Radiative Transfer in Reflection Nebulae

Edvald Ingi Gislason

Kongens Lyngby 2010

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

Being able to perform large-scale simulations on complex models is among the most important things in many fields of science, ranging from finance to astrophysics. Such simulations are of particular interest to astrophysicists as the majority of their studies resides in outer space, far from our reaches. Due to the remoteness of these objects, they can only be observed through data coming from instruments, like images produced by telescopes. Despite this, most of the physics and mathematical theory describing the origin and the composition of these objects are in fact, very mature, after decades of research.

One type of these distant interstellar objects are called reflection nebulae. They are dense regions of dust in space which reflect light coming from stars within them. With development of new algorithms and hardware, improvements can be made on existing methods for added complexity in lower computation times.

In this thesis, I present a method for simulating radiative transfer in reflection nebulae using volume photon mapping in CUDA and turning the photon map into a light field by convolving it with a filter using 3D FFT on the GPU. The light field is then used with my implementation of a GPU-based ray marching algorithm to give real-time visualizations of the radiative transfer in reflection nebulae.

Figure 1: The Orion nebula is a reflection nebula [NASA 2000]

Preface

This thesis was prepared at Informatics Mathematical Modeling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering.

The thesis deals with different aspects of mathematical modeling and real-time visualization of radiative transfer in reflection nebulae (heterogeneous volumes) using stochastic photon tracing with partial knowledge about noise, Fourier Transform and reflection nebulae environments.

Lyngby, September 2010

Edvald Ingi Gislason

Acknowledgments

I would like to thank my supervisor, assistant professor Jeppe Revall Frisvad, for his guidance and support and for accepting my thesis proposal. His unique skills and deep understanding of the theory involved proved vital to the success of this project.

I would also like to express my gratitude to Dr. Anja C. Andersen at the Dark Cosmology Center (Niels Bohr Institute, University of Copenhagen) and Allan Hornstrup at DTU Space for providing their support and consultancy in astrophysics.

Special thanks go to my fiancée Hulda Júlíana for her support and patience while writing a thesis of her own, as well as giving birth to our son, Jón Gísli on July 7 during the same time. Thanks also go to my parents Anna and Gísli whose parenting skills and guidance ultimately led to the writing of these very words.

I would also like to extend my gratitude to my supervisors and co-workers of the dpt. of Financial Products at Nykredit, for their support, patience and for granting me flexible work hours along with my studies.

Contents

CHAPTER 1

Introduction

To explain the appearance of many objects, ranging from the clouds we see in the sky to nebulae in the distant universe, we need powerful simulation frameworks that can accurately compute the mathematical and physical models that describe the appearance of these objects. These frameworks enable researchers to see accurate renderings of the results of new or existing models and allow the testing of different parameters into these models to compare with the actual real objects being modeled. They are an essential tool for any researcher if the goal is to gain a deeper understanding of e.g. the complex interactions of light and dust in interstellar space [Magnor et al. 2005]

Accurate physics-based simulations not only have scientific purposes but also in terms of entertainment. Game- and film-makers often try to make visualizations and special effects with the aim of maximizing realistic appearance of objects and natural phenomena. Artistic representations of these objects are constantly being replaced by very realistic renderings made by sophisticated physics-based appearance models.

In this thesis I present a method for simulating radiative transfer in reflection nebulae using volume photon mapping in CUDA and turning the photon map into a light field by convolving it with a filter using 3D FFT on the GPU. The light field is then used with my implementation of a GPU-based ray marching algorithm to give real-time visualizations of the radiative transfer in reflection nebulae. A complete application was made for handling input and output data from the models through a graphical user interface, along with visualization of the results in real-time on high frame-rates.

This is a significantly different approach to existing methods and extends previous work made in the field. While radiative transfer theory remains the same, the algorithms and tools used in accomplishing this task are state of the art, following the rapid development of algorithms and hardware in recent years.

1.1 Related work

A few publications exist that address scientific visualization of reflection nebulae. In 2005, Magnor et. al. presented an approach to model and visualize, in real-time, reflection nebulae in 3D [Magnor et al. 2005]. Their approach was based on the same physical models that are used in astrophysics research, to accurately calculate light scattering in procedurally generated dust distributions surrounding one or more stars. They covered all the aspects of anisotropic scattering, wavelength dependence, multiple scattering and provided real-time visualization of the results. Some of their results are shown in figure 1.1

In 2007, Lintu et.al. presented an approach to 3D reconstruction of reflection nebulae from a single image[Lintu et al. 2007]. Instead of modeling dust densities with noise functions, they reconstructed these densities from an image of an existing reflection nebula to give an approximation of the structure of the actual nebula. They then modeled the light transport to give a final result.

