DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Volume Rendering of Meteorological Simulation Data

Florian Märkl

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics: Games Engineering

Volume Rendering of Meteorological Simulation Data

Volumen-Rendering von Meteorologischen Simulationsdaten

Author:	Florian Märkl
Supervisor:	Prof. Dr. Rüdiger Westermann
Advisor:	Dr. Marc Rautenhaus
Submission Date:	16.04.2018

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Florian Märkl	Place, Date

Acknowledgments

I want to thank Dr. Marc Rautenhaus for providing the topic as well as giving constant constructive feedback and advice. In addition, I would like to thank him and all the other Met.3D contributors for their work and for building this software on an open and standardized foundation in the form of OpenGL as well as publishing it under a free software license, thus ensuring that everybody with an interest in using or contributing to it can do so without any restrictions of proprietary software. Furthermore, I want to thank Theresa Alina Diefenbach for providing me with the background knowledge for the conversion from liquid and ice water content to extinction coefficients and aiding me with my work by providing valuable help.

Abstract

This thesis presents the integration of two different volume lighting methods to be used for visualization of meteorological simulation data, in particular cloud data, in Met.3D, an open-source visualization application for meteorological uses. The first method is primarily a visuals-based approach, which simulates single scattering and imitates certain visual phenomena appearing in real-world clouds. The second, while generally requiring more computational effort, is a specific implementation of the photon mapping algorithm and directly simulates multiple scattering by tracing photons through the volume. It takes advantage of the Henyey-Greenstein function in order to be able to only use 3D textures as the data structure of the photon map. We use our implementations of these methods to produce a number of images and analyze them with respect to visual outcome, physical accuracy and performance. In addition, we compare images created with the photon mapping method to results that were generated using MFASIS, a method to simulate radiative transfer of clouds.

Contents

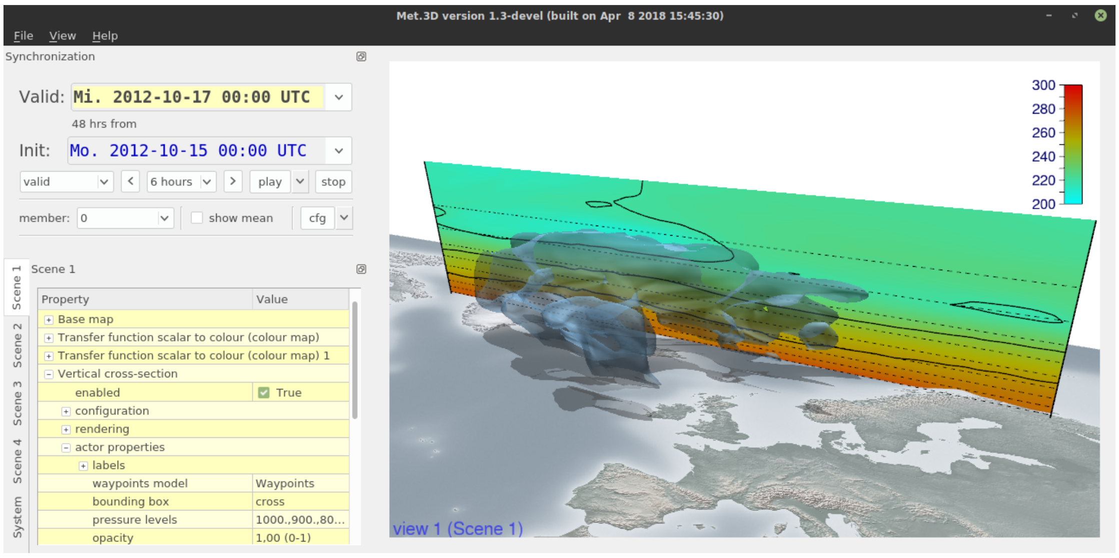
1. Introduction

Met.3D [Rau+15; M] is an open-source application for interactive visualization of meteorological numerical data sets, such as numerical ensemble weather predictions, developed at the Computer Graphics & Visualization Group in the Department of Informatics of the Technical University of Munich.

Among other types of visualization, it can display given data as a three-dimensional volume. Intuitively, such a visualization would seem to be fitting especially for rendering clouds, as it has been done in fig. 1.1b. User experience has shown, however, that the three-dimensional structure of given data is hard to grasp from images produced by the current volume rendering implementation in Met.3D. As an alternative to volume rendering, one type of visualization that is commonly used is rendering in the form of isosurfaces, but these have a limitation in the way that they only display discrete values.

Since humans, especially when viewing monoscopic imagery, take a majority of the visual cues they use to perceive three-dimensional structures and relationships of objects between each other from effects produced by lighting, such as specular highlights or shadows, adding some sort of lighting to the current volume rendering implementation would be a logical approach to solve the problem mentioned above.

Out of this motivation, in this thesis we will present the background as well as implementations of two different types of volume lighting in Met.3D and analyze them with respect to visual outcome, physical accuracy and performance.

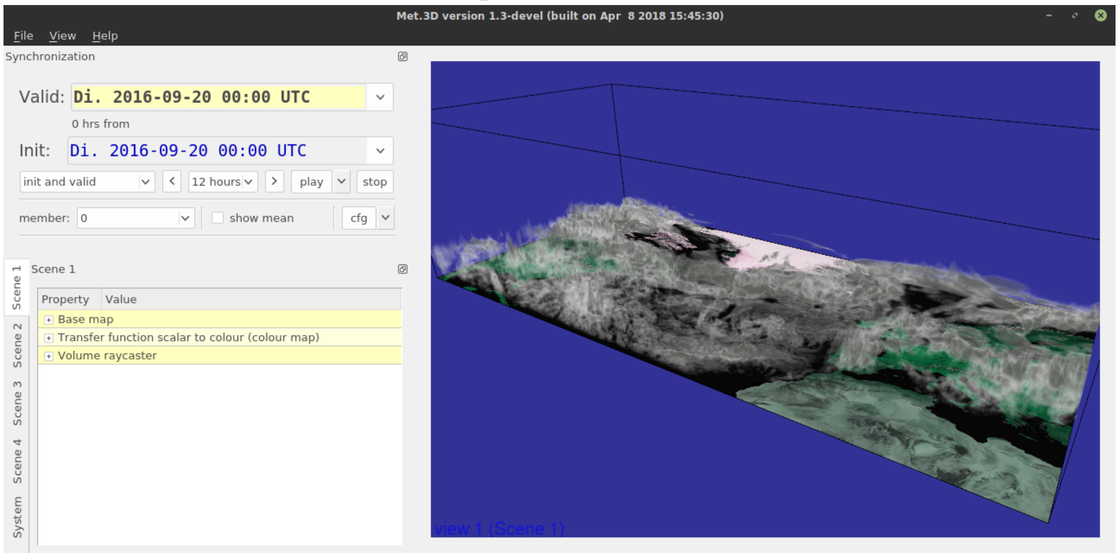


Valid: Mi. 2012-10-17 00:00 UTC
48 hrs from
Init: Mo. 2012-10-15 00:00 UTC
valid | 6 hours | play | stop
member: 0 | show mean | cfg

Scene 1

Property	Value
⊞ Base map	
⊞ Transfer function scalar to colour (colour map)	
⊞ Transfer function scalar to colour (colour map) 1	
⊟ Vertical cross-section	
enabled	✓ True
⊞ configuration	
⊞ rendering	
⊟ actor properties	
⊞ labels	
waypoints model	Waypoints
bounding box	cross
pressure levels	1000.,900.,80...
opacity	1,00 (0-1)

(a) Graphical user interface



Valid: Di. 2016-09-20 00:00 UTC
0 hrs from
Init: Di. 2016-09-20 00:00 UTC
init and valid | 12 hours | play | stop
member: 0 | show mean | cfg

Scene 1

Property	Value
⊞ Base map	
⊞ Transfer function scalar to colour (colour map)	
⊞ Volume raycaster	

(b) Cloud liquid water content visualized as a volume

Figure 1.1.: Screenshots of Met.3D

2. Background

2.1. Used Symbols

The following symbols are used throughout this thesis:

L	radiance $[\mathrm{W\,m^{-2}\,sr^{-1}}]$
Φ	flux [W]
κ	extinction, absorption or scattering coefficient $[\mathrm{m^{-1}}]$
τ	optical thickness [1]
Tr	transmittance [1]
θ	angle [rad]

2.2. Participating Media

A participating medium is a medium which affects light that travels through it in some way. Specific cases include glass, fog, steam, but also clouds. The following section gives a general overview on light transfer in such a medium.

2.2.1. Radiative Transfer Equation

The **radiative transfer equation (RTE)** models the change of radiance when light travels along a direction $\vec{\omega}$ through a medium at a point x. For participating media, three different phenomena have to be taken into account:

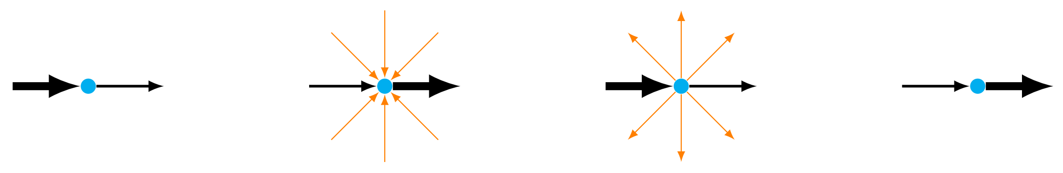

(a) Absorption (b) In-Scattering (c) Out-Scattering (d) Emission

Figure 2.1.: The different phenomena modeled by the radiative transfer equation in participating media

Absorption

Absorption, as illustrated in fig. 2.1a, describes the process of light energy being converted to heat energy. The fraction of the incoming energy being absorbed is given by $\kappa_a(x)$, the **absorption coefficient** at point x. Thus, this is the radiative transfer equation for pure absorption:

$$(\vec{\omega} \cdot \nabla)L(x \to \vec{\omega}) = -\kappa_a(x)L(x \to \vec{\omega}) \tag{2.1}$$

Scattering

When a photon travels through participating media, it not only moves along a straight line, but can also be deflected to a different direction a number of times until exiting the medium. This phenomenon is called **scattering**. When looking at a specific point x and direction $\vec{\omega}$ in the medium, the effect of scattering can be divided into two separate terms.

Out-scattering describes the process of radiance from the incoming light along $\vec{\omega}$ being scattered to a different direction and thus not participating in the outgoing light. The amount of light being scattered at point x is dependent on the scattering coefficient $\kappa_s(x)$.

$$(\vec{\omega} \cdot \nabla)L(x \to \vec{\omega}) = -\kappa_s(x)L(x \to \vec{\omega}) \tag{2.2}$$

In-scattering analogously describes light scattered from all incoming directions to $\vec{\omega}$. The radiance from incoming direction $\vec{\omega'}$ is integrated over all solid angles and multiplied by the phase function $p(\vec{\omega'} \to \vec{\omega})$, which gives the distribution of light being scattered along a certain angle:

$$\begin{aligned}(\vec{\omega} \cdot \nabla)L(x \to \vec{\omega}) &= \kappa_s(x)L_i(x \to \vec{\omega}) \\ &= \kappa_s(x)\int_{\Omega_{4\pi}} p(\vec{\omega'} \to \vec{\omega})L(x \leftarrow \vec{\omega'})d\omega'\end{aligned} \tag{2.3}$$

Refer to section 2.3 for more information about phase functions.

Emission

Emission describes light being emitted by the medium itself. It is given by the source function $Q(x \to \vec{\omega})$ which gives the emitted radiance at point x into direction $\vec{\omega}$:

$$(\vec{\omega} \cdot \nabla)L(x \to \vec{\omega}) = Q(x \to \vec{\omega}) \tag{2.4}$$

Full Radiative Transfer Equation

Taking into account all four phenomena, the complete radiative transfer equation for participating media can be formulated:

$$\begin{aligned}(\vec{\omega} \cdot \nabla)L(x \to \vec{\omega}) = &- \kappa_a(x)L(x \to \vec{\omega}) \\ &- \kappa_s(x)L(x \to \vec{\omega}) \\ &+ \kappa_s(x) \int_{\Omega_{4\pi}} p(\vec{\omega}' \to \vec{\omega})L(x \leftarrow \vec{\omega}')d\vec{\omega}' \\ &+ Q(x \to \vec{\omega})\end{aligned} \tag{2.5}$$

Extinction

The part of the RTE reducing the radiance by a factor is called **extinction**. In the case of participating media, the two phenomena contributing to it are absorption and out-scattering, thus the so-called **extinction coefficient** κ_t is the sum of their coefficients:

$$\kappa_t = \kappa_a + \kappa_s \tag{2.6}$$

The RTE for pure extinction is then formulated as follows:

$$(\vec{\omega} \cdot \nabla)L(x \to \vec{\omega}) = -\kappa_t(x)L(x \to \vec{\omega}) \tag{2.7}$$

2.2.2. Volume Rendering Equation

Given a medium with pure extinction, it is possible to calculate the attenuation of radiance from one point x_0 to another point x.

In a first step, the extinction coefficient needs to be integrated over the distance between the two points:

$$\tau(x_0 \leftrightarrow x) = \int_{x_0}^{x} \kappa_t(y)dy \tag{2.8}$$

This term is referred to as **optical thickness** or **optical depth**. In a homogeneous medium, the calculation can be reduced to a single multiplication with the distance:

$$\tau(x_0 \leftrightarrow x) = \kappa_t|x - x_0| \tag{2.9}$$

From optical thickness, a factor for the attenuation of radiance, called **transmittance**, can be calculated using **Beer's Law**:

$$T_r(x_0 \leftrightarrow x) = e^{-\tau(x_0 \leftrightarrow x)} \tag{2.10}$$

And thus, the incoming radiance at point x with $\vec{\omega}$ being the normalized direction from x_0 to x is defined as:

$$L(x \leftarrow \vec{\omega}) = T_r(x_0 \leftrightarrow x)L(x \rightarrow \vec{\omega}) \tag{2.11}$$

Also adding in-scattering and emission to the term yields the full **volume rendering equation**:

$$\begin{aligned} L(x \leftarrow \vec{\omega}) = & \; T_r(x_0 \leftrightarrow x)L(x \rightarrow \vec{\omega}) \\ & + \int_{x_0}^{x} T_r(y \leftrightarrow x)Q(y \rightarrow \vec{\omega})dy \\ & + \int_{x_0}^{x} T_r(y \leftrightarrow x)\kappa_s(y)L_i(y \rightarrow \vec{\omega})dy \end{aligned} \tag{2.12}$$

2.3. Phase Functions

As mentioned, when scattered, a light ray changes its direction. The distribution of scattered light is described by the **phase function** $p(\vec{\omega}_0 \rightarrow \vec{\omega})$, which gives the amount being scattered from direction $\vec{\omega}_0$ to $\vec{\omega}$, also commonly used in the form $p(\theta)$, given the angle θ between the two directions.

In order to describe a valid distribution, i.e. one that outputs the exact amount of light that is put into it, the phase function integrated over the whole sphere must evaluate to 1:

$$\int_{\Omega_{4\pi}} p(\theta)d\theta = 1 \tag{2.13}$$

2.3.1. Isotropic Phase Function

If light is scattered to all directions equally, the scattering is referred to as **isotropic** scattering, as opposed to **anisotropic** scattering. In such a case, the phase function must be a constant value. In order to satisfy eq. (2.13), this value is $\frac{1}{4\pi}$:

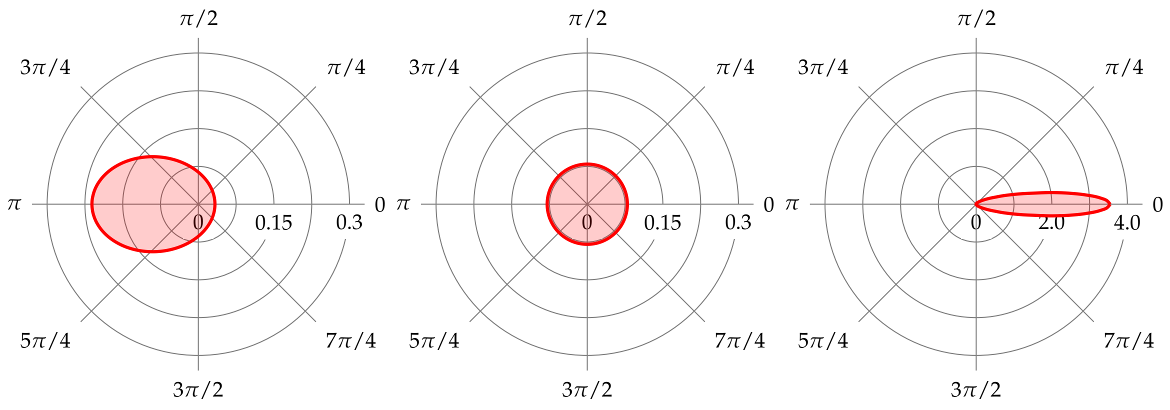
$$p(\theta) = \frac{1}{4\pi} \tag{2.14}$$

$$\int_{\Omega_{4\pi}} p(\theta)d\theta = \int_{\Omega_{4\pi}} \frac{1}{4\pi}d\theta = 1 \tag{2.15}$$

2.3.2. Henyey-Greenstein Function

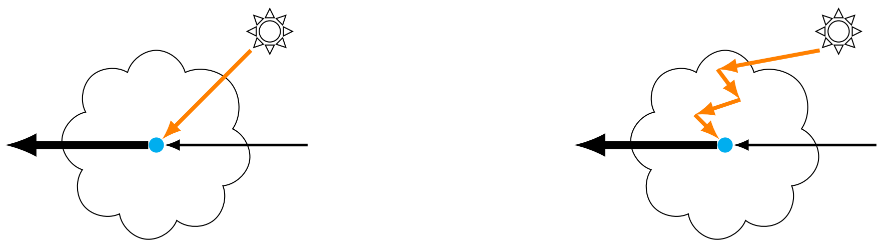
The Henyey-Greenstein function is a phase function that, given a parameter $g \in [-1, 1]$, can represent back scattering, forward scattering or isotropic scattering. It is defined as

$$p_{HG}(\theta, g) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g\cos\theta)^{1.5}} \tag{2.16}$$



(a) $g = -0.3$ (b) $g = 0$ (c) $g = 0.8$

Figure 2.2.: Henyey-Greenstein phase function for different values of g, ranging from back scattering (fig. 2.2a) through isotropic (fig. 2.2b) to almost pure forward scattering (fig. 2.2c). Note the different scaling in fig. 2.2c.



