to run, but could tell us more about where exactly our solution gives the less correct results.

Animated media Our analytically integrated densities are constant in time, but the method itself is not limited to static participating media. It could be especially interesting for video games to render visually attractive animations, such as a sand storm moving through a scene.

Occlusions and shadows Enhancing our method with volumetric effects, such as the crepuscular rays mentioned in Section 2.3, would certainly make our results look even more realistic. It would be interesting to explore the possibilities in this area, possibly calculating the occlusions in screen-space (Section 2.3) or in 3D cuboids (Section 2.4).

Angular spreading Our method is based on approximating multiple scattering by a Gaussian spatial spreading distribution on a planar sensor. Real cameras, however, detect the radiance incoming from different angles, then passing through lens, and finally reaching the sensor. Future exploration in this area is especially important because the original equation by Premože et al. [2004] assumes spatially invariant parameters, which is not true for non-homogeneous media.

Ambient and illumination before reflecting The inputs of our method are rendered in a vacuum, so the participating media are only taken into account on the paths after being reflected towards the camera. Not only is this a big approximation, it also disallows us to correctly render the scattering from the ambient light, because it should scatter even without reaching any surface. We tried experimenting with calculating the ambient term from the MIP maps, but without satisfying results.

Mixed media Since Chapter 1, we have only assumed a single participating medium with identical particles. It would be interesting to try to handle rendering of scenes with mixed media, such as underwater sand whirls. This could be especially useful in combination with the animated media.

Semi-transparency Finally, in the whole thesis, we have assumed that light reflects according to the bi-directional reflectance distribution function without considering semi-transparent materials. It is not exactly clear how our method could be used to render light transport in scenes with materials such as glass.

Bibliography

T. Akenine-Möller, E. Haines, and N. Hoffmann. *Real-time Rendering*. 3rd edition. 2008.

G. Becker. Photo, 2015. URL `https://unsplash.com/search/sun-rays?photo=-0_ww2ACIw8`. Accessed: 2017-05-04, Dedicated to the Public Domain.

M. N. Berberan-Santos, E. N. Bodunov, and L. Pogliani. On the barometric formula. *American Journal of Physics*, 65(5):404–412, 1997.

C. L. Braun and S. N. Smirnov. Why is water blue? *Journal of Chemical Education*, 70(8):612–614, 1993.

S. Chandrasekhar. *Radiative Transfer*. 1960.

O. Elek. *Efficient Methods for Physically-Based Rendering of Participating Media*. PhD thesis, Max Planck Institute for Computer Science, 2016.

O. Elek, T. Ritschel, and H.-S. Seidel. Real-time screen-space scattering in homogeneous environments. *IEEE Computer Graphics & Applications*, 2013.

O. Fialka and M. Čadík. FFT and convolution performance in image filtering on GPU. In *Tenth International Conference on Information Visualisation (IV'06)*, pages 609–614, 2006.

W. Jarosz. *Efficient Monte Carlo Methods for Light Transport in Scattering Media*. San Diego, 2008.

J. T. Kajiya. The rendering equation. *ACM SIGGRAPH Computer Graphics*, 20 (4):143–150, 1986.

F. Klawonn. *Introduction to Computer Graphics Using Java 2D and 3D*. Springer, 2008. ISBN 978-1-84628-847-0.

S. Lee, G. J. Kim, and S. Choi. Real-time depth-of-field rendering using anisotropically filtered mipmap interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 15(3):453–464, 2009.

J. Li, Y. Koudota, M. Barkowsky, H. Primon, and P. Le Callet. Comparing upscaling algorithms from HD to ultra HD by evaluating preference of experience. In *2014 Sixth International Workshop on Quality of Multimedia Experience (QoMEX)*, pages 208–213, 2014.

K. Mitchell. Volumetric light scattering as a post-process. *GPU Gems 3*, pages 275–284, 2007.

R. Montes and C. Ureña. An overview of BRDF models. 2012.

T. Persson. Practical particle lighting, 2012. URL `http://www.roxlu.com/downloads/scholar/008.rendering.practical_particle_lighting.pdf`. Accessed: 2017-04-23.

S. Premože, M. Ashikhmin, R. Ramamoorthi, and S. Nayar. Practical rendering of multiple scattering effects in participating media. *Proceedings of the Fifteenth Eurographics conference on Rendering Techniques*, pages 363–374, 2004.