Many publications address algorithms used for solving these models. Dr. Henrik Wann Jensen developed and published a book covering realistic image synthesis using photon mapping[Jensen 2001]. This method traces photons from light-sources to accurately simulate natural lighting in complex 3D environments. Photon mapping has been used to accurately simulate caustics and sub-surface scattering of light to model e.g. physically realistic looking skin as seen in motion pictures like "Lord of the Rings" and "Avatar". Some examples of Dr. Jensen's work are shown in figure 1.2

In 1938, L.G. Henyey and Jesse L. Greenstein presented a theory describing the radiative transfer through nebulae to interpret observations of the colors of

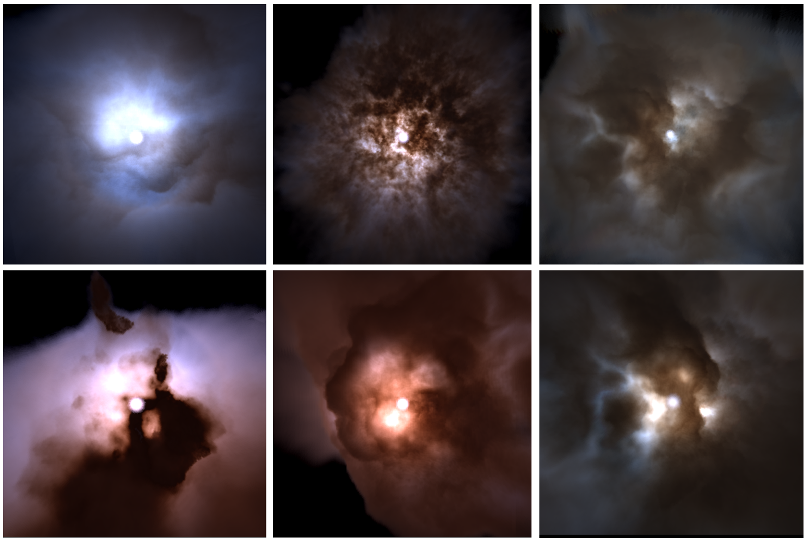

Figure 1.1: Example results from *Reflection Nebula Visualization* [Magnor et al. 2005]

Figure 1.2: Photon mapping used to simulate caustics (left), general global illumination (center) and volumetric caustics (right) [Jensen 2000]. Note that these are all purely computer generated images (CGI).

reflection nebulae[Henyey and Greenstein 1938]. Their work yielded the well known Henyey-Greenstein phase function which they presented in a paper two years later [Henyey and Greenstein 1940]. The Henyey-Greenstein phase function is generally used in astrophysics ([Gordon 2004], [Andersen 2007]) and combined with the photon mapping thoery, is a fundamental part of the methods presented in this thesis.

1.2 Parallel computing

Algorithms for calculating the radiative transfer rely heavily on Monte Carlo simulations and are therefore mostly *embarrassingly parallel*[Foster 1995]. Embarrassingly parallel algorithms can easily seperate their workload into a number of parallel tasks as there is little or no dependency between those tasks. Monte Carlo methods rely on repeated random sampling to compute their results to gradually converge to a solution. They converge much more quickly than numerical methods, require less memory and are easier to program. Their success and popularity have recently grown fast with the rapidly growing GPU industry which provides highly affordable hardware for use in general purpose computations.

To utilize the power of the GPUs a parallel-programming framework is needed.

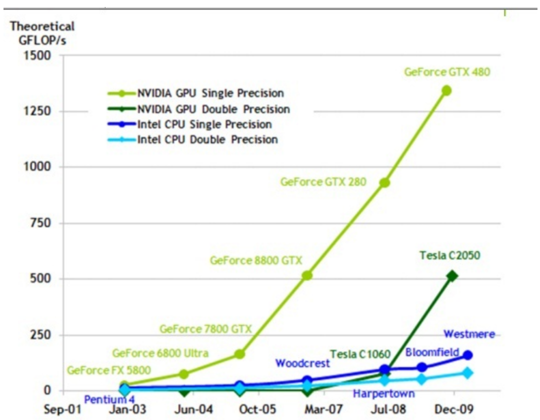

Figure 1.3: Evolution of theoretical GFLOP/s of CPUs and GPUs.

One such framework is CUDA, it stands for Compute Unified Device Architecture and was developed by the NVIDIA corporation for the use of graphics processing units (GPUs) for general purpose computing (GPGPU). GPUs can be thought of as a collection of small processors and can range anywhere from

8 to over 512 processors (or cores) but these numbers grow fast by every year. The computational power of processors or systems of processors is measured in millions of floating point operations per second or *GFLOP/s*. As the number of cores on GPUs increases their theoretical GFLOP/s limits increase and since the increase of cores per GPU is far greater than that of CPUs, the GPUs have become far superior as shown in figure 1.3. This only applies to parallelizeable algorithms as the high number of cores used contribute greatly to how many GFLOP/s are performed.

The gaming industry has financed the rapid growth and mass production of these GPUs for more than a decade. This doesn't come as a surprise considering the video-game industry has surpassed both the music- and movie-industry in revenues and growth [Ars Technica 2008] and video-game producers are driving this development by constantly competing in pushing the extreme limits of current GPUs to create the next blockbuster.
The gamers of the world are literally paying for the development of affordable desktop-supercomputers[Computerworld 2008].