(a) Single scattering (b) Multiple scattering

Figure 2.3.: Illustrations of single and multiple scattering

The **asymmetry factor** g determines the characteristic of the phase function. Negative values result in backwards-oriented scattering, positive in forward-oriented scattering and with a value of 0, the function collapses to a constant value of $\frac{1}{4\pi}$, representing isotropic scattering as described above. The plotted function for different values of g can be seen in fig. 2.2.

2.4. Single and Multiple Scattering

When simulating scattering, a distinction is often made between **single** and **multiple scattering**. Single scattering, as illustrated in fig. 2.3a, describes a ray of light coming directly from the light source being scattered exactly once at a single point, then keeping the scattered direction. Multiple scattering, which is illustrated in fig. 2.3b, allows an arbitrary amount of scattering events to occur until the ray obtains its final direction.

3. Method

We present two different methods for volumetric lighting of cloud data. The first is henceforth referred to as the "simple" method while the second is called the "photon mapping" method.

3.1. Simple

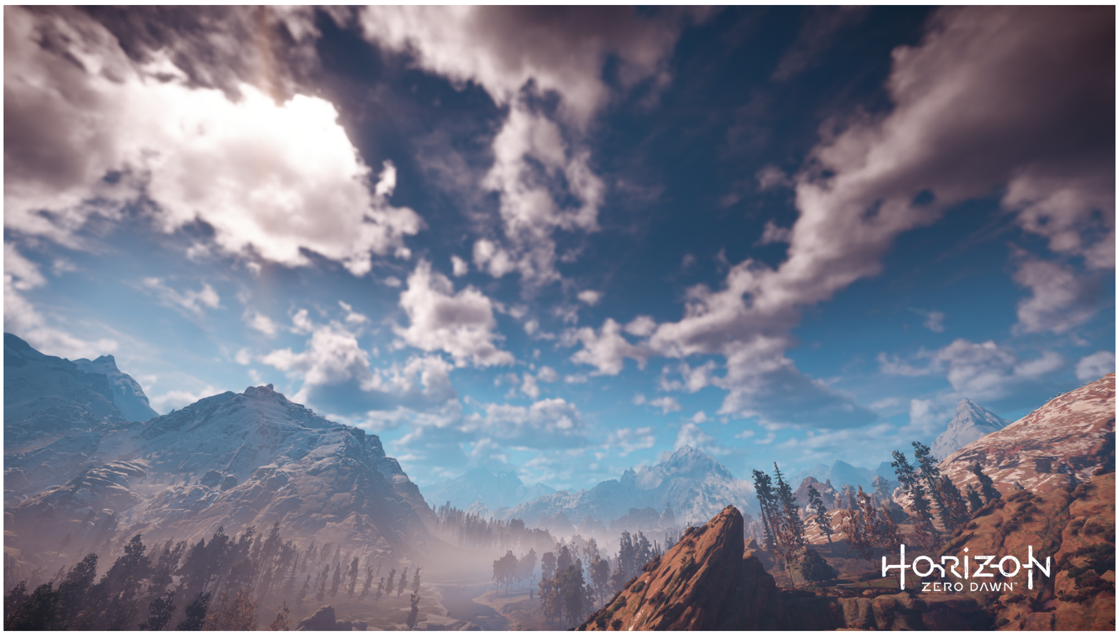


Figure 3.1.: A cloudscape in Horizon Zero Dawn. Screenshot of the final game taken on a Playstation 4.

The first method is based on the approach presented by Guerrilla Games as part of a talk at SIGGRAPH 2015 [SV15], which was used in their game "Horizon Zero Dawn". Because of the intended use in a game, it does not primarily attempt to be physically accurate. Instead, a number of visual phenomena observed in real world clouds are replicated:

- direct illumination through single scattering

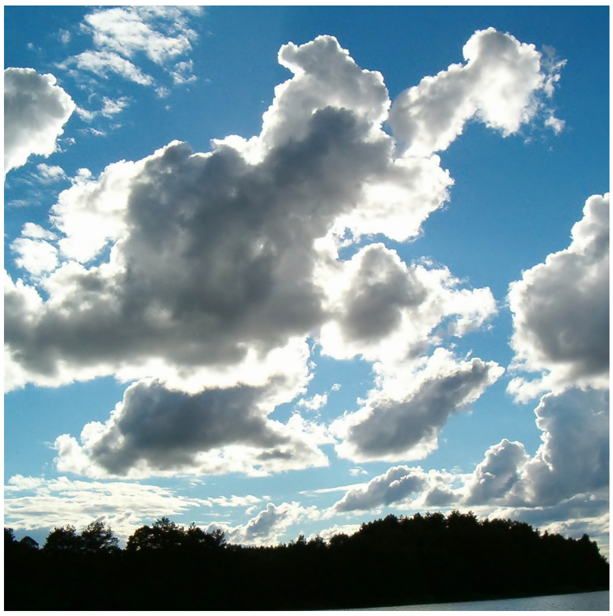


Figure 3.2.: Silver lining when looking at clouds in direction of the sun. One of the phenomena replicated by our simple lighting method. Photo by User:Brosen (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html), CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/) or CC BY 2.5 (http://creativecommons.org/licenses/by/2.5)], via Wikimedia Commons

- silver lining, see fig. 3.2.
- dark edges, see fig. 3.3.

3.1.1. Single Scattering

In order to determine the amount of light that may be scattered directly towards the camera, Beer's Law is used to calculate the transmittance from the light source to a point x in the volume with equation eq. (2.10).

3.1.2. Powder Effect

The observed dark edges, henceforth referred to as the **powder effect**, are a result of multiple scattering inside the clouds. The idea is that at a point x in an area with less dense cloud material, there will be less material from which light could be scattered towards x and thus less in-scattering can be observed. This is illustrated in fig. 3.4. In an application where multiple scattering is directly simulated, this effect would automatically be replicated.

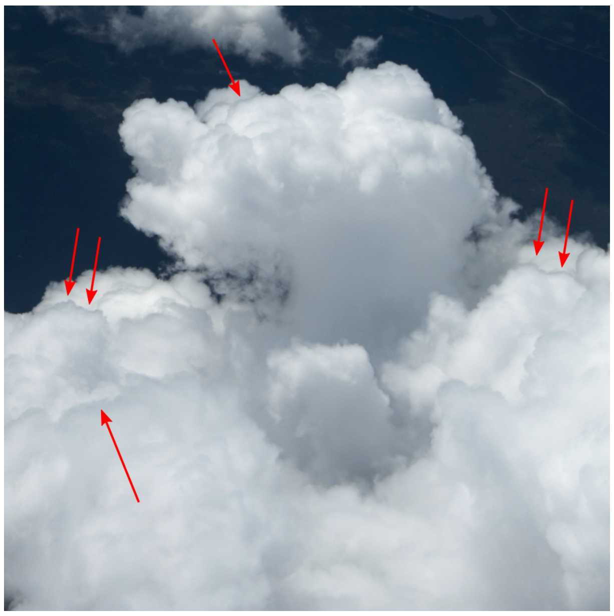


Figure 3.3.: Dark edges of clouds, here referred to as the powder effect. One of the phenomena replicated by our simple lighting method. Photo by Staff Sgt. Stephany Richards U.S. Department of Defense Current Photos (140807-F-IG195-010) [Public domain], via Wikimedia Commons

However, because such an implementation may be impractical to be used in games as the result of a high performance penalty, [SV15] introduces an additional formula:

$$Powder(x_0 \leftrightarrow x) = 1 - e^{-powderDepth * \tau(x_0 \leftrightarrow x)} * powderStrength \tag{3.1}$$

The variable *powderDepth* influences how deep into the cloud the visual effect should reach, i.e. whether only a thin border should be darkened or a larger portion, while *powderStrength* is a factor to adjust the effect to the desired intensity. In the original form as part of [SV15], *powderDepth* is a fixed value of 2 while *powderStrength* is exactly 1.

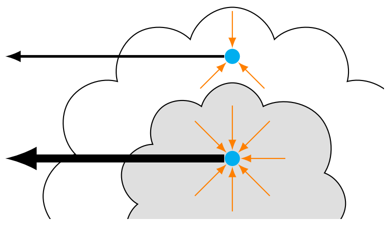


Figure 3.4.: Illustration of the powder effect. At positions with denser surrounding cloud material, a greater amount of in-scattering is observed.

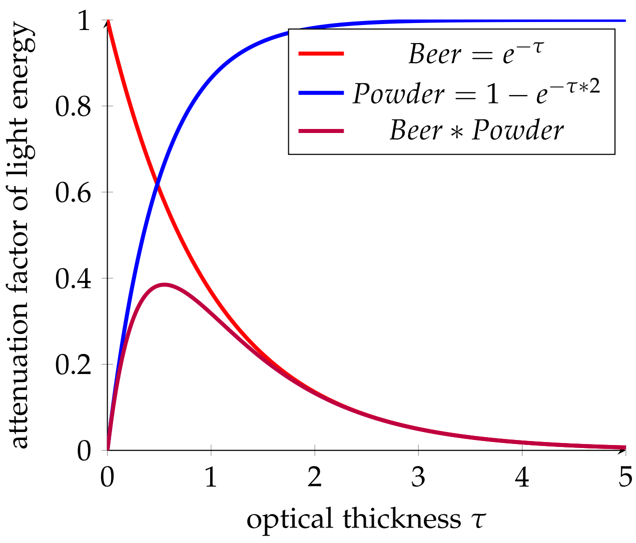


Figure 3.5.: Plot of the Beer and powder functions as well as the combination of both. Beer calculates the transmittance through optical depth τ, powder attenuates areas with low optical thickness from the light, resulting in darkened cloud edges.

The powder term is then directly multiplied to the transmittance calculated using Beer's Law. The resulting function is visualized in fig. 3.5.

3.1.3. Silver Lining

The silver lining effect, as shown in fig. 3.2 appears when the view direction approaches the direction opposite to the light direction. To simulate this behaviour, the Henyey-Greenstein function is used with a value of $g > 0$, which results in a forward-oriented distribution as seen in fig. 2.2c.

3.1.4. Full Lighting

Putting all the terms together results in the following estimation of in-scattered radiance at a point x in direction $\vec{\omega}$, given the light direction $\vec{\omega}_l$ and the medium boundary position x_0:

$$L_i(x \to \vec{\omega}) = \frac{1}{\kappa_s} Tr(x_0 \leftrightarrow x) Powder(x_0 \leftrightarrow x) p_{HG}(\vec{\omega}_l \to \vec{\omega}, g) \tag{3.2}$$

3.2. Photon Mapping

Photon Mapping is a two-pass method that can be used to simulate single and multiple scattering in participating media. In the first pass, photons are shot from the light source, traced through the medium and stored in a data structure. Using the obtained data, the volume can then be rendered with lighting in the second pass.

3.2.1. Photon Points

This is the original photon mapping algorithm, which was introduced by Jensen et al. [JC98] and uses so-called **photon points** stored throughout the medium.

Photon Pass

Depending on the light sources in the scene, a number of photons are created and traced through the medium, each carrying a certain flux Φ_p, starting at a position x_p with a direction $\vec{\omega}_p$.

The probability that a photon interacts with the medium, i.e. being scattered or absorbed, after travelling a distance d is given by the **probability density function** derived from Beer's Law:

$$pdf(x_p \to \vec{\omega}_p, d) = 1 - e^{-\tau(x_p \leftrightarrow (x_p + d\vec{\omega}_p))} \tag{3.3}$$

In a homogeneous medium, this can be simplified to:

$$pdf(d) = 1 - e^{-\kappa_t d} \tag{3.4}$$

From this equation, it is possible to derive a formula to generate values with this probability distribution:

$$d = \frac{log(1-\xi)}{-\kappa_t} \tag{3.5}$$

where $\xi \in [0, 1)$ is a uniformly distributed random value. This is used to generate the distance until the next interaction event in the medium. Whether this event is scattering or absorption is determined using Russian roulette. If the event is determined to be scattering, the new photon direction can obtained by importance sampling of the phase function.

In both cases, the photon is stored as a compound of the event position, its flux and direction in the photon map.

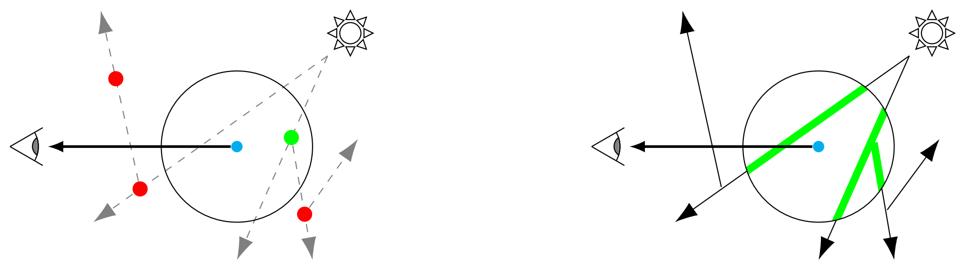

(a) Photon Points (b) Photon Beams

Figure 3.6.: Illustration of the difference between querying photon points and photon beams.

Rendering Pass

After the photon map has been generated, the radiance at a point x can be estimated from it. Therefor, a query volume R, for example a sphere with radius r, around x is chosen, from which photons will be taken into account.

The in-scattering radiance is computed as:

$$L_i(x \to \vec{\omega}) = \frac{1}{\kappa_s * \mu_R} \sum_{p \in R} p(\vec{\omega}_p \to \vec{\omega}) \Phi_p \tag{3.6}$$

with μ_R being the volume of R, for example $\frac{4}{3}\pi r^3$ in the case of a sphere. This query is illustrated in fig. 3.6a.

3.2.2. Photon Beams

Jarosz et al. [Jar+11] extend the photon mapping technique by introducing photon beams.

Photon Pass

When tracing a photon, instead of calculating the distance to the next interaction event and only storing the photon there, the flux is distributed in smaller steps along a complete beam. For simplicity, we will temporarily assume a homogeneous medium and use a fixed step size Δt. Given a start position x_b, start flux Φ_b and a direction $\vec{\omega}_b$, the flux for each deposited photon at x_t is calculated as:

$$\Phi_{p_b} = \kappa_s Tr(x_b \leftrightarrow x_t) \Phi_b \Delta t \tag{3.7}$$

The collection of these photons is called a single **discrete photon beam**. This beam is traced through the entire medium.

The simulation of multiple scattering works very similarly to the original photon tracing method. For each photon beam, a distance to the next interaction event is chosen, as in eq. (3.5), the scattering direction is calculated and a new photon beam is started at the interaction position with its flux calculated as:

$$\Phi_{b_1} = \frac{\kappa_s}{\kappa_t} \Phi_{b_0} \tag{3.8}$$

Rendering Pass

Estimating the radiance from the photon map is again similar to the photon point method. Given the query volume R, all beams intersecting it are chosen and stored photons from these accumulated:

$$L_i(x \to \vec{w}) = \frac{1}{\kappa_s * \mu_R} \sum_{b \in R} \sum_{p_b \in R} p(\theta_{p_b}) \Phi_{p_b} \tag{3.9}$$

In practice, it is not necessary to know which stored photons belong to which discrete photon beam as long as the their positions are known to determine whether they lie in R. Then, this formula will be the same as eq. (3.6).

3.2.3. Regular Photon Grid

Until now, we have assumed the photon map to be an abstract data structure that stores photons with their respective position, direction and flux and can be queried for photons positioned in a given volume. One concrete data structure commonly used for this task is a k-d tree. However, such a structure would be less suited for the highly parallel implementation we are aiming for, for example because of the complexity of the memory layout of such a structure.

As an alternative, Elek et al. [Ele+12] present a way to store photons in a regular grid of only two values per cell, which, in practice, could be easily implemented as a 3D texture. The idea is that the first value is simply the sum of flux in the cell, while the second value represents the directions of the photons.

Henyey-Greenstein Basis

In order to be able to represent the directions of multiple photons by a single value, Elek et al. [Ele+12] make use of the characteristics of the Henyey-Greenstein function depending on its asymmetry factor g, as well as the precondition that a single global directional light source that is responsible for the majority of lighting in the scene, for example the sun, is present.