Í. Quílez. Better fog, 2010. URL `http://www.iquilezles.org/www/articles/fog/fog.htm`. Accessed: 2017-04-22.

Í. Quílez. Sphere density, 2015. URL `http://www.iquilezles.org/www/articles/spheredensity/spheredensity.htm`. Accessed: 2017-04-22.

M. Segal and K. Akeley. *The OpenGL^{TM} Graphics System: A Specification.* Silicon Graphics, Inc., 1994.

M. Shinya, Y. Dobashi, M. Shiraishi, M. Kawashima, and T. Nishita. Multiple scattering approximation in heterogeneous media by narrow beam distributions. *Computer Graphics Forum*, 35(7):373–382, 2016.

C. Sigg and M. Hadwiger. Fast third-order texture filtering. *GPU Gems 2*, pages 313–329, 2005.

B. Wronski. Volumetric fog: Unified compute shader-based solution to atmospheric scattering, 2014. URL `http://advances.realtimerendering.com/s2014/wronski/bwronski_volumetric_fog_siggraph2014.pdf`. Accessed: 2017-04-22.

List of Figures

Attachment 1 – User reference

System requirements

The demo application has been tested on several computers with different setups. With regards to our observation, the recommended system configuration is:

- Operating system: Windows 7, 8.1, or 10,
- CPU: x86 or x64, frequency 2.0 GHz or higher,
- Physical memory: 1 GB or more,
- GPU[1]: NVIDIA GeForce GTX 285 or better, or AMD equivalent (tested on Radeon HD 7670M); newest drivers and support for OpenGL 3.1 or newer,
- Hard drive space: approx. 100 MB free space if not executing from CD,
- Resolution: 1280×720 or more.

It is also required to have the necessary C++ components available on the PC. For this purpose, Visual C++ Redistributable for Visual Studio 2015 should be installed. The package can be either downloaded[2] or installed directly from the electronic attachment (file `dependencies/vc_redist.x86.exe`).

Startup

The demo application does not have to be installed and can be started directly by running the file `LightTransportDemo.bat`. This batch file executes the application with the correct working directory.

If your computer cannot run the demo fast enough, try lowering the window size. Furthermore, you can run a low-quality version with disabled point light sources by executing `LightTransportDemo-NoLights.bat`. As explained in Section 5.4, the rendering of point lights can be very slow on certain types of GPUs, because our lighting shader renders all 38 point lights in each frame.

In case the demo is executed from a read-only directory, e.g., the CD attachment, it is not possible to save new presets. Therefore, it is recommended to copy the whole electronic attachment into a directory with read-write permissions.

General usage

Upon executing the demo, two windows should appear. The window with a text output is a console and reports the state of the application. In case any error occurs, information should be available in the console. The graphical window should display the demo scene with a default participating medium. It is possible to move and look around the 3D scene without collisions with the geometry:

- `W/A/S/D` keys control the camera movement,
- holding `Ctrl` or `Alt` slows down or speeds up the movement, respectively,
- holding the right mouse button while moving the cursor rotates the camera.

[1] On laptops with multiple GPUs, make sure that the executable `demo/builds/vs/Release/LightTransportDemo.exe` runs on the dedicated GPU according to your laptop's manual.

[2] `https://www.microsoft.com/en-us/download/details.aspx?id=48145`

The main window should display additional text information in the top left corner, including the frame time and a list of hotkeys. The text can be hidden and shown again by pressing `Tab`. The `F1` key displays detailed runtime information provided by the bgfx library. Pressing `F7` controls the vertical synchronization that can limit the frame time.

Main menu

Furthermore, the main menu can be open to control various parameters of the demo application. The `Enter`, `Esc`, and arrow keys are used for navigation in the menu. Menu items can be selected using the `Enter` key.

Values (in brackets, e.g., `[0.000625]`) can be changed using the left and right arrows. Holding `Ctrl` or `Alt` can further control how much the values are changed. Pressing `Space` resets the value to minimum. Certain values, e.g., vectors and colors, can be represented as multiple items.

In certain cases, where it makes sense, the demo application does not disallow entering incorrectly low or high values. This way, the method can be tested even with extreme parameters. It can result in severe artifacts according to Section 4.5.