Chapter 2

Theory

2.1 Reflection nebulae

Reflection nebulae consist of either high-density or diffuse dust, usually illuminated by a single or small number of nearby stars[Gordon 2004]. The structure of these dusty regions is revealed as light scatters and gets absorbed by the dust particles. The stars inside these regions are usually of low mass and do not have enough energy to ionize the gas particles in their dusty neighborhood, but enough so that the reflected light can be observed (see figure 1 on page ii) [Lintu et al. 2007].

The color, i.e. the visible-light part of the electromagnetic spectrum, is determined by the type of the central star(s) and the scattering properties of interstellar dust particles. Light scatters differently at different wavelengths. This is the reason why reflection nebulae tend to be more blue as light at blue wavelengths scatters much more than light at the red end of the visible spectrum.
Interstellar dust is mainly composed of carbons and silicates and stems predominantly from so called AGB[1] stars. The particles vary in sizes ranging between

[1]Asymptotic Giant Branch

100nm and 1μm[Magnor et al. 2005]. The scattering properties of single dust particles are well described by the Lorenz-Mie theory which is a complete analytical solution of Maxwell's equations for the scattering of electromagnetic radiation by spherical particles [Bohren and Huffman 1983]. The scattering model of interstellar dust uses two parameters. One is the scattering *albedo(α)* and the other is the angular scattering distribution of dust, i.e. a scattering phase function $\Phi(\theta)$.

The albedo $a \in [0,1]$ determines the probability of a scattering event to occur or the average ratio of radiation incident on the dust particle that is being scattered. If all incident radiation is absorbed, $a = 0$. On the other hand, if all incident radiation is scattered, $a = 1$, in other words, the medium is highly scattering[Magnor et al. 2005]. From the average absorption coefficient σ_{abs} and scattering coefficient σ_{sct}, combined into an extinction coefficient σ_{ext}, a is defined

$$a = \frac{\sigma_{sct}}{\sigma_{abs} + \sigma_{sct}} = \frac{\sigma_{sct}}{\sigma_{ext}} \tag{2.1}$$

The angular scattering distribution, or the scattering anisotropy, is modeled using the Henyey-Greenstein phase function:

$$\Phi(\theta, g) = \frac{1 - g^2}{(1 + g^2 - 2g\cos\theta)^{3/2}} \tag{2.2}$$

This Henyey-Greenstein phase function is a good approximation for dust grains, except possibly in the far-ultraviolet [Gordon 2004]. At visible wavelengths, the scattering albedo(a) and the scattering anisotropy factor (g) are both approximately 0.6 [Magnor et al. 2005; Gordon 2004].

2.2 Phase functions

A phase function, defined $p(\mathbf{x}, \vec{\omega}', \vec{\omega})$ or $p(\mathbf{x}, \vec{\omega}' \to \vec{\omega})$ describes the angular distribution of scattered radiation at a point. Phase functions have been developed to model e.g. Rayleigh scattering and Mie scattering. The Rayleigh model can be used to accurately model scattering from particles that are smaller than the wavelength of light. An example of such particles are the molecules in our planet's atmosphere. In other words, the Rayleigh scattering can explain why the sky is blue and the sunset red[Pharr and Humphreys 2004]. Mie scattering is based on a more general theory, derived from Maxwell's equations and can describe scattering from wider range of particles sizes, e.g. water droplets and fog. Again, to put that into a more general context, it can explain how rainbows work.

The Henyey-Greenstein phase function is widely used in computer graphics and other fields and was developed by L. Henyey J. Greenstein to explain the scattering by interstellar-dust. It has a single parameter, g, which is referred to as the asymmetry parameter of the phase function and ranges from -1 for complete back scattering to 0 for isotropic scattering to 1 for complete forward scattering. Given an arbitrary phase function, the anisotropy parameter g or the average cosine of scattered directions, can be computed as

$$g = \int_{\Omega_{4\pi}} p(\mathbf{x}, \vec{\omega}' \to \vec{\omega}) \cos\theta d\vec{\omega}' \tag{2.3}$$

The phase functions are very useful in stochastic ray tracing since they can easily be importance sampled, which is an essential part of modeling radiative transfer in interstellar dust. Figure 2.1 shows randomly sampled directions of two different values of g:

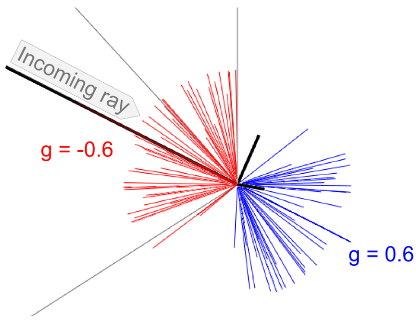

Figure 2.1: Random directions sampled with the Henyey-Greenstein phase function with two different asymmetry parameters. If g was set to 0 it would yield directions randomly sampled in all directions (isotropic).