Every cell in the grid, in addition to the flux, stores a value called the **anisotropy coefficient**, which models the distribution of light outgoing from this cell with respect to the direction of the global light source. This is illustrated in fig. 3.7.

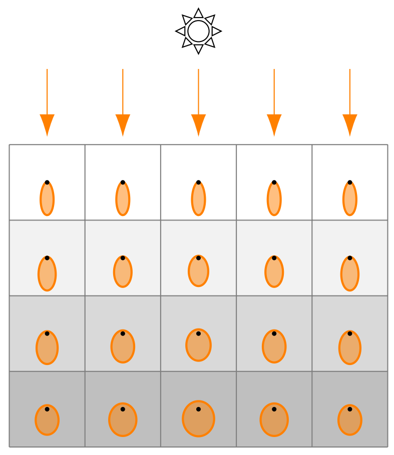


Figure 3.7.: Illustration the anisotropy coefficients saved in the regular grid photon map. In the upper cells, only little incoming light from scattering events is present. Thus, the saved anisotropy coefficient yields a strong forward-oriented light distribution with respect to the global light direction. Further down inside the volume, more light scattered from other directions is encountered, so the saved values approach a more isotropic distribution.

The actual value of the anisotropy coefficient g' is simply the average cosine of the angle between the directions of all photons stored in the cell represented by volume R and the global light direction $\vec{\omega}_l$.

$$g'(R) = \frac{1}{|p \in R|} \sum_{p \in R} \vec{\omega}_p \cdot \vec{\omega}_l \tag{3.10}$$

Reconstruction of In-scattered Radiance

In order to render the volume from a camera direction, we need to be able to estimate the in-scattered radiance at any point x in direction $\vec{\omega}$. Recall the formula for photon points (eq. (3.6)):

$$L_i(x \to \vec{\omega}) = \frac{1}{\kappa_s * \mu_R} \sum_{p \in R} p(\vec{\omega}_p \to \vec{\omega}) \Phi_p \tag{3.11}$$

Ignoring the photon directions for a moment, adapting the calculation of total radiance from our regular grid is easy. As R, we simply choose the volume of a single grid cell and μ_R is calculated accordingly. The sum of flux can directly be taken from the grid and it is even possible to use interpolation to sample at positions other than the cell centers. Thus, the equation is simplified to:

$$L_i(x \to \vec{\omega}) = \frac{1}{\kappa_s * \mu_R} \Phi_{total}(R) * distribution(R \to \vec{\omega}) \tag{3.12}$$

with $distribution(R)$ modelling the distribution of $\Phi_{total}(R)$ to $\vec{\omega}$. Elek et al. [Ele+12] showed that, given the single major directional light source, this distribution can be estimated from the saved anisotropy coefficient by altering the asymmetry factor of the Henyey-Greenstein function:

$$distribution(R) = p_{HG}(\vec{\omega}_l \rightarrow \vec{\omega}, g * g'(R)) \tag{3.13}$$

3.2.4. Woodcock Tracking

So far, we have only seen how to generate the distance to the next interaction event in a homogeneous medium, using eq. (3.5). One way to approximate this distance for a heterogeneous medium is to use a technique called **Woodcock tracking** or **delta tracking** as originally introduced by Woodcock et al. [Woo+65]. The idea is to first determine $\kappa_{t,max}$, the maximum value for the extinction coefficient anywhere in the medium.

When using the photon beams approach, at each step taken for a beam, we first assume the medium has this maximum extinction coefficient everywhere. We replace the fixed value for the step size Δt in eq. (3.7) with a random distance to an interaction event using eq. (3.5):

$$\Delta t = \frac{log(1-\xi)}{-\kappa_{t,max}} \tag{3.14}$$

Next, we move the photon to the destination position x' and store it there. Then, the actual extinction coefficient is sampled and it is probabilistically decided whether a scattering event occurs at this position or not using the following probability:

$$P = \frac{\kappa_t(x')}{\kappa_{t,max}} \tag{3.15}$$

If this determines that an actual event occurs, we call this a **real scattering event**, otherwise we call it a **virtual scattering event**. For a real scattering event we continue tracking and saving the rest of the current beam, as well as emit a new one at x', just like in section 3.2.2. A concrete example on how this works is illustrated in fig. 3.8.

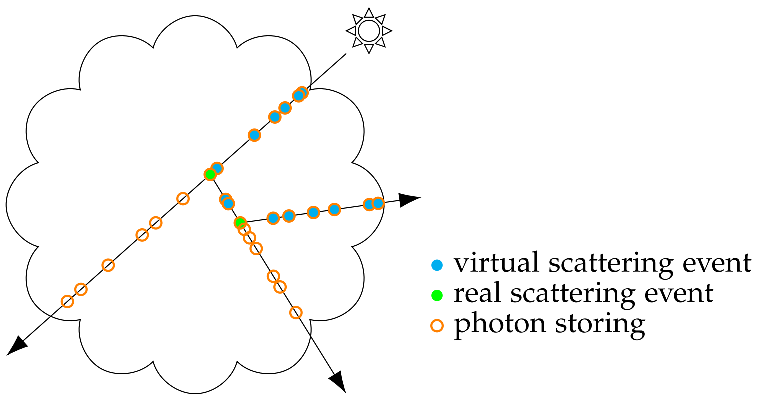


Figure 3.8.: Photon Beams combined with Woodcock Tracking. Random step sizes are used for tracking the beams. At each step, it is determined whether a scattering event occurs and if that is the case, a new beam is emitted.

4. Implementation

We now implement both the simple and the photon mapping method described in chapter 3 as additions to Met.3D. The program, which is free and open-source software available under the GNU General Public License v3, is written in C++ using Qt as a GUI toolkit and OpenGL 4 compatibility profile with preprocessed GLSL from GLFX for rendering. It is organized in so-called **actors**, which can be added to a **scene** and are each responsible for rendering a specific kind of object or data visualization inside of it. For example, there exist actors for rendering a bounding box, a base map or a cross-section of a given three-dimensional data field.

Our Volume Lighting methods build upon the "Volume Raycaster" actor, which can be used to render three-dimensional data as isosurfaces, but also supports direct volume rendering through raycasting.

4.1. Direct Volume Rendering

The direct volume rendering in Met.3D works as follows: The faces of the bounding box around the volume are rendered using the standard OpenGL rasterization pipeline. In the fragment shader, the color that would be seen from the camera at this surface position is calculated. Therefor, a ray starting at the fragment position in world space and going along the camera direction is traced through the volume.

In each step, the data at the current position is sampled and mapped to a color value as well as an extinction coefficient. This mapping can for example be defined by using a transfer function. With the extinction coefficient, Beer's Law is used with the step size to calculate the transmittance of the current step, which is then used as an alpha value in addition to the color. The resulting RGBA colors of all steps are continuously blended together using front-to-back blending.

This produces an image as seen in fig. 4.1.

4.2. Coordinate System Scaling

For rendering a scene, the world coordinate system used in Met.3D is a right-handed coordinate system oriented such that x and y form the ground plane and z is the vertical axis. The values for x and y coordinates are longitude and latitude measured in degrees. The z axis does not directly correspond to any physical unit, but instead the data is scaled by a user-adjustable factor.

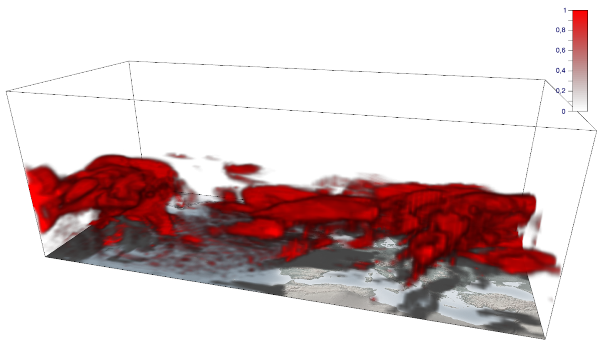


Figure 4.1.: Three-dimensional data rendered without any lighting using DVR and a transfer function in Met.3D.

In order to correctly calculate optical thickness, one has to be aware of the physical units used for distances. Thus, we implement a conversion from world coordinates to meters:

```
#define EARTH_PERIMETER_METERS 40030173.0

uniform float spaceScale;
uniform bool earthCosineEnabled;
uniform float worldZMeterScale;

vec2 lonLatToMeters(vec2 lonLat)
{
    vec2 ret = vec2(
        (lonLat.x / 360.0) * EARTH_PERIMETER_METERS,
        (lonLat.y / 360.0) * EARTH_PERIMETER_METERS
    ) * spaceScale;

    if(earthCosineEnabled)
    {
        ret.x *= cos(lonLat.y * M_PI / 180.0);
    }

    return ret;
}

float worldZToMeters(float z)
{
    return z * worldZMeterScale * spaceScale;
}
```

Converting latitude to the distance from the equator in meters is done simply by scaling it accordingly. For longitude, we also have to account for the fact that the actual distance corresponding to a fixed difference in degrees scales with the cosine of latitude. The z coordinate is scaled by `worldZMeterScale`, which is a factor that has been pre-calculated for the user-defined vertical scaling. The user can select whether to use a realistic vertical scaling derived from pressure altitude or simply the same scaling as used for latitude and longitude, which is not realistic but ensures the same scaling for all axes.

Using the `earthCosineEnabled` flag, we enable the user to disable the cosine effect for longitude, so it is also possible to get a uniform scaling if explicitly wanted. Similarly, we provide the parameter `spaceScale` to add an additional scaling to the whole space. For only altering the effects on the rendered image without affecting lighting calculation, there is also a parameter for using the vertical scale for all axes, as well as another `spaceScale` parameter, which only affects the final volume rendering. This way we can base our calculations on real physical units while still being able to manually control the scaling as desired and being explicit about where such an unnatural scaling happens.

4.3. Determining Extinction Coefficients for Liquid and Ice Water Content

As mentioned above, the extinction coefficients used for rendering and lighting can be defined by the user when using a transfer function that maps values from a single data variable accordingly. However, cloud data is not commonly available as a single variable that directly maps to the real-world extinction coefficient. Instead, the representation in data we will be using is split up in liquid water content *LWC* and ice water content *IWC*, giving the amount of liquid or ice water for per an amount of air. Henceforth, we will use the unit $[\mathrm{kg\,kg^{-1}}]$ for both variables, meaning the mass of water or ice in [kg] per 1 kg of air.

For *LWC*, Hu et al. describe a parameterization that can be used to derive the extinction coefficient [HS93]. The formula we will use is:

$$\kappa_{t,LWC}[\mathrm{m^{-1}}] = \frac{3}{2}\frac{LWC[\mathrm{kg\,kg^{-1}}] * \rho_{air}[\mathrm{kg\,m^{-3}}]}{\rho_{water}[\mathrm{kg\,m^{-3}}] * r_{effective,LWC}[\mathrm{m}]} \tag{4.1}$$

with $\rho_{air} \approx 1.225\,\mathrm{kg\,m^{-3}}$ being the density of air, $\rho_{water} \approx 1000\,\mathrm{kg\,m^{-3}}$ the density of water and $r_{effective,LWC}$ the so-called **mean effective radius**, a weighted mean of the size distribution of droplets.

Similarly, a formula for *IWC* exists, as described by Fu [Fu96]:

$$\kappa_{t,LWC}[\mathrm{m}^{-1}] = \frac{4 * \sqrt{3}}{3} \frac{IWC[\mathrm{kg\,kg}^{-1}] * \rho_{air}[\mathrm{kg\,m}^{-3}]}{\rho_{ice}[\mathrm{kg\,m}^{-3}] * r_{effective,IWC}[\mathrm{m}]} \tag{4.2}$$

analogous, with $\rho_{ice} \approx 917\,\mathrm{kg\,m}^{-3}$ being the density of ice and $r_{effective,IWC}$ the effective radius for ice droplets.

We use constant values for all required densities and let the effective radii be user-configurable values in [µm], so they have to be divided by a factor of 1000000 to be converted to meters. The resulting GLSL code, using input data that is already converted to [$\mathrm{kg\,m}^{-3}$], is the following:

```
uniform float effectiveRadiusLWCum;
uniform float effectiveRadiusIWCum;

float computeLWCExtinctionCoefficient(float lwcKgPerM3)
{
    float effectiveRadiusMeter = effectiveRadiusLWCum / 1000000.0;
    float densityOfWaterKgPerM3 = 1000.0;
    return 1.5 * lwcKgPerM3
        / (densityOfWaterKgPerM3 * effectiveRadiusMeter);
}

float computeIWCExtinctionCoefficient(float iwcKgPerM3)
{
    float effectiveRadiusMeter = effectiveRadiusIWCum / 1000000.0;
    float densityOfIceKgPerM3 = 917.0;
    return (4.0 * sqrt(3.0) / 3.0) * iwcKgPerM3
        / (densityOfIceKgPerM3 * effectiveRadiusMeter);
}
```

Because in clouds, there is almost no absorption, but nearly pure scattering, we will assume $\kappa_t = \kappa_s$ in this implementation.

4.4. Simple Lighting

In order to implement the simple lighting method described in section 3.1, we restrict the light direction to always be exactly $(0, 0, -1)$ in world space, i.e. along the vertical axis. This assumption gives us some major advantages when implementing the method as seen hereafter.

4.4.1. Optical Thickness Buffer

We create an additional 3D texture with a user-defined resolution carrying a single 32-bit float component. This texture is uniformly mapped to the bounding box used for rendering the volume and will contain the total optical thickness between the light source and the respective position. The z texture coordinate is oriented so that 0 corresponds to the top position and the maximum value is at the bottom in world space.

The calculation of the optical thickness to be saved in a cell with coordinate (x, y, z) is the integration of the extinction coefficient from the light source. Because of our fixed light direction, this calculation can also be done recursively:

$$\tau(x,y,z) = \tau(x,y,z-1) + \int_{(x,y,z-1)}^{(x,y,z)} \kappa_t(y)dy \tag{4.3}$$

We approximate the integral by sampling the extinction coefficient at (x, y, z) and multiplying it with the vertical distance between the two cells in meters.

Thus, we can implement the creation of the whole texture as multiple parallel processes, one for each (x, y) coordinate, iteratively calculating the value for each z coordinate. We do this using a compute shader with the following (simplified) GLSL code:

```glsl
layout(r32f, binding = 0) uniform image3D lightingMapOut;

shader CSSimpleLighting()
{
    float stepSizeMeters = // ...

    ivec3 texelCoord = ivec3(gl_GlobalInvocationID.xy, 0);
    float currentOpticalThickness = 0.0;

    while(texelCoord.z < size.z)
    {
        vec3 worldPos = getLightingWorldPositionFromTexelCoord(texelCoord);


        float extCoeff = sampleExtinctionCoefficient(worldPos);
        currentOpticalThickness += stepSizeMeters * extCoeff;

        imageStore(lightingMapOut, texelCoord, vec4(
            currentOpticalThickness, vec3(0.0)));

        texelCoord.z++;
    }
}
```

`gl_GlobalInvocationID.xy` carries the x and y coordinates to be used. Because each invocation of this compute shader works on its very own part of the texture, we can use `imageStore()` to save the values here without the need for any kind of synchronization.

4.4.2. Rendering

We integrate the lighting method into the volume renderer by replacing the color fetched from the transfer function by an estimation of radiance as described in section 3.1.4:

```
uniform float volumeLightingIntensity;
uniform float volumeLightingAmbient;
uniform float volumeLightingHenyeyGreensteinG;
uniform float volumeLightingPowderStrength;
uniform float volumeLightingPowderDepth;

vec3 volumeLightingAtPos(vec3 rayDir, vec3 pos)
{
    float opticalThickness = sampleLightingVolumeAtPos(pos);

    float beer = exp(-opticalThickness);
    float powder =
        1.0 - exp(-volumeLightingPowderDepth * opticalThickness)
              * volumeLightingPowderStrength;

    float lightCos = dot(-rayDir, vec3(0.0, 0.0, -1.0));
    float hg = calculateHenyeyGreensteinCos(lightCos,
        volumeLightingHenyeyGreensteinG);

    float lightingValue = 2.0 * beer * clamp(powder, 0.0, 1.0) * hg;

    return vec3(lightingValue * volumeLightingIntensity +
        volumeLightingAmbient);
}
```

`volumeLightingAmbient` is an additional user-defined factor that can be used to add an additional ambient lighting component.

4.5. Photon Mapping

In our implementation, we use the regular grid photon map described in section 3.2.3 in combination with photon beams (section 3.2.2) and Woodcock tracking (section 3.2.4) as proposed by Elek et al. [Ele+12] and implement the tracing of photons in compute shaders. Because in the photon pass, we do not only trace photons along the exact light direction, the fixed light direction as used for our simple method loses the advantages described above and we can instead allow an arbitrary light direction.