View buffer selection

It is possible to select which buffer should be displayed on the screen. The different buffers correspond to Figure 5.4 and to our method (Chapter 4). For MIP chains (denoted by `[MIP]`), the level to be displayed can be selected. When a decimal value is entered, the level is linearly interpolated.

Shader configuration

Different parameters such as the filter sizes and interpolation techniques can be changed. Certain values correspond to parameters in the equations in this thesis:

- “Scaling constant” corresponds to c in Equations 3.7, 4.22, and others,
- “Masking threshold width” corresponds to ε in Equation 4.20,

and the following values correspond to parameters in Equations 4.25 and 4.28:

- “Separation – minimum luminance” corresponds to T_y,
- “Separation – luminance threshold width” corresponds to ε_y,
- “Separation – maximum optical depth” corresponds to $T_{d'}$,
- “Separation – optical depth threshold width” corresponds to $\varepsilon_{d'}$,
- “Separation – depth MIP level” corresponds to normalized k_{ssc}/K.

Notes regarding optimal values can be found in Section 4.5.4.

Shader configuration / Gathering algorithm

This menu category enables to execute the naive gathering algorithm. Please note the limitations of our implementation in Section 5.3.7.

Once the “Compute & show gathering result” item is selected, the GPU will start computing the gathering algorithm with the selected neighborhood size. This operation may result in **Windows aborting the application** because the

GPU driver will probably timeout. The gathering algorithm is very slow and can take up to several minutes to finish. Once the result is computed, it is displayed on the screen and stays until the hide option is selected.

In order to prevent Windows from aborting the application, it is necessary to edit the Windows registry. The simple way to do it is to execute the file `dependencies/tdr_timeout.reg` and confirm the changes. It will prolong the delay to 5 minutes. Please note that it is necessary to restart the computer after making the change. Advanced users of the Windows operating system may change the value themselves according to the Windows documentation[3].

Medium parameters

It is possible to change various parameters of the participating medium in the scene. The "Ambient & sky & emissive" menu contains three colors: the ambient term L_a (Equation 1.6), the color of the sky, and the emissive term L_e (Equation 4.29).

We can also change and configure the density function and its factor $\varrho' \geq 0$ (Section 4.3). The more specific parameters of the density functions are changed separately. The "Set exponential parameters" menu contains the falloff parameter b, the direction $\mathbf{n}$, and the offset $\mathbf{o}$ (Section 4.3.2). The "Set sphere parameters" menu contains the sphere radius R and the center $\mathbf{o}$ (Section 4.3.3).

Furthermore, the absorption and scattering cross-sections C_a and C_s can be set along with the scattering asymmetry factor g (Section 1.2).

Shader & medium presets

Because setting the parameters can be time consuming and difficult, the configurations can be saved and loaded later. Several predefined configurations are available so even inexperienced users can try how the demo application behaves with different parameters.

The presets are stored in the `demo/presets/` directory and can be loaded from the menu. By pressing the "Save current preset", the shader and medium parameters are saved into a new file named `NewPreset_X.json`, where `X` is the lowest available number. The name is always printed to the console window. It is recommended to rename the file in the Windows file explorer.

Camera selection

The last menu category enables to change the field of view (FOV) of the camera and teleport to predefined numbered locations in the scene. The FOV parameter is useful when taking screenshots, because more objects can fit on the screen with high FOV. Unfortunately, the proportions of the objects may deform.

[3] `https://docs.microsoft.com/en-us/windows-hardware/drivers/display/tdr-registry-keys`

Attachment 2 – CD contents

The contents of the accompanying CD are organized as follows:

- `demo/` — the demo application with full source codes,

- `demo/builds/vs/` — Visual Studio solution and compiled binaries,
- `demo/lib/` — the C++ libraries required to compile the project,
- `demo/media/` — the demo scene,
- `demo/presets/` — predefined configurations of the demo application,
- `demo/shaders/` — vertex and fragment shaders,
- `demo/src/` — C++ source codes,

- `dependencies/vc_redist.x86.exe` — Visual C++ redistributable installation,
- `dependencies/tdr_timeout.reg` — Windows registry update file to prevent GPU driver timeouts,

- `thesis/thesis.pdf` — this thesis,

- `LightTransportDemo.bat` — main startup file,
- `LightTransportDemo-NoLights.bat` — low quality startup file with disabled point lights shader,
- `README.txt` — more information including the licenses of the libraries,
- `README-COMPILE.txt` — information about compiling the application.