2.3 Radiative transfer equation

The RTE[2] itself represents the change of radiance of photons as they interact with particles in participating media, demonstrated in figure 2.2

[2]Radiative Transfer Equation

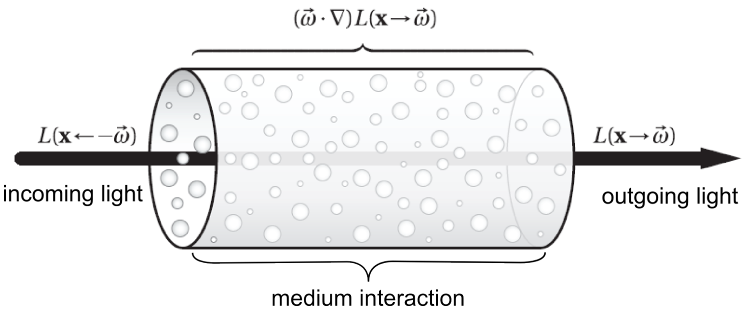

Figure 2.2: The RTE represents the change of radiance as the photons interact with particles in a medium they travel through [Gutierrez et al. 2009]

In this thesis, the radiative transfer equation is used to calculate the radiance received at each voxel in a discretized volume. It captures any event that affects radiance, namely emission, in-scattering, out-scattering and absorption and is defined as follows:

$$\begin{aligned}(\vec{\omega} \cdot \nabla) L(\mathbf{x} \rightarrow \vec{\omega}) = \; & \overbrace{-\sigma_t(\mathbf{x}) L(\mathbf{x}, \vec{\omega})}^{Extinction} \\ & + \overbrace{\sigma_s(\mathbf{x}) \int_{\Omega_{4\pi}} p(\mathbf{x}, \vec{\omega}', \vec{\omega}) L(\mathbf{x}, \vec{\omega}) d\omega'}^{In-scattering} \\ & + \underbrace{L_e(\mathbf{x}, \vec{\omega})}_{Emission}\end{aligned} \tag{2.4}$$

To break the equation down into components; the change in radiance L in the direction $\vec{\omega}$ due to out-scattering is given by:

$$(\vec{\omega} \cdot \nabla) L(\mathbf{x} \rightarrow \vec{\omega}) = -\sigma_t(\mathbf{x}) L(\mathbf{x}, \vec{\omega}) \tag{2.5}$$

and change due to absorption is:

$$(\vec{\omega} \cdot \nabla) L(\mathbf{x} \rightarrow \vec{\omega}) = -\sigma_a(\mathbf{x}) L(\mathbf{x}, \vec{\omega}) \tag{2.6}$$

Together, out-scattering and absorption contribute to the loss of radiance and combined they form the extinction coefficient:

$$\overbrace{\sigma_t(\mathbf{x})}^{extinction} = \overbrace{\sigma_a(\mathbf{x})}^{absorption} + \overbrace{\sigma_s(\mathbf{x})}^{out-scattering} \tag{2.7}$$

As a ray travels through a volume it can also accumulate radiance due to in-scattering of light, which is given by:

$$(\vec{\omega} \cdot \nabla)L(\mathbf{x} \to \vec{\omega}) = \sigma_s(\mathbf{x}) \int_{\Omega_{4\pi}} p(\mathbf{x}, \vec{\omega}', \vec{\omega})L(\mathbf{x}, \vec{\omega})d\omega' \tag{2.8}$$

where the incident radiance is integrated over all directions on the sphere $\Omega_{4\pi}$ and the phase function, explained in the previous section, is used for describing the distribution of the scattered light. There can also be a gain in radiance due to emission L_e from either the medium itself (e.g. ionized gas) or in the case of the topic here, a *glowing* star. L_e is given by:

$$(\vec{\omega} \cdot \nabla)L(\mathbf{x} \to \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) \tag{2.9}$$

To use the RTE to calculate how radiance is distributed throughout participating media, it needs to be derived into the volume rendering equation by integrating Eq. 2.4 on both sides for a segment of length s [Jensen 2001]. The volume rendering equation is derived from the RTE as:

$$\begin{aligned} L(x, \vec{\omega}) = \; & \int_0^s e^{-\tau(x,x')}\sigma_a(x')L_e(x', \vec{\omega})dx' \\ & + \int_0^s e^{-\tau(x,x')}\sigma_s(x') \int_{\Omega_{4\pi}} p(x', \vec{\omega}', \vec{\omega})L_i(x', \vec{\omega}')d\vec{\omega}'dx' \\ & + e^{-\tau(x,x+s\vec{\omega})}L(x - s\vec{\omega}, \vec{\omega}) \end{aligned} \tag{2.10}$$

where the optical depth $\tau(x, x')$ is given by:

$$\tau(x, x') = \int_x^{x'} \sigma_t(t)dt \tag{2.11}$$

Realistic Image Synthesis Using Photon Mapping by Dr. Henrik Wann Jensen covers the theory behind the radiative transfer equation and it's derivatives in great detail. In the book he presents the Photon mapping method for efficiently solving the them.

2.4 Photon mapping

Photon mapping is a global illumination algorithm. Global illumination algorithms are physically-based simulations of all light-scattering in a synthetic model. The goal of photon mapping, along with other similar algorithms, is to produce an accurate prediction of the intensity of light at any given point in the model. The input into such models can be volume definitions, description of geometry, material properties and light-sources. Calculating how much radiation is received *directly* at a given point is usually a trivial part of those models. Indirect lighting, however, is far more complex as it is the result of multiple scattering. When working with heterogeneous media, the problem grows even more complex as volume density must be sampled along the light's path to account for higher extinction in denser regions, whereas only the total distance traveled through homogeneous media and an extinction constant directly effects the extinction along the path.