4.5.1. Pseudo Random Number Generation in GLSL

Because this method requires the generation of random numbers, we need to implement a pseudo random number generator in GLSL that reliably generates uniformly distributed values. We use the approach proposed by Howes et al., which is a combination of a combined Tausworthe generator and a linear congruential generator [HT07]:

```
uint rand_z1, rand_z2, rand_z3, rand_z4;

uint randTausStep(inout uint z, int S1, int S2, int S3, uint M)
{
   uint b = ((z << S1) ^ z) >> S2;
   return z = ((z & M) << S3) ^ b;
}

uint randLCGStep(inout uint z, uint A, uint C)
{
   return z = A*z + C;
}

float randHybridTaus()
{
   return 2.3283064365387e-10 * float(
      randTausStep(rand_z1, 13, 19, 12, 4294967294U) ^
      randTausStep(rand_z2, 2, 25, 4, 4294967288U) ^
      randTausStep(rand_z3, 3, 11, 17, 4294967280U) ^
      randLCGStep(rand_z4, 1664525, 1013904223U)
   );
}
```

For setting the initial values of `rand_z1`, `rand_z2`, `rand_z3` and `rand_z4`, we use the approach by Mohanty et al. [MMC12] with an unsigned integer value as the seed, which could for example be the invocation id of a compute shader:

```
void randSetSeed(uint seed)
```

```
{
    rand_z1 = seed * 1099087573U;
    rand_z2 = rand_z1 * 1099087573U;
    rand_z3 = rand_z2 * 1099087573U;
    rand_z4 = rand_z3 * 1099087573U;
}
```

4.5.2. Photon Pass

For the regular grid photon map, we create a 3D texture carrying two 32-bit float components for the total flux and anisotropy coefficient. To build this map, we use a compute shader that traces one photon in each invocation. Unfortunately, as mentioned before, we do not know beforehand where each invocation will store values in the texture and multiple invocations might also write to the same location. Thus, it is necessary to synchronize the write access to the texture. Therefor, we use OpenGL's atomic image operations, in particular `imageAtomicAdd()`. This imposes a severe limitation on the image we write to, namely the restriction that the image can only be of the format `GL_R32I` or `GL_R32UI`. Another issue is that we can not directly compute the average value for the anisotropy coefficient by only storing the values one after another.

In order to still generate the desired two-component float texture, we split the generation of the map into two passes: We create three additional 32-bit integer textures for total photon flux, the sum of all anisotropy coefficients and the number of photons stored in the cell. We use the contents of these textures as fixed-point values and run the tracing compute shader on them. In the second pass, we convert the textures to our desired format. This is again done by a compute shader, which, in each invocation, divides the total anisotropy coefficient by the photon count and stores the resulting value along with the total flux in the destination texture.

Tracing Photons

In the tracing compute shader, we first define a struct to store the state of a photon:

```
struct PhotonState
{
    vec3 posMeters;
    vec3 posWorld;
    vec3 dirMeters;
    float flux;
};
```

We will use posWorld to find the respective destination coordinates in the photon map as well as sampling the input data while posMeters and dirMeters are used for the actual lighting calculations. The two positions will be kept in sync.

The following function is used to store such a photon, as part of a photon beam, in the map:

```
const float PHOTON_MAP_NORMALIZATION_VALUE = 128.0;

uniform vec3 volumeLightingDirection;

layout(r32ui, binding = 0) uniform uimage3D photonFluxMapOut;
layout(r32i, binding = 1) uniform iimage3D photonCosineMapOut;
layout(r32ui, binding = 2) uniform uimage3D photonCountMapOut;

void depositPhoton(PhotonState photon, float stepDistMeters)
{
    vec3 dirWorld = normalize(photon.posWorld
        - metersToWorldPos(photon.posMeters + photon.dirMeters));
    float photonCosine = dot(volumeLightingDirection, dirWorld);
    float cosineValue = photonCosine * PHOTON_MAP_NORMALIZATION_VALUE;

    float fluxValue = photon.flux * PHOTON_MAP_NORMALIZATION_VALUE *
        stepDistMeters;

    ivec3 coord = getLightingTexelCoordFromWorldPos(photon.posWorld);
    imageAtomicAdd(photonFluxMapOut, coord, uint(fluxValue));
    imageAtomicAdd(photonCosineMapOut, coord, int(cosineValue));
    imageAtomicAdd(photonCountMapOut, coord, uint(1));
}
```

The Woodcock tracking method is implemented as one function that moves the photon for a single step and samples the extinction coefficient at the destination and another function that also evaluates whether a real scattering event occurs:

```
void woodcockStepVirtual(inout PhotonState photon, in float maxExtCoeff,
                         out float currentExtCoeff, out float distMeters)
{
    distMeters = -log(1.0 - randHybridTaus()) / maxExtCoeff;
    photon.posMeters += photon.dirMeters * distMeters;
    photon.posWorld = metersToWorldPos(photon.posMeters);
    currentExtCoeff = sampleExtinctionCoefficient(photon.posWorld);
}
```

```
bool woodcockStepComplete(inout PhotonState photon, in float maxExtCoeff,
                          out float currentExtCoeff, out float distMeters)
{
    woodcockStepVirtual(photon, maxExtCoeff, currentExtCoeff, distMeters);

    if(currentExtCoeff > 0.0)
        return randHybridTaus() < currentExtCoeff / maxExtCoeff;
    else
        return false;
}
```

To randomly choose a scattering direction with the distribution given by the Henyey-Greenstein function, we use the formula explained in [Har] as part of the following function, which generates a cosine for the scattering direction given a random value r and the asymmetry factor for the Henyey-Greenstein function:

```
float sampleHenyeyGreenstein(float r, float g)
{
    float a = (1.0 - g*g) / (1.0 + g * (2.0*r - 1.0));
    return (1.0 / (2.0 * g)) * (1.0 + g*g - a*a);
}
```

The compute shader is then implemented as the following code:

```
#define PHOTON_FLUX_THRESHOLD 0.01;

uniform float volumeLightingMaxExtinctionCoeff;

shader CSPhoton()
{
    randSetSeed(gl_GlobalInvocationID.x);

    PhotonState photon;
    photon.posWorld = randomPhotonEntry();
    photon.posMeters = worldPosToMeters(photon.posWorld);
    photon.dirMeters = normalize(worldPosToMeters(photon.posWorld +
        volumeLightingDirection) - photon.posMeters);
    photon.flux = 1.0;

    while(positionInLightingVolume(photon.posWorld))
    {
        float distMeters;
        float extCoeff;
        bool realScatterEvent = woodcockStepComplete(photon,
            volumeLightingMaxExtinctionCoeff, extCoeff, distMeters);

        photon.flux *= transmittance(distMeters, extCoeff);

        depositPhoton(photon, distMeters);

        if(realScatterEvent)
        {
            trackRestPhotonBeam(photon);

            photon.flux = 1.0;
            float scatterCos = sampleHenyeyGreenstein(randHybridTaus(),
                volumeLightingHenyeyGreensteinG);
            photon.dirMeters = scatterDirection(photon.dirMeters,
                scatterCos);
        }

        if(photon.flux < PHOTON_FLUX_THRESHOLD)
            break;
    }
}
```

We continuously step along the current photon beam using Woodcock tracking until a real scattering event occurs. In this case, the rest of the current beam is first traced and saved to the map, then a new beam is emitted at the scattering location with the flux being reset and the scattering direction calculated using `sampleHenyeyGreenstein()`.

The function `trackRestPhotonBeam()` essentially performs the same tracing, but without generating real scattering events:

```
void trackRestPhotonBeam(in PhotonState photon)
{
    while(positionInLightingVolume(photon.posWorld))
    {
        float distMeters;
        float extCoeff;
        woodcockStepVirtual(photon, volumeLightingMaxExtinctionCoeff,
            extCoeff, distMeters);

        if(extCoeff >= 0.0)
            photon.flux *= transmittance(distMeters, extCoeff);

        depositPhoton(photon, maxExtCoeff);
        if(photon.flux < PHOTON_FLUX_THRESHOLD)
            break;
    }
}
```

The initial flux of the photon depends on a number of factors, such as the power of the light source and the number of emitted photons in order to keep the total amount of flux of all photons constant for a varying number of photons, however all these aspects can be combined into a single multiplication. Instead of applying these factors in the tracing shader, we simply start with a flux of 1 and postpone the scaling to the estimation of radiance later. This is a great benefit for our implementation, because, since we use fixed-point values in the temporary textures, the precision of the values in these maps is more predictable.

In addition, a tiny threshold `PHOTON_FLUX_THRESHOLD` is used to stop tracing the beam, as proposed in [Ele+12].

4.5.3. Rendering

When actually rendering the volume, the following function is used to estimate the radiance:

```
uniform float volumeLightingIntensity;
uniform vec3 volumeLightingDirection;
uniform float volumeLightingHenyeyGreensteinG;

vec3 volumeLightingAtPos(vec3 rayDir, vec3 pos)
{
    float lightingValue = samplePhotonMapAtPos(pos).r;
    float cosine = samplePhotonMapAtPos(pos).g;

    lightingValue /= getLightingTexelVolumeInMetersAtWorldPos(pos);

    float lightCos = dot(-rayDir, volumeLightingDirection);
    lightingValue *= calculateHenyeyGreensteinCos(lightCos,
        cosine * volumeLightingHenyeyGreensteinG);
    lightingValue *= volumeLightingIntensity;

    return vec3(lightingValue + volumeLightingAmbient);
}
```

At this point, `volumeLightingIntensity` is already divided by the total number of photons and gives the actual amount of flux emitted for each photon. The photon map is sampled and the radiance is then estimated just as descibed in eq. (3.12).

5. Results

We will now present a number of images rendered in Met.3D using our methods and examine the visual outcome as well as measure performance.

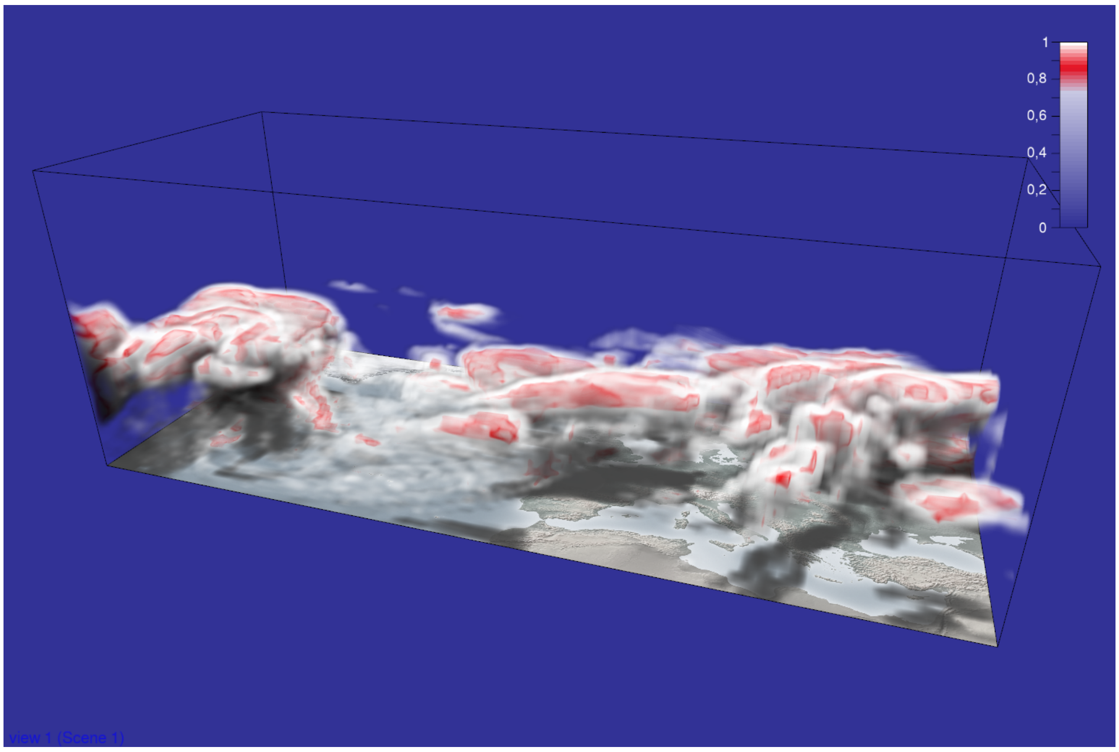
All examples were executed on Arch Linux with an Intel Core i7 4790k CPU, 16GB of RAM and an NVIDIA GeForce GTX 1080. Time measurements were taken using `gettimeofday()` with an invocation of `glFinish()` before and after each measured process on the GPU. This ensures all GPU work issued by calls before starting the work that is of interest has been finished before starting to measure time as well as that all work that should be measured has already finished when stopping to measure time. While this explicit synchronization of CPU and GPU may in practice lead to parts of the GPU's resources to be unused while they could be used without the calls to `glFinish()`, thus leading to more total time taken for the complete work, it does ensure that what is measured is exactly the time taken for the work that we are interested in.

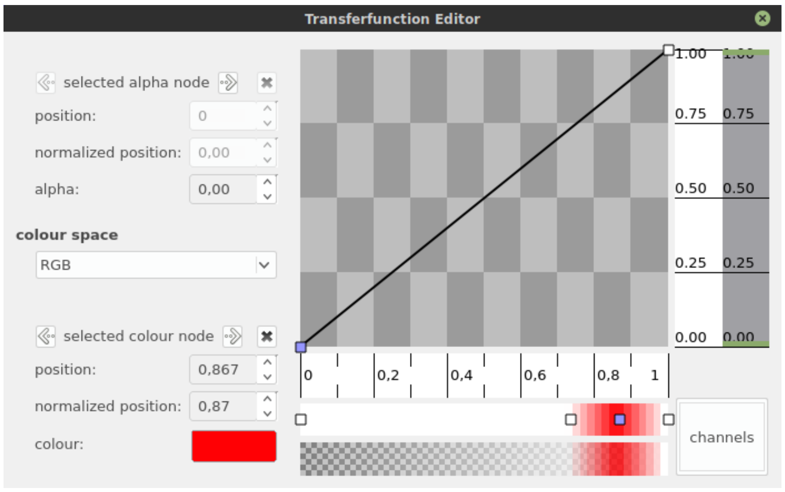
5.1. Simple

5.1.1. Transfer Function

Figure 5.1 shows an image rendered using data from the ECMWF Ensemble Prediction System (ENS), specifically a prediction of the fraction of cloud cover on Mo. 2012-10-15 00:00 UTC. The original data, which gives the fraction of the respective grid cell that is covered by clouds as values from 0 to 1, is mapped to both a color and a value for the extinction coefficient using a transfer function. Lighting is applied in addition to the mapped color using our simple method and the mapped extinction coefficient. The settings used are the following:

account for earth's curvature	Off
use real vertical scale	Off
space scale	0.001
intensity	7.0
ambient	0.0
density scale	0.003
powder strength	1.0
powder depth	100.0
Henyey-Greenstein g	0.0
map resolution (x, y, z)	256x128x128





selected alpha node
position: 0
normalized position: 0,00
alpha: 0,00

colour space
RGB

selected colour node
position: 0,867
normalized position: 0,87
colour: 🟥

Figure 5.1.: Fraction of cloud cover directly mapped to the extinction coefficient using a transfer function

5.1.2. Liquid and Ice Water Content

Instead of mapping cloud coverage using a transfer function, we will now derive the extinction coefficients from LWC and IWC data, coming from the same prediction as the data used above. The resulting image can be seen in fig. 5.2a with all settings being the same as in the previous example and using an effective radius of 10 µm for *LWC* and 25 µm for *IWC* in the extinction coefficient calculation. While, because of the space scale of a value other than 1 as well as the unrealistic vertical scaling, this is not a physically accurate visualization of clouds as they would be visible when taking a photograph, the transparency of the clouds is still directly proportional to the real transparency.

5.1.3. Powder Effect

Figure 5.2 shows the contribution of the powder effect to the overall image. Compared to fig. 5.2b, where the effect has been completely disabled, the top edges along the clouds in fig. 5.2a are slightly darkened, serving as a visual cue for the volume's structure.

Figure 5.2c shows the effect of the powder depth value. The lower the value, the deeper the effect reaches into the volume.

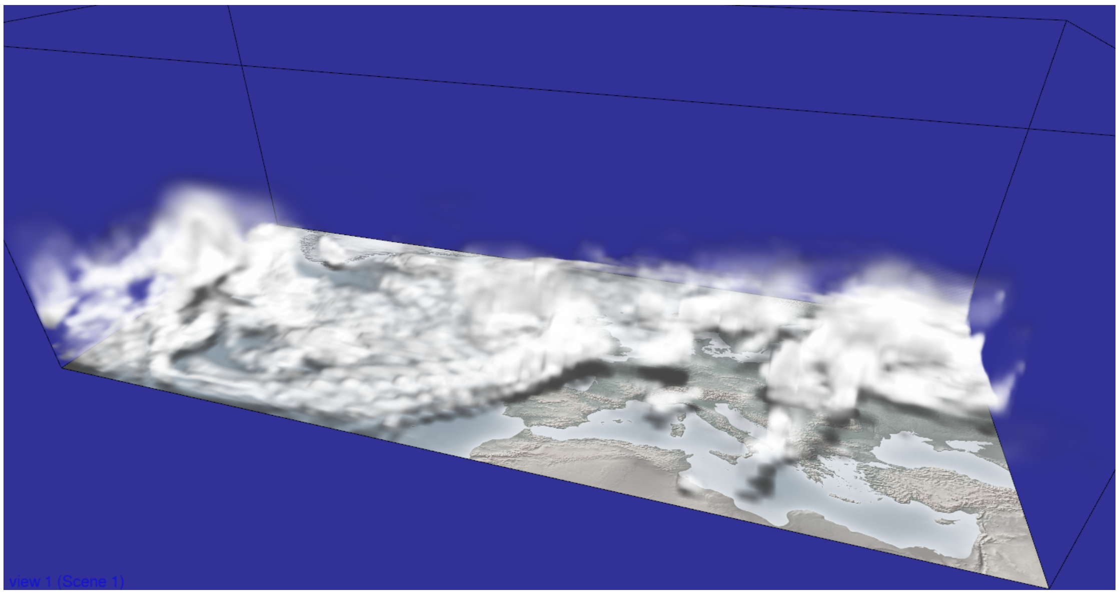
5.1.4. Henyey-Greenstein Component

Until now, we have simply used 0.0 as the anisotropy factor for the Henyey-Greenstein component of the simple lighting method, thus essentially disabling it. Figure 5.3 shows rendered images with a value of 0.7 instead. It can be clearly seen that more light is simulated to be scattered in directions closer to the original light direction, which leads to the desired silver-lining phenomenon.