The goal of photon mapping is to partly solve or speed up the calculation of the radiative transfer equation (RTE) defined in Eq. 2.4

A photon map is constructed of photons emitted from the light sources and traced through the model. Each photon traced from the source can go through multiple scattering-events as it propagates along its way until it is absorbed. Due to this, the photon map can capture the complex characteristics of light and it's interaction with matter in complex models.

Consider Fig. 3.1 on page 26. It demonstrates how a ray is used to sample a volume at given intervals. Regular algorithms would have to perform expensive stochastic sampling at every step to estimate radiance from in-scattering. The photon map eliminates this problem by tracing photons from the light source into the volume and spatially storing the power of these photons where they *land*. When the ray is traced to sample radiance in the volume, a simple photon-density estimate around each sample is performed to estimate the average total radiance.

2.5 Volume Ray marching

A volume here is defined as a cubic grid of texels or in graphics terminology, a 3D texture. A 3D texture can be explained as an array of 2D images, but instead of having two dimensional pixels like an image, pixels are defined in three dimensions. The term *pixel* is explained in figure 2.3.

Visualization of a volume is possible by using e.g. volume ray marching. The

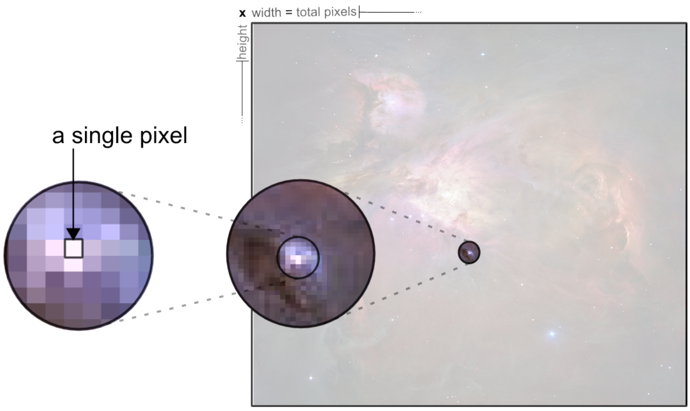

Figure 2.3: An image of dimensions **W**idth and **H**eight has **W** × **H** pixels

task is to project a 3D volume on to a 2D surface/image, preferably at high frame-rates. In volume ray marching, where a ray is traced through a scene for every pixel on the 2D surface to give a value, represented in color(RGB), i.e. a color each pixel is *seeing*. This can of course be done in a custom software ray-tracer, but so called GPU shaders are very well suited for this kind of work. GPU shaders replace the old fixed-function graphics pipeline in commonly used graphics API's[3] and are very popular in modern video-games as they can, among other things, solve complex lighting equations in real-time and are optimized for vector calculus.

Volume ray marching is easily achieved with GPU shaders as they are very specialized in handling texture data and performing 3D and 2D projections, fully utilizing the mathematical power of the GPU.

Fragment shaders, also referred to as pixel shaders, are blocks of code that handle the calculation of the color of individual pixels in the programmable

[3] Application Programming Interface

graphics pipeline. To perform volume ray tracing in a fragment shader, the volume needs to be uploaded to the GPU memory as a 3D texture. The fragment shader can then sample positions inside the volume with unit-vectors. Given an *eye* position, look-at point, the volume boundary(defined as a unit cube) and a 3D texture holding the actual RGBA values for the volume, enough information is available for the shaders to execute the volume ray marching algorithm.