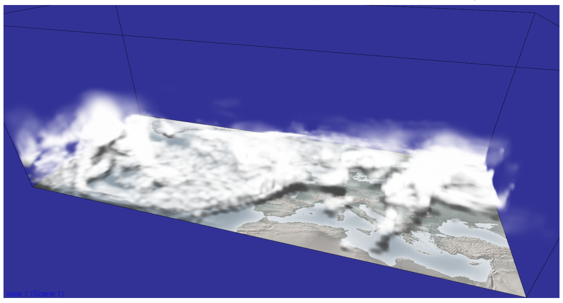
It should be noted, however, that this effect is the same for any point in the volume since it only accounts for single scattering, i.e. scattered light initially coming directly from the light source.

5.1.5. Performance and Effects of Resolution

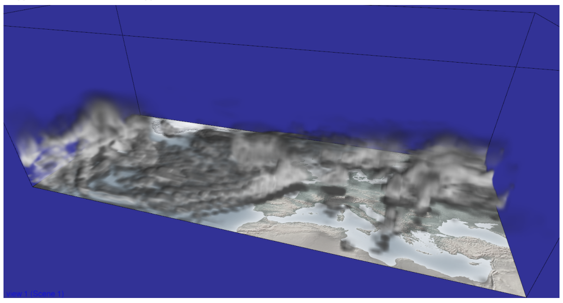
Using the settings from section 5.1.2 while varying the resolution used for the optical thickness buffer, the following durations for generating the buffer were measured:



(a) powder strength = 1.0, powder depth = 100.0

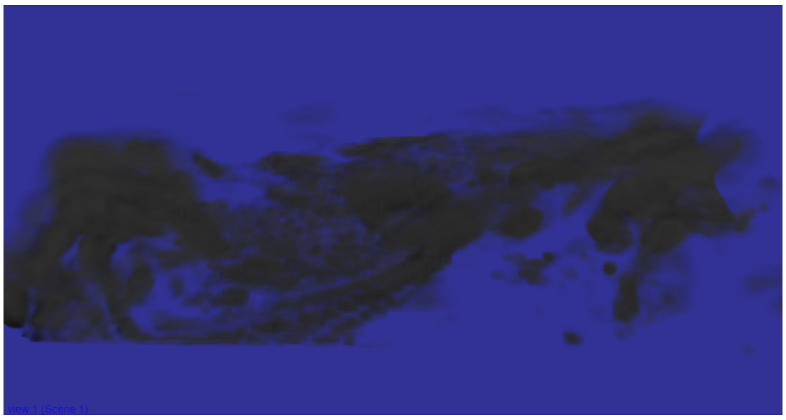


(b) powder strength = 0.0, powder depth = 100.0

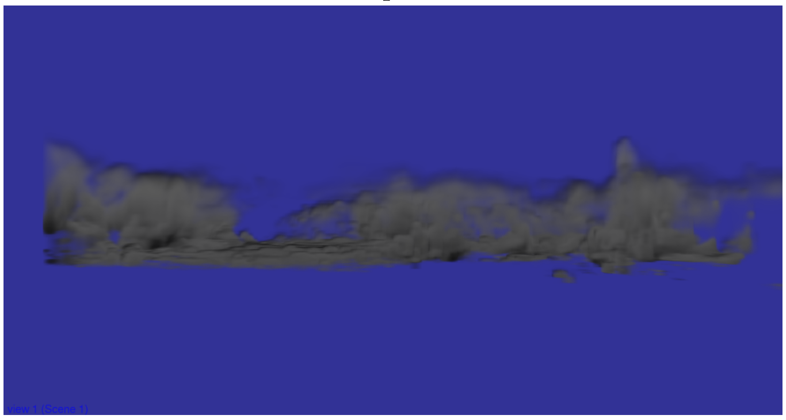


(c) powder strength = 1.0, powder depth = 10.0

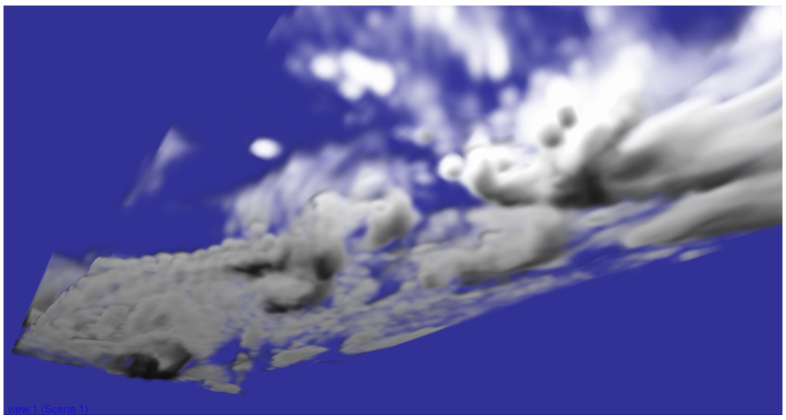
Figure 5.2.: Volume rendered with extinction coefficients derived from LWC and IWC. Lighting is simulated using our simple method with different settings for the powder effect.



(a) top view



(b) side view



(c) bottom view

Figure 5.3.: Demonstration of the effect of the Henyey-Greenstein function in the simple lighting method.

map resolution (x, y, z)	measured time (ms)
32x16x16	0.20
64x32x32	1.53
128x64x64	9.50
256x128x128	72.86
512x256x256	568.06
1024x512x512	4515.85

All resulting images can be seen in appendix A.1.

5.1.6. Proposal for the Illustration of a Scientific Publication

Our simple method has been used to render the volume in fig. 5.4, which is a proposal for an illustration to be used as part of the publication of the paper "Flow-dependent reliability: A path to more skillful ensemble forecasts" by Rodwell et al. [Rod+18]. The isosurface shows wind speed, while the volume visualizes convective tendency.

For this particular render, the simple method has been slightly modified. The powder effect at any point is not calculated depending on the optical thickness towards the light source, but instead on the extinction coefficient at the current position. The isosurface is also being taken into account when generating the optical thickness buffer in order to produce the visible shadow cast by the isosurface on the volume. In addition, direct volume rendering has been combined with the rendering of the isosurface in a single raycasting pass in order to correctly account for alpha blending where both volumes visually overlap or intersect.

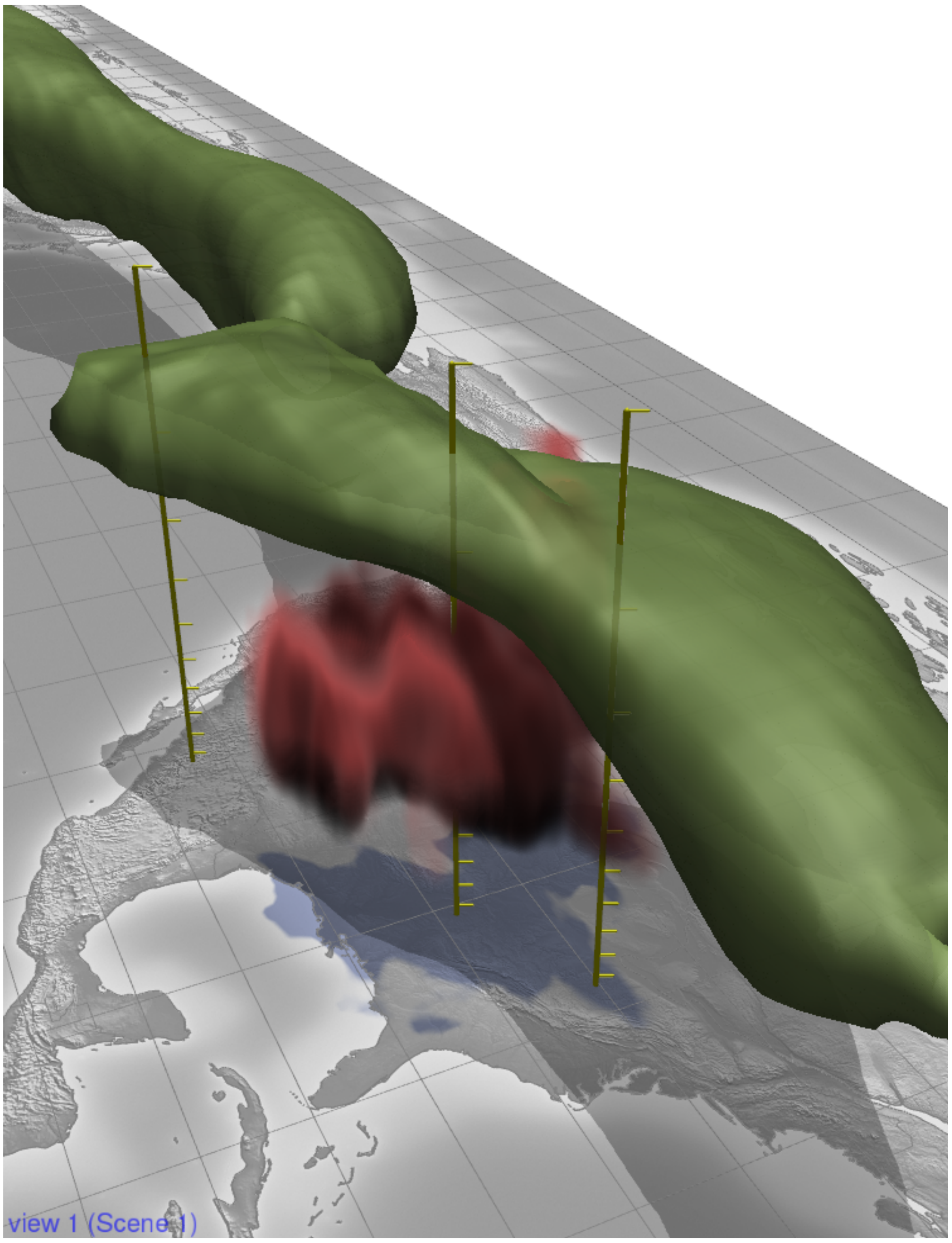


Figure 5.4.: Proposal for an illustration to be used as part of the publication of the paper “Flow-dependent reliability: A path to more skillful ensemble forecasts” by Rodwell et al. [Rod+18], rendered using our simple lighting method

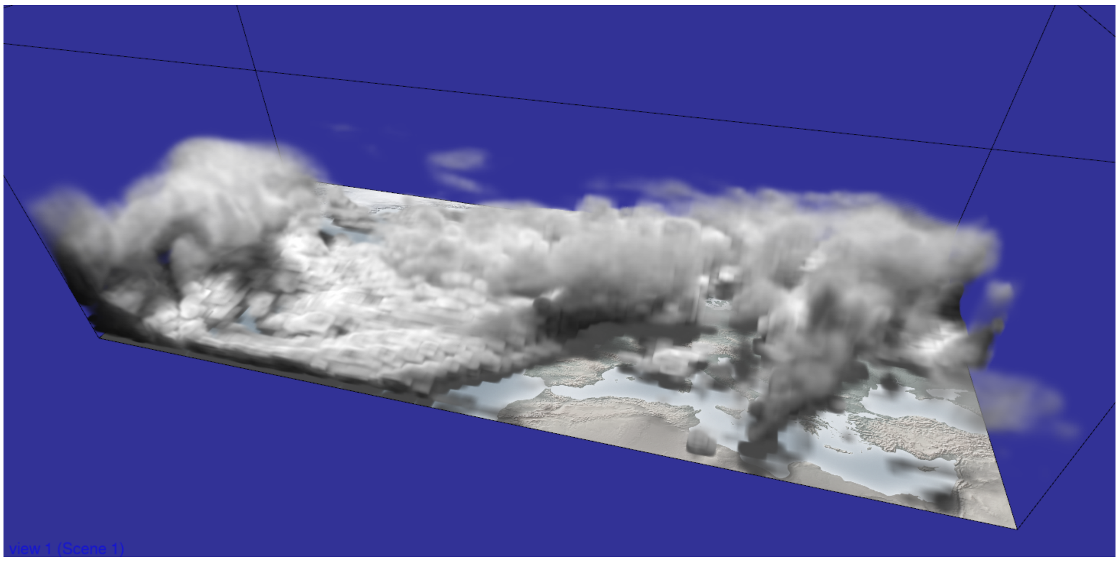


Figure 5.5.: An image rendered with the photon mapping method and isotropic scattering

5.2. Photon Mapping

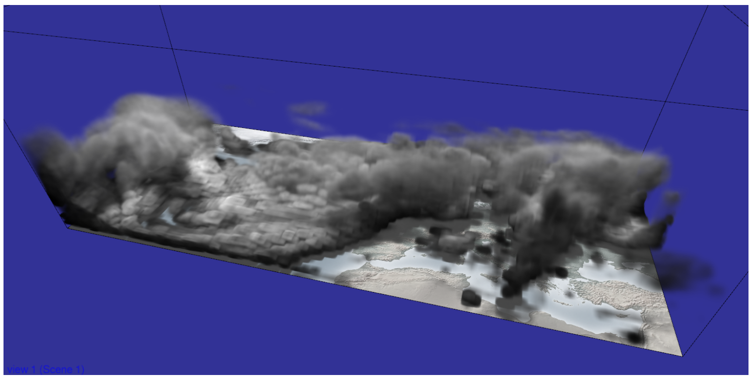
Figure 5.5 shows an image rendered with the photon mapping method and the following settings:

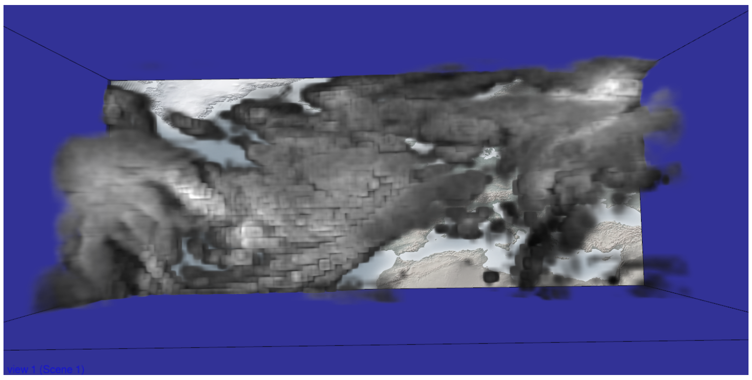
account for earth's curvature	Off
use real vertical scale	Off
space scale	0.01
intensity	0.0007
ambient	0.0
photon count	1,000,000
max extinction coefficient (for Woodcock tracking)	0.01
Henyey-Greenstein g	0.0
light direction (x, y, z)	(0, 0, -1)
photon map resolution (x, y, z)	128x64x64

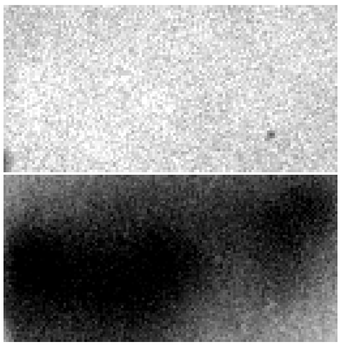
5.2.1. Effects of Scattering

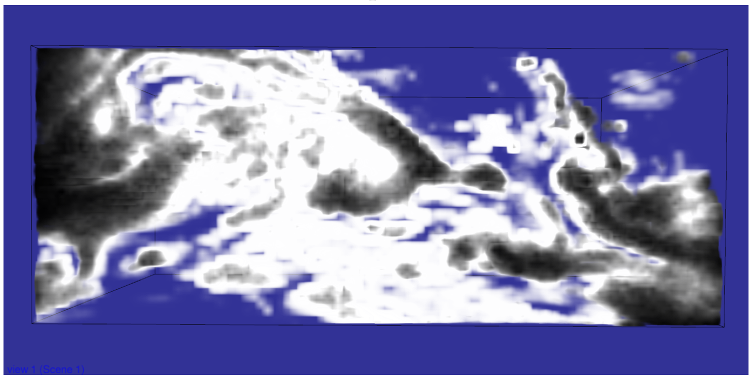
As opposed to our simple method, the photon mapping method does directly simulate multiple scattering instead of only trying to imitate some of its visual phenomena. While the previous example used an anisotropy factor of 0, we will now raise it to 0.85 to cause forward-oriented scattering and analyze the results.

The rendered images, as well as parts of the raw photon map data, which have been


(a) Side view


(b) Top view


(c) Slice with z=34 from the photon map


(d) Bottom view

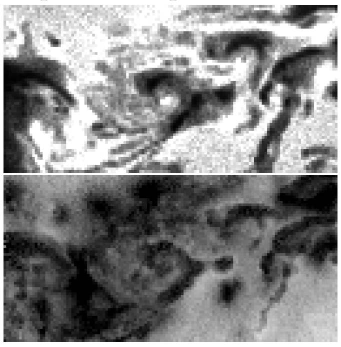

(e) Slice with z=58 from the photon map

Figure 5.6.: Demonstration of the effects of scattering in the photon mapping method. All three rendered images have been created with the exact same settings and are only viewed from different angles. Figure 5.6c and fig. 5.6e show horizontal slices from the raw data of the photon map of size (128, 64, 64). The top image in these figures shows flux while the bottom one displays the anisotropy coefficient.

extracted from the running application using RenderDoc [RD], can be seen in fig. 5.6. It can be observed that, from the top view in fig. 5.6b, parts of the volume that are inside a denser area appear brighter than edges of the volume as well as overall thinner areas. This is because in denser areas, more scattering appears, thus raising the probability of a photon changing its direction to fly back towards the light when taking into account the forward-oriented scattering we use. This is in our case simulated by the anisotropy coefficient component of the photon map, as shown in the bottom part of fig. 5.6c. Here, darker pixels mean lower values, thus a more isotropic distribution of scattered light while in brighter areas, the distribution still approaches the original forward-oriented characteristic.

Conversely, the inverse phenomenon can be seen in fig. 5.6d. When looking from the bottom, thinner areas, thus areas that still mostly employ the original forward-oriented distribution, appear much brighter, which results in the silver-lining effect.

Comparison with Simple

As mentioned before, the simple method applies the Henyey-Greenstein function with the same asymmetry factor to the whole volume. The photon mapping method, however, directly accounts for the effects caused by multiple scattering at any point in the volume, thus producing a result that is much more accurate to real-world scattering than the simple method.

While the resulting silver lining phenomenon seen in fig. 5.3c may look similar to the one in fig. 5.6d, the results seen from the top in fig. 5.3a clearly show this weakness when being compared to fig. 5.6b.

The effect of less light being scattered back towards the light is only modeled using the powder effect in the simple method, which, as mentioned, is merely an imitation of the observed visual outcome rather than the result of an actual simulation.

5.2.2. Real-World Scaling

We will now use the conversion to real length units in order to render an image which is as physically realistic as it can be with our method. For this example, we will use a different data set with a higher resolution. Specifically, this is ECMWF analysis data of Mo. 2016-09-26 12:00 UTC. The following settings were used to produce the results seen in fig. 5.7:

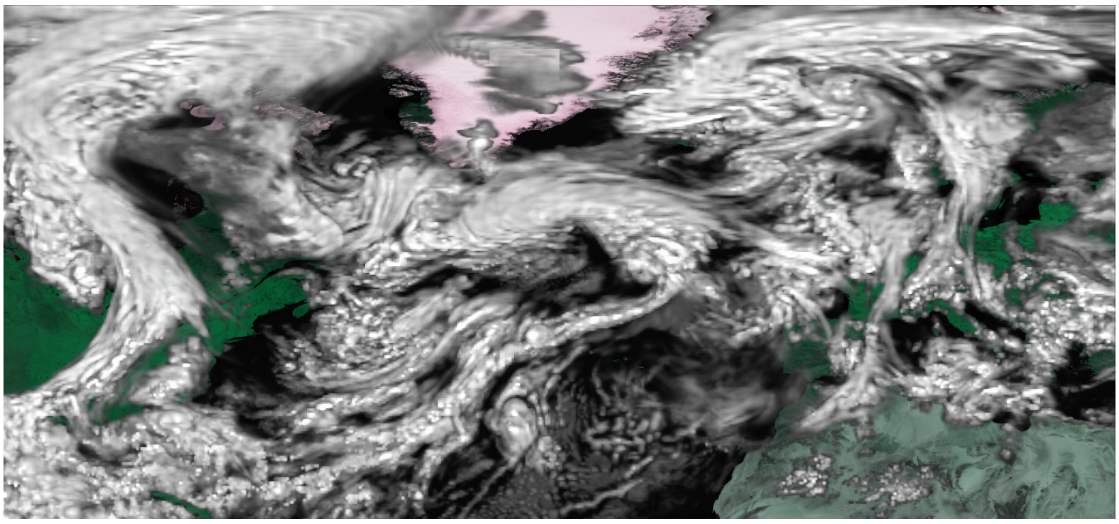
account for earth's curvature	On
use real vertical scale	On
space scale	1.0
force uniform scale for view	On
effective radius lwc	10 µm
effective radius iwc	25 µm
intensity	13.0
ambient	0.0
photon count	10,000,000
max extinction coefficient (for Woodcock tracking)	0.01
Henyey-Greenstein g	0.3
light direction (x, y, z)	(0, 0, -1)
photon map resolution (x, y, z)	128x64x64

Because of the excessive vertical scale in relation to the horizontal one that is used when rendering the volume in 3D, it would normally look more opaque than one would expect when looking at it from any direction other than a perfectly vertical one. Think of a very big and flat volume being compressed to a cube while still remaining all of its opaqueness. Because of this, an orthographic projection is used to render images seen from the top.

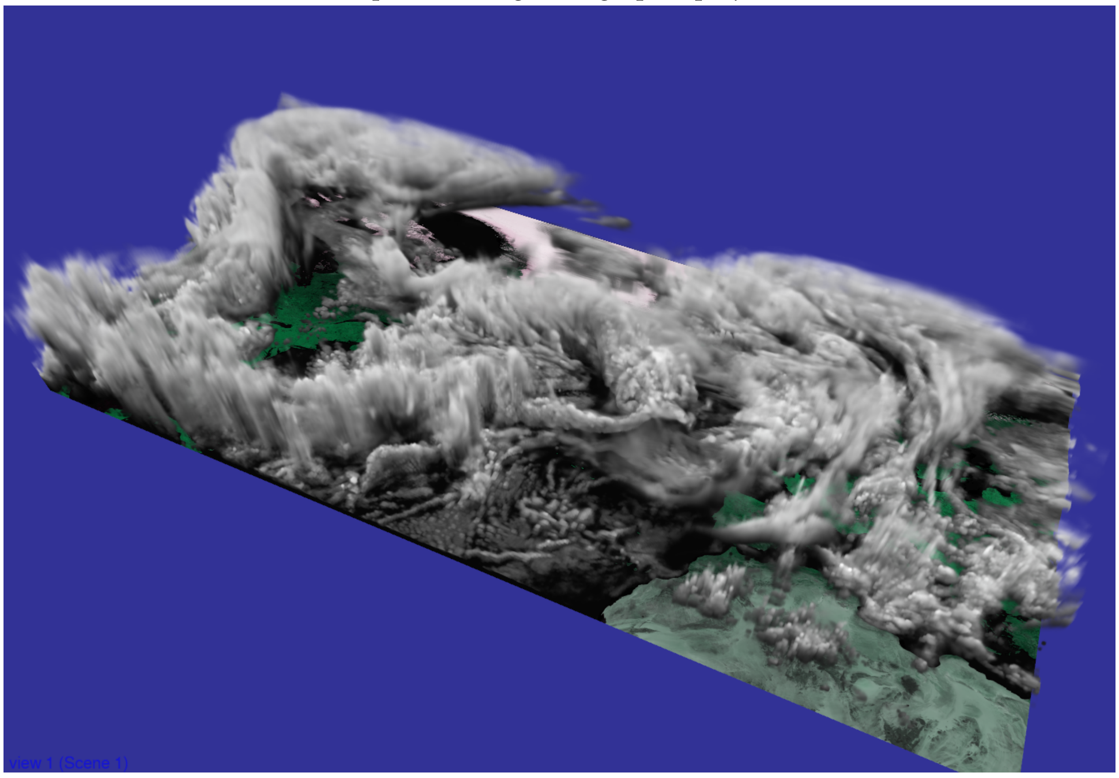
In order to still be able to produce images seen from other angles, the option "force uniform scale for view" is used to apply the scaling used for the vertical axis to all axes. This only applies when rendering the final volume and does not affect the calculation of lighting. However, the images seen from side angles do not accurately represent the actual transparency of the clouds in reality as opposed to the ones rendered from the top.

5.2.3. Comparison with Images Rendered with MFASIS

In direct comparison of fig. 5.7a with fig. 5.8a, which shows an image rendered from the same input data, only at a lower resolution, using the implementation of another simulation of light transfer for clouds, called MFASIS (Method for Fast Satellite Image Simulation), which has been integrated in Met.3D by Diefenbach [Die17], it is apparent that the clouds rendered with our method appear overall much more opaque. One

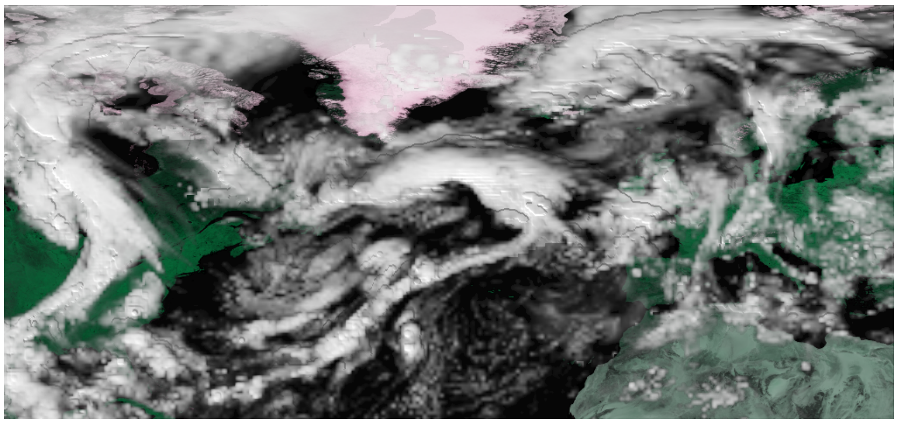


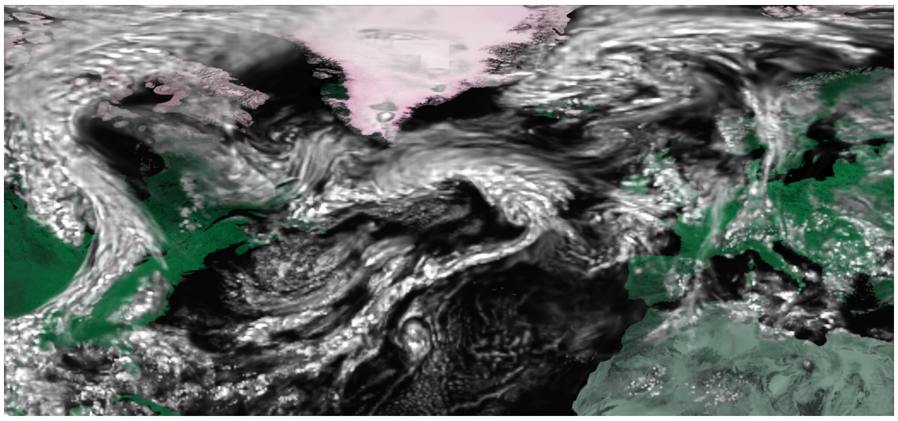
(a) Top view using orthographic projection



(b) Side view

Figure 5.7.: Images rendered with the photon mapping method using realistic scaling of length units.


(a) Image rendered using MFASIS


(b) Image rendered using photon mapping and compensated transparency

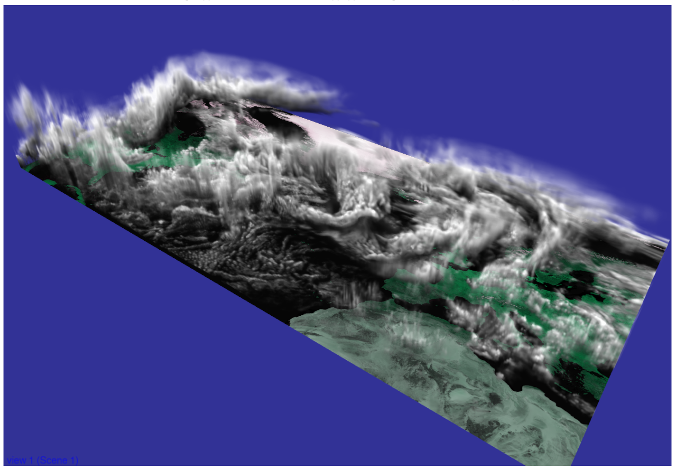

(c) Side view rendered using photon mapping

Figure 5.8.: Comparison of lighting from our photon method with MFASIS.

aspect that MFASIS takes into account, but our method does do not, is light coming from the ground and being scattered inside the clouds, but almost directly forward and thus being visible in the form of higher transparency of the cloud volume.

As a compensation, we will set the space scale for only the final volume rendering, not the lighting calculation, to 0.1, thus lowering all values for optical thickness by a factor of 10 and making the clouds overall more transparent. While this is only an arbitrarily selected value to match the transparency, which removes physical accuracy from our calculations, the result in fig. 5.8b can at least be used to compare the visual outcome of the lighting method.

One major advantage of our method is, however, that it is possible to take the lighting data calculated using realistic scaling in all axes and view the volume from arbitrary directions by enabling "force uniform scale for view", as it has been done for fig. 5.8c, instead of being limited to the orthographic projection from the top.

5.2.4. Performance and Effects of Resolution and Photon Count

We will now analyze the time it takes to generate the photon map using different combinations of photon map resolution and the number of photons as well as the resulting images. All cases use the same high-resolution data set as fig. 5.7a with the following settings:

account for earth's curvature	On
use real vertical scale	On
space scale	1.0
force uniform scale for view	On
effective radius lwc	10 µm
effective radius iwc	25 µm
intensity	13.0
ambient	0.0
max extinction coefficient (for Woodcock tracking)	0.01
Henyey-Greenstein g	0.3
light direction (x, y, z)	(0, 0, -1)

All resulting images can be seen in appendix A.2. The following durations for generating the photon map have been measured:

photon count	16x8x8	32x16x16	64x32x32	128x64x64
100	0.086 s	0.085 s	0.085 s	0.086 s
1000	0.127 s	0.126 s	0.127 s	0.126 s
10000	0.692 s	0.690 s	0.694 s	0.692 s
100000	4.383 s	4.382 s	4.386 s	4.393 s
1000000	42.266 s	42.409 s	42.601 s	42.718 s
10000000	425.378 s	426.363 s	426.839 s	427.696 s

The values above only represent the time for the actual photon tracing step without the conversion to the floating point buffer. The time for this conversion ranges between 0.1 ms and 2 ms.

The result shows only a neglectable impact of the resolution on performance in the measured range. Conversely, the resolution has a major impact on the resulting images in combination with the photon count. Logically, the higher the resolution for a fixed photon count, the less photons will fall into a single cell and thus noise becomes more visible.

A problem can be observed in fig. A.10, fig. A.11 as well as fig. A.16. These are all cases with a low resolution but high photon count. Because an excessive amount of photons falls into each cell, overflows occur in the integer 3D textures when generating the photon map, eventually leading to incorrect results. However, because with a lower photon count, noise is already barely visible, thus invalidating the need for more photons, we can simply accept the errors for these cases since they are not practically useful.

6. Conclusion and Future Work

We have implemented two different volume lighting methods in Met.3D and compared the results. The simple method provides an efficient way to render volumes with lighting composed of simulated single scattering and visual imitations of several phenomena normally caused by multiple scattering, but sacrificing physical accuracy. The second method overcomes this limitation by employing photon mapping. Imposing several restrictions to the photon mapping algorithm enables it to be implemented using only regular 3D textures as temporary buffers, thus making it suitable to be integrated in Met.3D's OpenGL-based environment.

While our performance measurements have shown that the time the simple method takes to generate its optical thickness buffer usually lies below one second for settings leading to an acceptable result, which is appropriate for a pre-pass in an interactive environment, the time for generating a complete photon map can range much higher, depending on the desired resolution and photon count needed to produce an acceptable result with this resolution.

Elek et al. [Ele+12] present two aspects in their work which have not been touched in thesis. First, they initially use a lower-resolution photon map, which would also reduce the needed photon count, and then use the original cloud data as guidance to upsample the photon map to a higher resolution. Second, they do not generate the whole photon map in only a single pass, but split up the photons across several maps in a circular buffer that, combined, form the complete photon map. In every rendered frame, only one of these partial buffers is re-generated, leading to the total work being distributed across multiple frames. This is especially useful in scenarios where the clouds slowly change over time and the photon map has to be updated accordingly. Because data visualized in Met.3D is commonly not animated, a similar approach, which would be suited for our case, is to also distribute the work across several rendered frames, but using the single, complete photon map as it is implemented currently, leading to a continuously improving visual outcome that the user can observe.

An additional possible performance optimization would be to employ empty space skipping when tracing the photon beams in order to avoid calculations and memory operations that will not contribute to the final result.