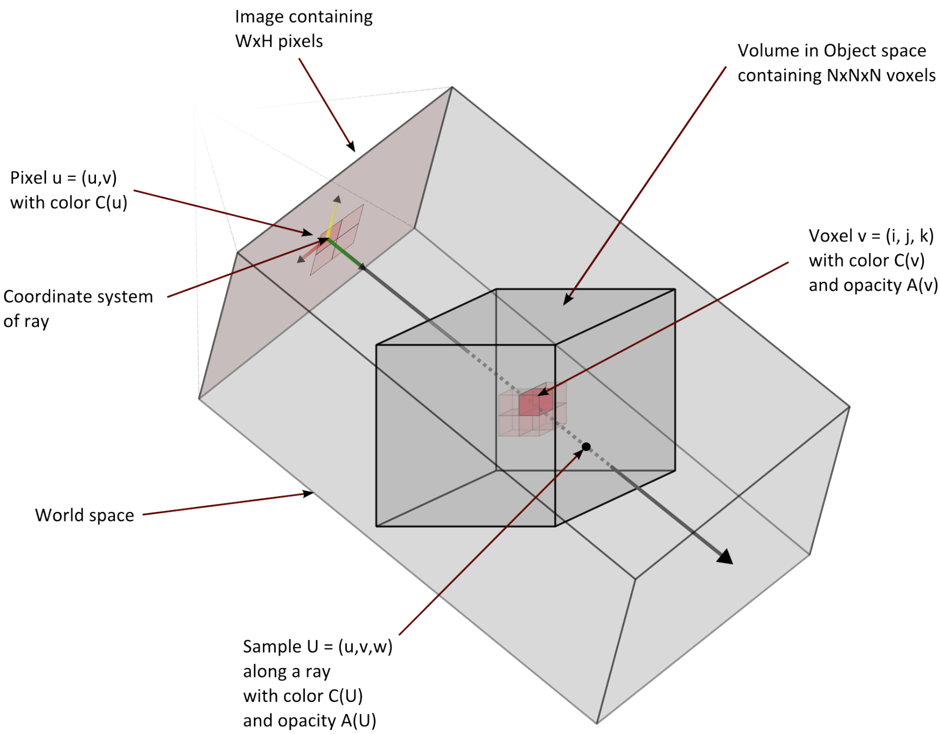

Figure 2.4: Volume Ray marching demonstrated, figure inspired by [Levoy 1990]

In an attempt to demonstrate the process described above, I present a diagram of the volume ray marching process in Fig. 2.4. It shows how a ray is traced from a pixel into the scene, where it hits a volume. To find the points where a ray enters a volume and where it exits, a suitable ray-volume intersection algorithm must be used, see section 2.5.1. Inside the volume, samples are taken in a number of steps and accumulated along the ray until either the ray exits the volume boundary or it has accumulated alpha values summing up to 1. The discretized version of the volume integral function is defined:

$$
\begin{aligned}
&\text{Accumulated color:} && C = \sum_{i=0}^{N} C_i A_i \\
&\text{Accumulated opacity:} && A = \sum_{i=0}^{N} A_i \\
&\text{Final pixel value:} && C_{pixel} = C(A) + C_{bg}(1 - A)
\end{aligned}
\tag{2.12}
$$

Here C_{bg} is a background color which can be any constant or a function, e.g. a texture-lookup from a cube-map. Please refer to Algorithm 4 on page 28 for the specifics on my implementation.

2.5.1 Ray-Volume Intersection

Kay and Kayjia developed an algorithm for speeding up ray-object intersection calculations that is several times faster than other published algorithms [Kay and Kajiya 1986] and is widely used in graphics where speed is usually a requirement. The algorithm applies to any object, but in this thesis it is used to find intersection of a ray with the volume boundary, yielding intervals on the ray where it enters a boundary and exits it, given as t_{start} and t_{end} respectively. Objects are bounded by so called slabs, which can be made to fit convex hulls arbitrarily tightly. A slab is basically the space between two parallel planes. The way the algorithm forms these slabs(boundaries) around objects is out of scope here since we already have a boundary for the volume, a cube. Further information can be found in the original paper [Kay and Kajiya 1986].

2.6 Filtering

Convolution is an important step in producing a final result, which removes high frequency noise generated by the Monte Carlo simulation methods. This is demonstrated in figure 2.5 where two 2D kernels, a signal and a filter, are convolved. The convolution of two real-valued functions, or kernels, is defined as:

$$(f * g)(x) = \int_{\mathbf{R}^d} f(y)g(x-y)\,dy = \int_{\mathbf{R}^d} f(x-y)g(y)\,dy \tag{2.13}$$

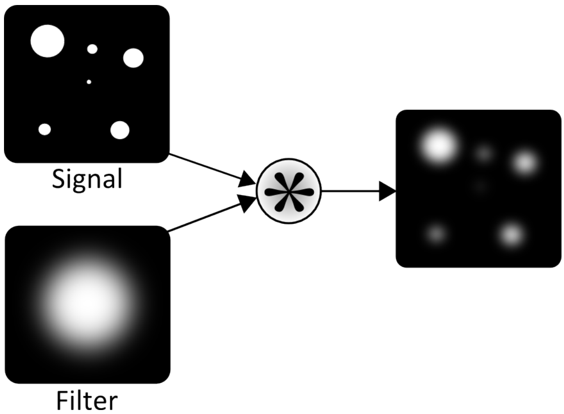

Figure 2.5: The convolution process of 2D kernels

Solving an integral equation like that computationally within acceptable execution times is easily done today, thanks to Fourier transforms. The convolution theorem states that the Fourier transform of a convolution is the point-wise product of Fourier transforms. Convolution in one domain(e.g. time domain) equals point-wise multiplication in the other domain (e.g. frequency domain). Let $\mathcal{F}(f)$ denote a Fourier Transform of a function f, $*$ the convolution of two functions and $f \cdot g$ the point-wise multiplication of f and g respectively, then:

$$\begin{aligned} \mathcal{F}\{f * g\} &= \mathcal{F}\{f\} \cdot \mathcal{F}\{g\} \\ f * g &= \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\} \end{aligned} \tag{2.14}$$

Discrete Fourier Transform is used for doing Fourier transforms of discrete functions/sequences and can be computed efficiently using the Fast Fourier Transform (FFT). FFT algorithms bring the computational complexity of evaluating Discrete Fourier Transforms from $O(N^2)$ to $O(N \log N)$. The FFT therefore not only makes the convolution much simpler in analytical terms (Eq. 2.13 compared to Eq. 2.14) but it also speeds up the computations significantly, making it very suitable for large scale simulations.