A. Additional Rendered Images

A.1. Simple

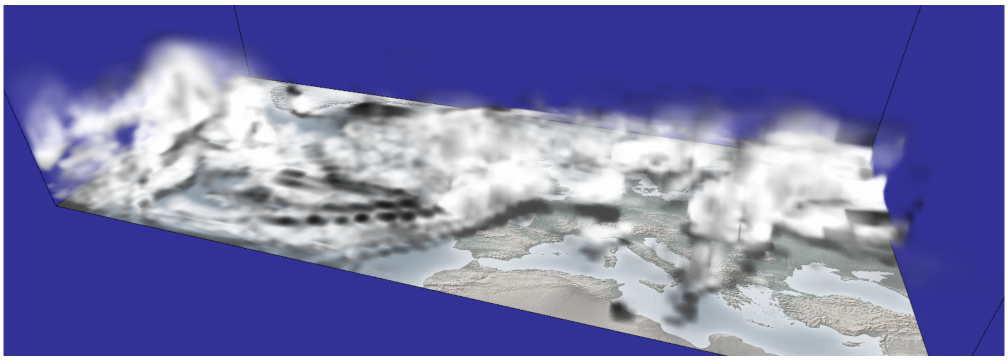


Figure A.1.: Resolution: 32x16x16

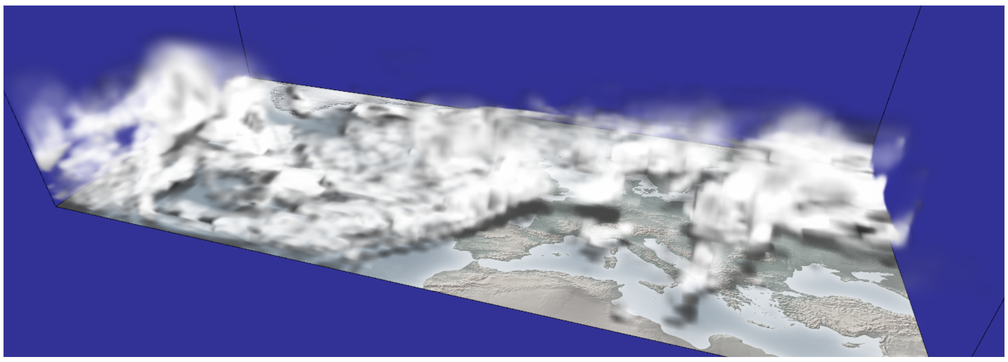


Figure A.2.: Resolution: 64x32x32

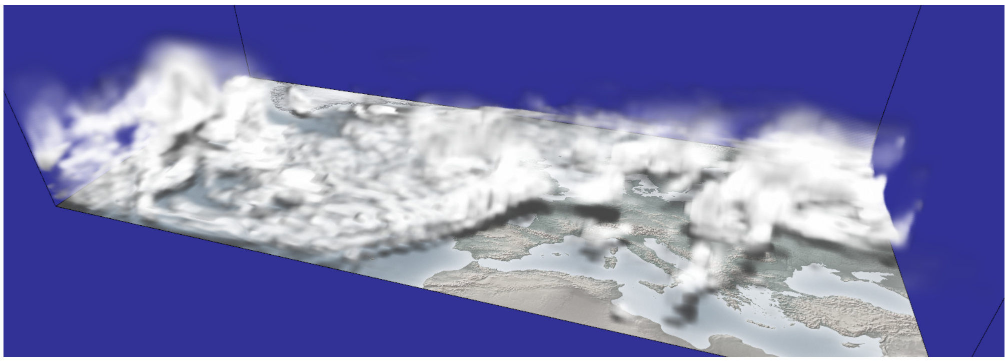


Figure A.3.: Resolution: 128x64x64

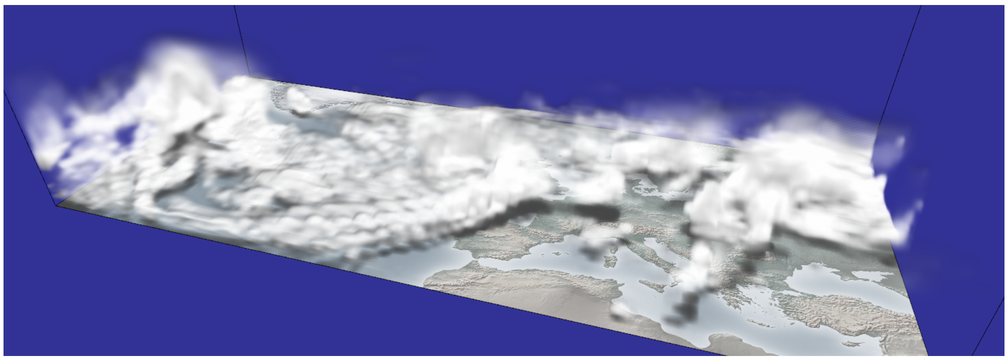


Figure A.4.: Resolution: 512x256x256

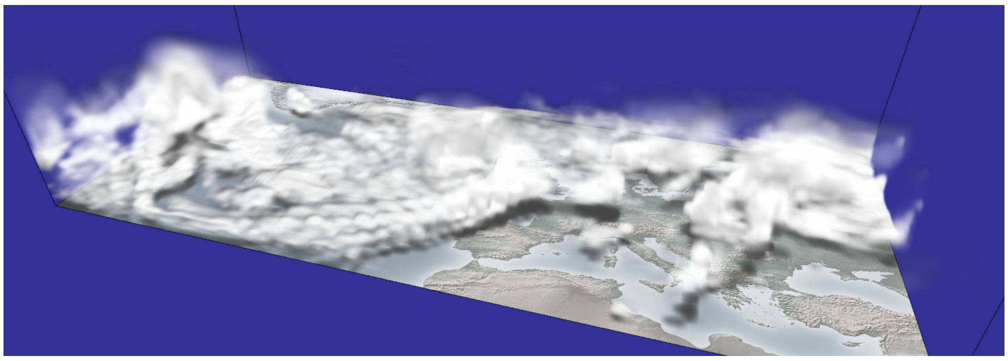


Figure A.5.: Resolution: 1024x512x512

A.2. Photon Mapping

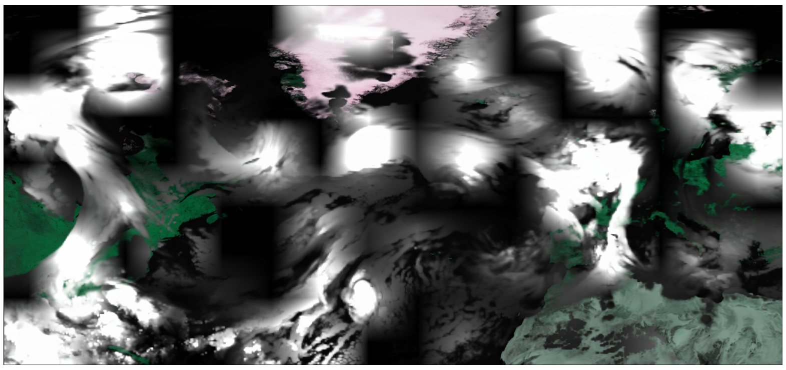


Figure A.6.: Resolution: 16x8x8; Photon Count: 100

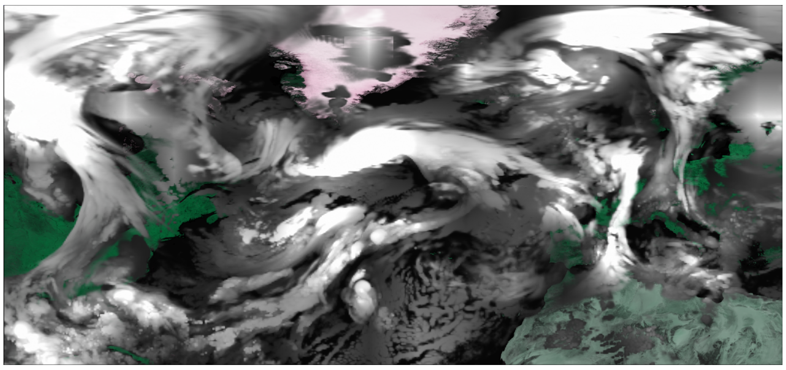


Figure A.7.: Resolution: 16x8x8; Photon Count: 1000

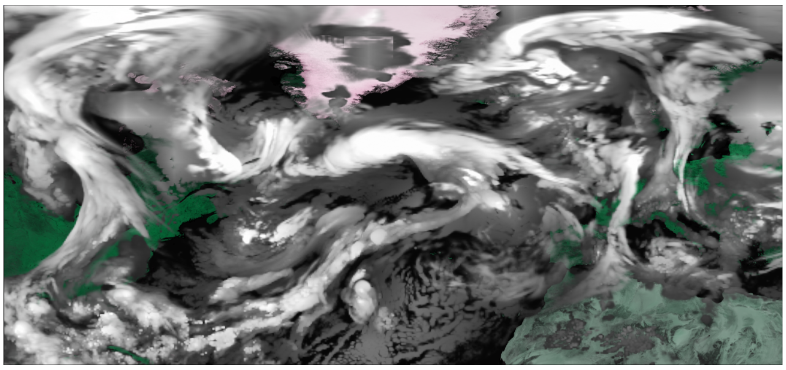


Figure A.8.: Resolution: 16x8x8; Photon Count: 10000

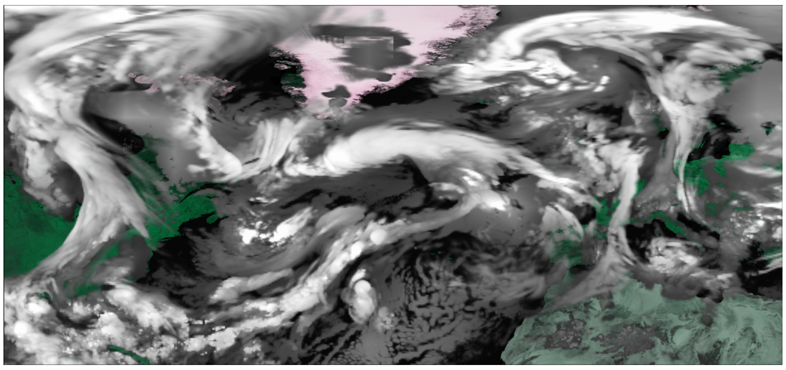


Figure A.9.: Resolution: 16x8x8; Photon Count: 100000

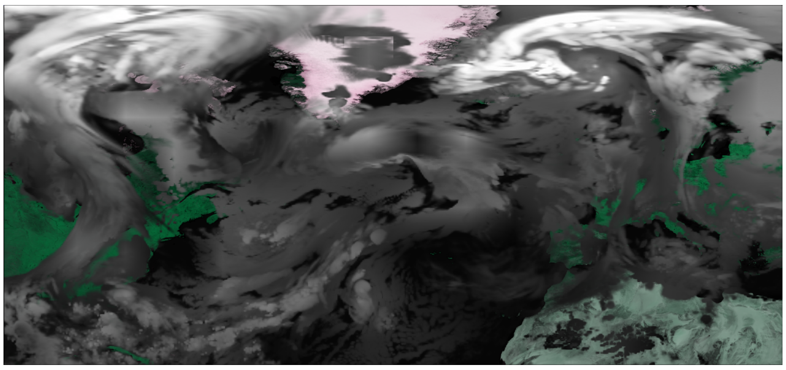


Figure A.10.: Resolution: 16x8x8; Photon Count: 1000000 (showing an integer overflow in the photon map)

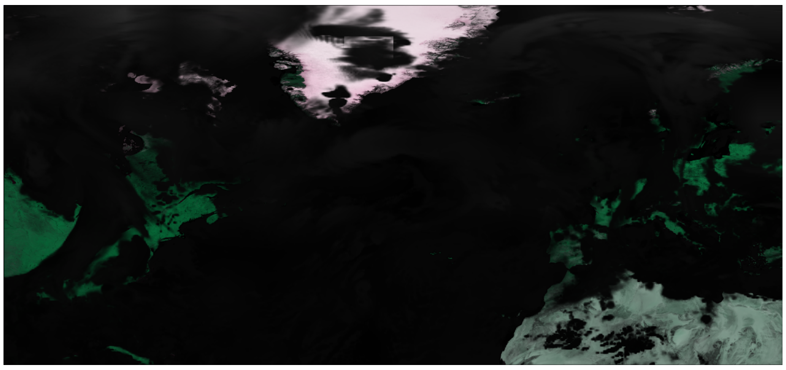


Figure A.11.: Resolution: 16x8x8; Photon Count: 10000000 (showing an integer overflow in the photon map)

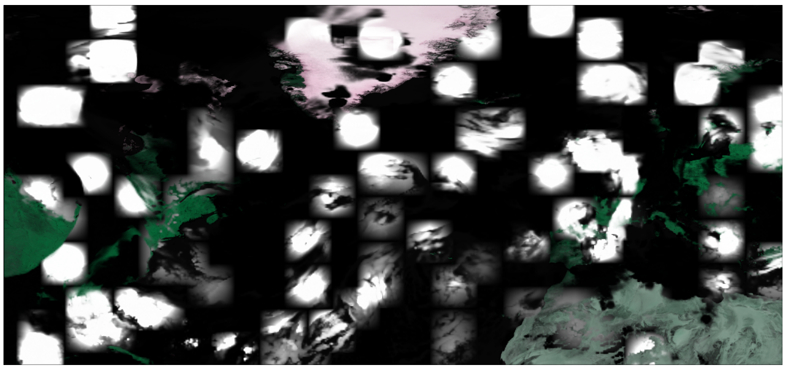


Figure A.12.: Resolution: 32x16x16; Photon Count: 100

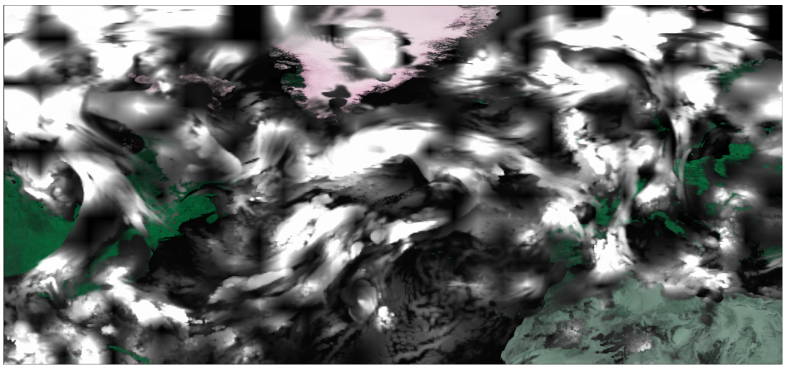


Figure A.13.: Resolution: 32x16x16; Photon Count: 1000

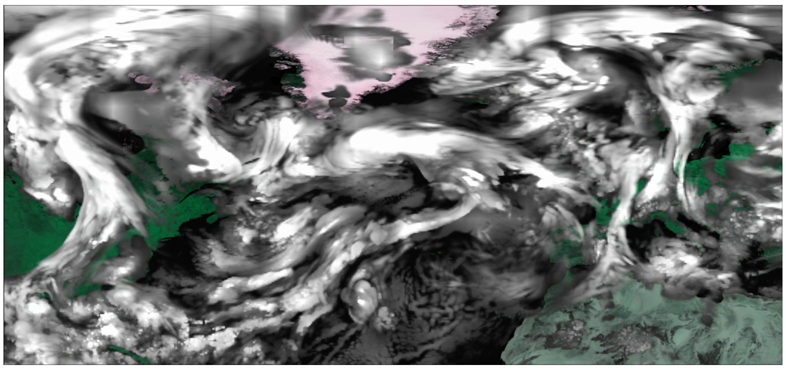


Figure A.14.: Resolution: 32x16x16; Photon Count: 10000

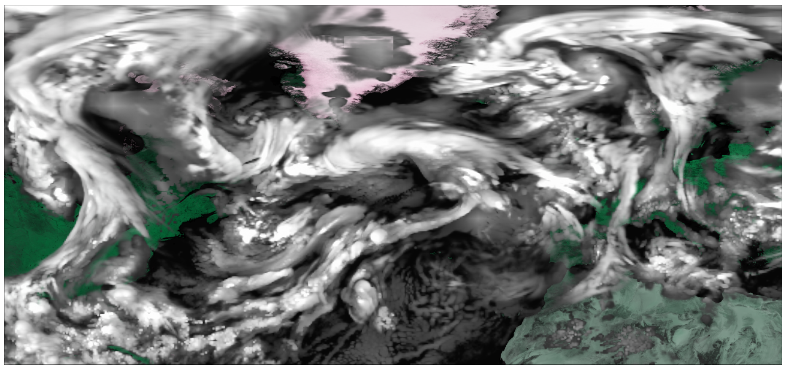


Figure A.15.: Resolution: 32x16x16; Photon Count: 100000

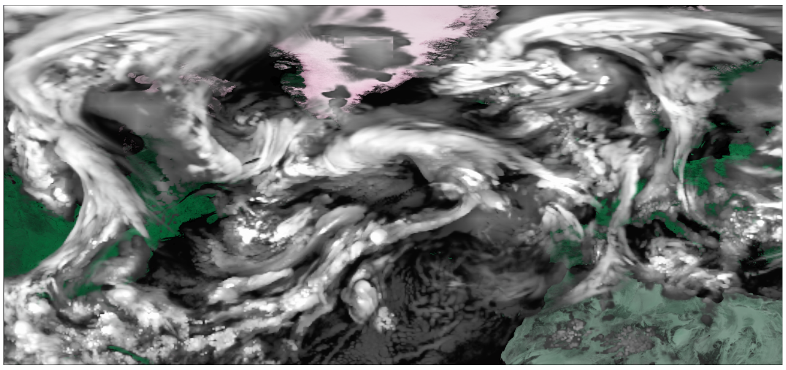


Figure A.16.: Resolution: 32x16x16; Photon Count: 1000000

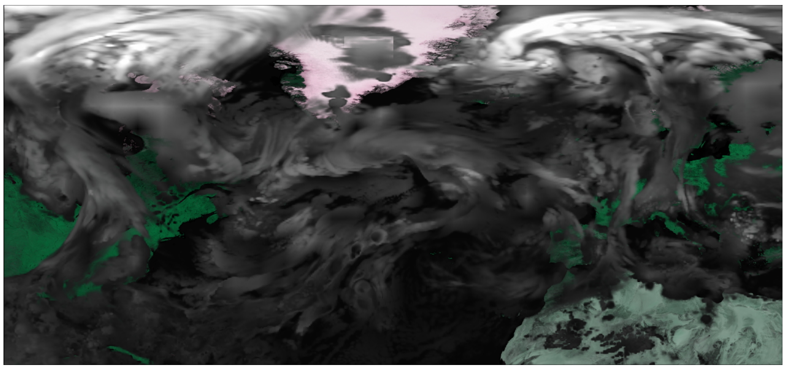


Figure A.17.: Resolution: 32x16x16; Photon Count: 10000000 (showing an integer overflow in the photon map)