Note that the result of a Fourier Transform of a real-valued function is in complex space: $\mathcal{F}\colon \mathbb{R}^n \to \mathbb{C}^n$ so the pointwise product of Fourier transforms is in fact a point-wise product of complex numbers as defined in the following equation:

$$(a+bi)(c+di) = ac + bci + adi + bdi^2 = (ac - bd) + (bc + ad)i \qquad (2.15)$$

2.7 Random number generators (RNGs)

This project relies heavily on random numbers for everything from modeling the structure of nebulae to solving integrals using Monte-Carlo simulations. Neither CUDA nor shaders have built-in random number generators. Several types of random number generators exist:

Pseudo random number generators (PRNGs) also called Deterministic random bit generator (DRBG)[Barker et al. 2005] are algorithms that produce a sequence of bits that are uniquely determined from an initial value called a seed.

True random number generators (TRNGs) use a physical source of randomness to provide truly unpredictable numbers. Most operate by measuring unpredictable natural processes, such as thermal noise, atmospheric noise or nuclear decay[Jun and Kocher 1999]. TRNGs are mainly used in cryptography due to their unpredictable nature. They are however too slow for simulation purposes but are sometimes used in combination with PRNGs as random seed generators.

Quasi random number generators(QRNGs) aim to construct point sets which fill out the s-dimensional (s-D) unit cube as uniformly as possible. Sequences produced by QRNGs are more uniform than pseudo-random sequences. [Sen and Reese 2006]

PRNGs can give good random sequences and are very fast, which makes them very suitable to GPU computing. In any parallel processing framework, such as CUDA, each thread can use it's own index, or thread ID as a seed into the RNGs, given that each thread ID is unique in the whole simulation. This produces a unique uniformly sampled sequence of random numbers for each thread. Of the many PRNG implementations that exist, one with high enough period to fit the task at hand must be chosen. Periodicity, or the period of a RNG, is the maximum length of the random sequence it generates before it begins to repeat itself. Since PRNGs need to be seeded with initial values, all experiments are repeatable. Given a specific seed, the PRNG will always give the same sequence of random numbers, which is often very important to be able to reproduce results. PRNGs are used in 3 different parts of the project:

2.7.1 Modeling dust density: 3D Noise

A 3d noise function, defined as $f\colon \mathbb{R}^3 \to [-1,1]$, was chosen to give an acceptable approximation to dust densities in interstellar-space. A noise function is not a typical random function that tries to produce white noise, rather it gives repeatable and smooth results on a well-specified range.

For this project, Perlin's Simplex Noise was chosen as it has lower computational complexity, $O(n)$ for each dimension n compared to $O(2^n)$ of classic Noise, fewer multiplications, scales to higher dimensions (3 in this case) and has no directional artifacts [Gustavson 2005]. The last part is particularly important in this project to be able to produce naturally looking dust density distributions in 3 dimensions. Stefan Gustavson's implementation of Simplex noise [Gustavson 2005] was used with minor adjustments to make it run on parallel threads in CUDA.
To get a variety of high- and low-frequency dust density distributions, several octaves of noise are needed. The turbulence function ([Perlin 1985; Frisvad and Wyvill 2007]) can be used to achieve this. Although, to avoid discontinuities and provide smooth noise, the absolute value of the noise function is removed:

$$\mathbf{turbulence}_{smooth}(x) = \sum_{f=f_{low}}^{f_{high}} \frac{\mathbf{noise}(2^f x)}{2^f} \tag{2.16}$$

where **noise** is the 3D Simplex noise function. Note that, like the noise function itself, this function generates noise in the range of [-1, 1] which needs to be taken care of when used in modeling dust density, see Eq. 3.2 on page 22.

2.7.2 Monte Carlo random sampling in CUDA

Stochastic photon tracing is a Monte Carlo simulation technique and therefore relies heavily on RNGs to generate random samples. The RNGs must have a high period, since theoretically, depending on input data, very many random samples may need to be taken at each thread. Another criteria is good statistical quality, the RNG must be able to produce unique, uncorrelated random streams on each parallel node. A suitable RNG was found to be a three-component combined Tausworthe(*"taus88"*) and a 32-bit Linear Congruential Generator(LCG) as described in [Howes and Thomas 2007, Ch. 37]. Individually these RNGs provide relatively good statistical quality but combined they give random streams with statistical defects completely removed. This RNG comprises four 32-bit values and provides an overall period of around 2^{121} [Howes and Thomas 2007].