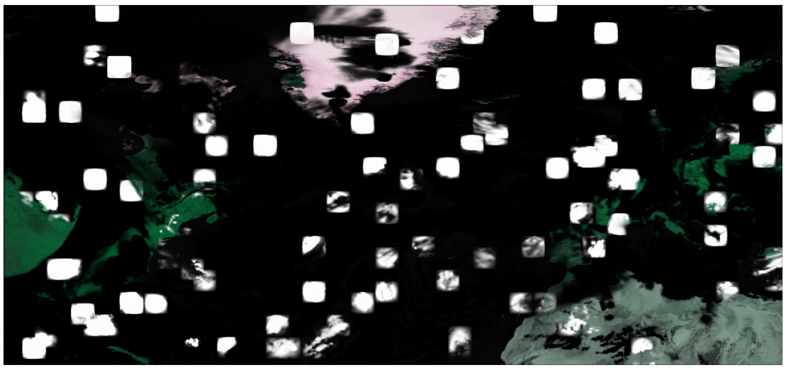


Figure A.18.: Resolution: 64x32x32; Photon Count: 100

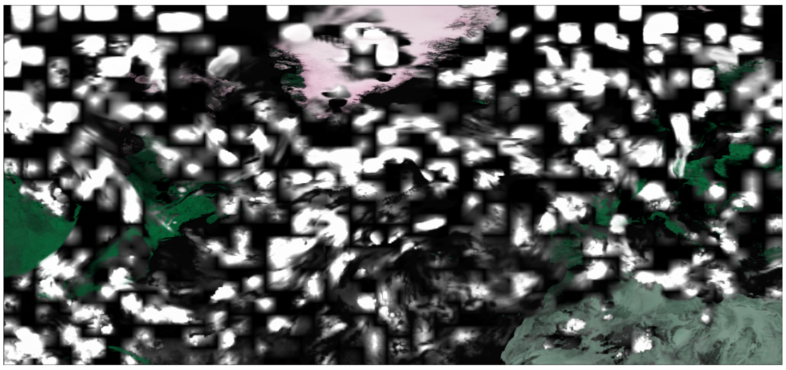


Figure A.19.: Resolution: 64x32x32; Photon Count: 1000

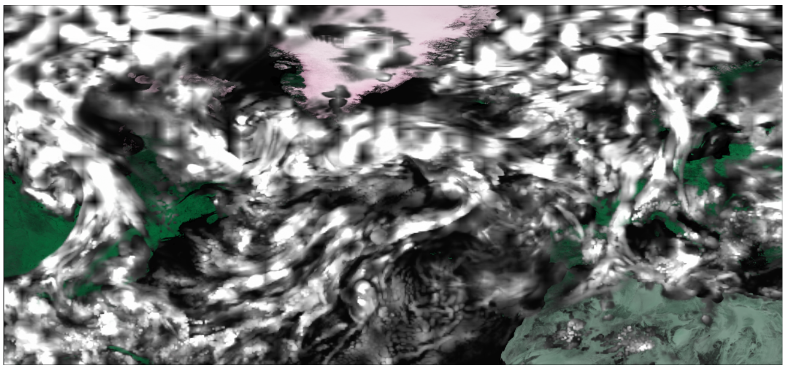


Figure A.20.: Resolution: 64x32x32; Photon Count: 10000

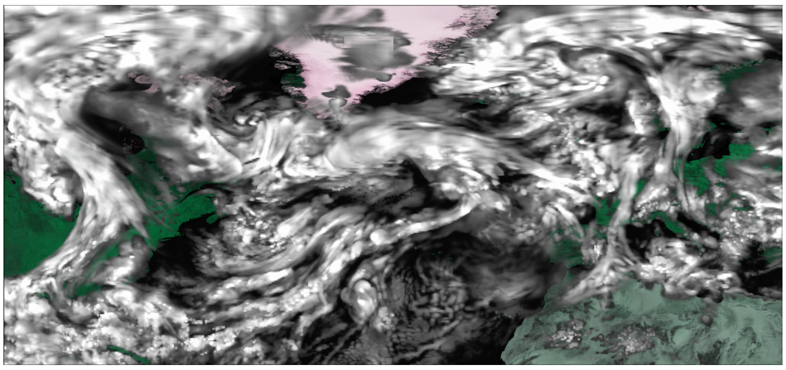


Figure A.21.: Resolution: 64x32x32; Photon Count: 100000

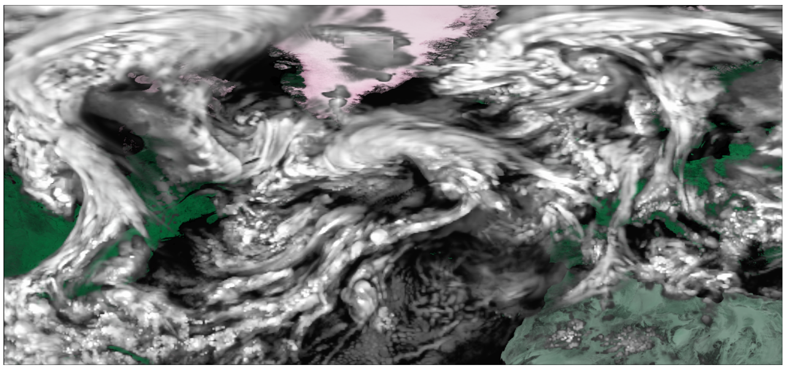


Figure A.22.: Resolution: 64x32x32; Photon Count: 1000000

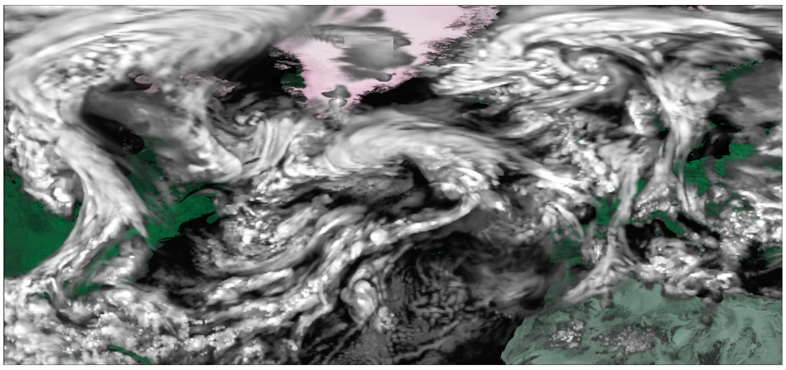


Figure A.23.: Resolution: 64x32x32; Photon Count: 10000000

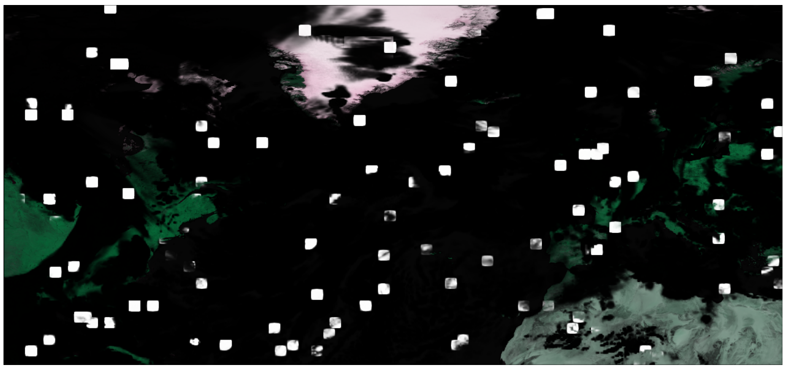


Figure A.24.: Resolution: 128x64x64; Photon Count: 100

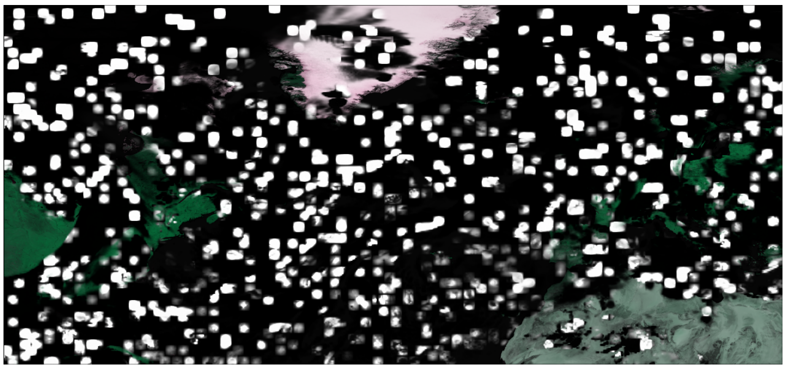


Figure A.25.: Resolution: 128x64x64; Photon Count: 1000

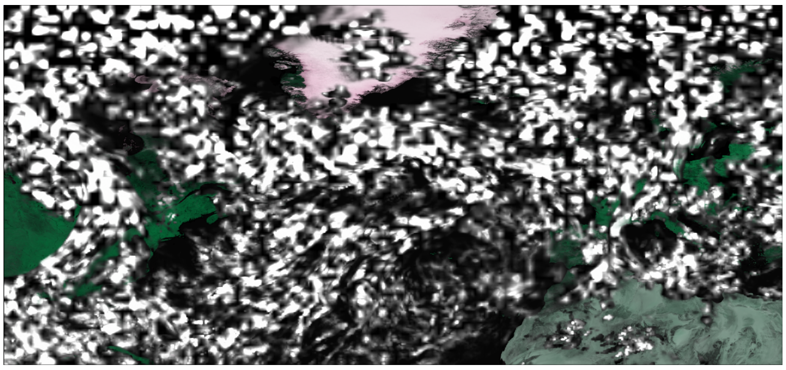


Figure A.26.: Resolution: 128x64x64; Photon Count: 10000

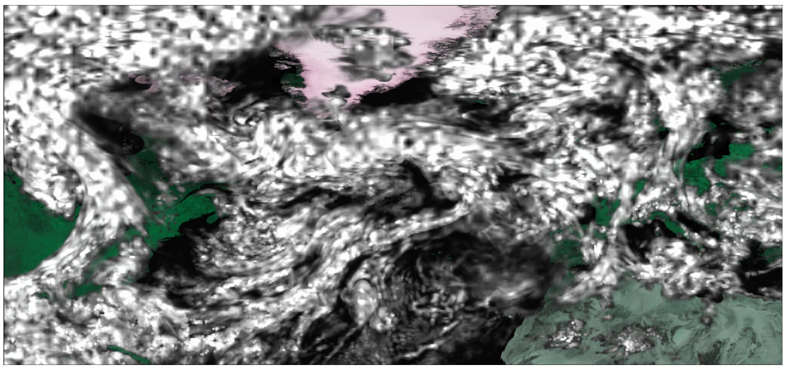


Figure A.27.: Resolution: 128x64x64; Photon Count: 100000

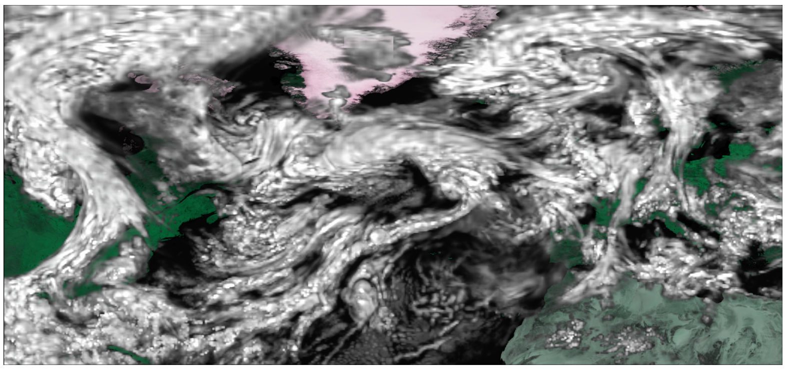


Figure A.28.: Resolution: 128x64x64; Photon Count: 1000000

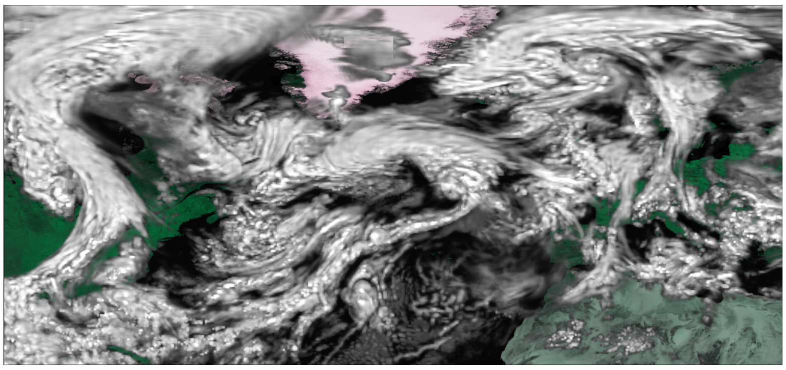


Figure A.29.: Resolution: 128x64x64; Photon Count: 10000000

List of Figures

Bibliography

[Die17] T. A. Diefenbach. "GPU-based Visualization of Simulated Clouds in an Interactive Three Dimensional Framework." MA thesis. Ludwig-Maximilians-Universität München, Aug. 2017.

[Ele+12] O. Elek, T. Ritschel, A. Wilkie, and H.-P. Seidel. "Interactive cloud rendering using temporally coherent photon mapping." In: *Computers & Graphics* 36.8 (2012), pp. 1109–1118.

[Fu96] Q. Fu. "An accurate parameterization of the solar radiative properties of cirrus clouds for climate models." In: *Journal of Climate* 9.9 (1996), pp. 2058–2082.

[Har] J. P. Harrington. *The Henyey-Greenstein phase function*. URL: `https://www.astro.umd.edu/~jph/HG_note.pdf`.

[HS93] Y. Hu and K. Stamnes. "An accurate parameterization of the radiative properties of water clouds suitable for use in climate models." In: *Journal of climate* 6.4 (1993), pp. 728–742.

[HT07] L. Howes and D. Thomas. "Efficient random number generation and application using CUDA." In: *GPU gems* 3 (2007), pp. 805–830. URL: `https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch37.html`.

[Jar+11] W. Jarosz, D. Nowrouzezahrai, I. Sadeghi, and H. W. Jensen. "A comprehensive theory of volumetric radiance estimation using photon points and beams." In: *ACM Transactions on Graphics (TOG)* 30.1 (2011), p. 5.

[JC98] H. W. Jensen and P. H. Christensen. "Efficient simulation of light transport in scenes with participating media using photon maps." In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*. ACM. 1998, pp. 311–320.

[M] *Met.3D - Interactive 3D visualization of meteorological (ensemble) simulations*. URL: `https://met3d.wavestoweather.de`.

[MMC12] S. Mohanty, A. K. Mohanty, and F. Carminati. "Efficient pseudo-random number generation for monte-carlo simulations using graphic processors." In: *Journal of Physics: Conference Series* 368.1 (2012), p. 012024.

[Rau+15] M. Rautenhaus, M. Kern, A. Schäfler, and R. Westermann. "Three-dimensional visualization of ensemble weather forecasts – Part 1: The visualization tool Met.3D (version 1.0)." In: *Geoscientific Model Development* 8.7 (2015), pp. 2329–2353. DOI: `10.5194/gmd-8-2329-2015`. URL: `https://www.geosci-model-dev.net/8/2329/2015/`.

[RD] *RenderDoc*. URL: `https://renderdoc.org`.

[Rod+18] M. J. Rodwell, D. S. Richardson, D. B. Parsons, and H. Wernli. "Flow-dependent reliability: A path to more skillful ensemble forecasts." In: *Bulletin of the American Meteorological Society* (2018). DOI: `10.1175/BAMS-D-17-0027.1`.

[SV15] A. Schneider and N. Vos. "The Real-Time Volumetric Cloudscapes of Horizon Zero Dawn." In: SIGGRAPH '15. Aug. 2015.

[Woo+65] E. Woodcock, T. Murphy, P. Hemmings, and S. Longworth. "Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry." In: *Proc. Conf. Applications of Computing Methods to Reactor Problems*. Vol. 557. 2. 1965.