2.7.3 Random numbers in GPU Shaders

Without random sampling, the results of the volume ray marching algorithm, described in Chapter 3.3, show visual artifacts in terms of aliasing. To overcome this problem, the entry-point of each fragment into the volume boundary must be randomly translated along the incident direction ray. Shaders do not offer any built in random functions so one needs to be implemented by hand, one that gives random sequences unique to each fragment(pixel). Only a single random sample is needed per fragment so any simple pseudo-random function suffices, such as the following one of unknown origin but widely used in graphics:

$$rand(\tilde{\mathbf{x}}, \vec{a}, b) = frac(sin(x \cdot a) * b) \tag{2.17}$$

where $\tilde{\mathbf{x}}$ is the normalized 2D coordinate of a pixel, $\vec{a}$ a 2-dimensional random vector and b is a random constant. The function frac returns the fractional (non-integer) part of a real number x and is defined as follows:

$$frac(x) = \begin{cases} x - \lfloor x \rfloor & \text{if } x \geq 0 \\ x - \lceil x \rceil & \text{if } x < 0 \end{cases} \tag{2.18}$$

As an example, given $\tilde{\mathbf{x}}$ as the normalized 2-dimensional coordinate of each pixel ($\tilde{\mathbf{x}}_\mathbf{i} = [0, 1]$), the following call to this random function would produce a pseudo-random number between 0 and 1 for each pixel: $rand(\tilde{\mathbf{x}}, [12.9898, 78.233], 43758.5453)$.

Chapter 3

Implementation

3.1 Overview

The implementation is split into two parts since this is a two-pass algorithm. The first one being a pre-computation stage where the radiative transfer equation (RTE) is solved for N^3 voxels where N is in the power of 2, e.g. 128, 256, 512, for sake of old GPU habits and simplification. In terms of memory usage, 256^3 voxels require approximately 262MB of memory on the GPU only for storing the results of the computation:

$$\frac{256^3 \times 4 \times sizeof(float)}{1024 \cdot 1000} \approx 262MB \tag{3.1}$$

Since each voxel is a 4-component RGBA[1] vector of floating point numbers, each taking 4 bytes of memory (sizeof(float) = 4bytes).
The total memory required by the algorithm in total is about twice the amount derived in Eq.3.1 as memory is required for storing photons, FFT kernels and other things. The pre-computation step is executed nearly solely on the GPU using CUDA. The results are copied into an OpenGL texture-memory address made available to the shaders for visualization. The following sections describe

[1] Red,Green,Blue,Alpha

the implementations of both the pre-computation and the volume visualization algorithms in detail.

3.2 Precomputation

The precomputation step discretizes the volume into N^3 voxels, forming a 3-dimensional array of voxels. It allocates memory required by the algorithms and handles the execution of parallel threads on the GPU through CUDA. Memory must be allocated specifically on the GPU and the results later copied from GPU memory to the host memory.

3.2.1 Dust density

Each voxel, indexed by a 3D coordinate $\tilde{\mathbf{v}} = [x, y, z] \cdot \dfrac{1}{N}$ is assigned a dust density value which is calculated in the following way:

$$\begin{aligned} density(\tilde{\mathbf{v}}) = \quad & cubic(|(\tilde{\mathbf{v}} - \tilde{\mathbf{s}})|, r) \quad + \\ & max(0.0, turbulence(\tilde{\mathbf{v}} \cdot \mathbf{t})) \end{aligned} \tag{3.2}$$

where $\tilde{\mathbf{s}}$ is the position of a star, t is used to down-scale the output of the turbulence function and *cubic* is a cubic-filter function [Frisvad and Wyvill 2007] defined as:

$$cubic(d, r) = \begin{cases} (1 - d^2/r^2)^3 & , d^2 < r^2 \\ 0 & , d^2 \geq r^2 \end{cases} \tag{3.3}$$

The *cubic* function serves the purpose of generating density at the star's location and tightly around it, where the parameter r is used to control the radius of the star. The turbulence function is explained in Eq. 2.16 on page 19. Once the dust densities have been calculated for the whole grid, the values are uploaded to a GPU texture for quick lookup in the device function that solve the RTE.

3.2.2 Photon tracing

The next step of the algorithm is to trace the actual photons and store them in a photon array. The photons must be traced separately for each of the three color-bands, RGB, as scattering properties are different for different wavelengths of

electromagnetic radiation. To adapt the photon-tracing algorithm to the CUDA kernel programming-model, each thread is assigned a task to trace N photons. A total of N^3 photons are traced for each channel and the GPU executes a number of threads in parallel depending on the type of GPU. A card that has 256 CUDA cores can execute two-dimensional thread-blocks of 16^2 threads. Each thread is assigned a 2D thread-ID which combined with an 2D ID of the current thread-block gives a unique ID for each particular execution.
As argued before, each thread traces N photons from the star's origin. Every photon traced from the sun is initialized with a random normalized direction $\tilde{\mathbf{d}}$, sampled isotropically on a sphere. Here I simply randomly sampled from the Henyey-Greenstein phase function with a g parameter of 0, which gives isotropic samples on a sphere. The length of this direction-vector, i.e. the exact distance to the next event of this photon is found by the following equation:

$$\nabla t = \frac{-\log(\xi)}{\sigma_t} \cdot w_{band} \tag{3.4}$$

σ_t is directly proportional to the dust density at the current *position* of the traced photon and ξ is a uniformly sampled random variable. The w_{band} represents a weight, or an extinction factor, for the current band being traced. These extinction factors have been found to be $R \approx 0.748$, $G \approx 1.0$, $B \approx 1.324$. In dense dust clouds the extinction factors become $R \approx 0.8$, $G \approx 1.0$, $B \approx 1.2$ [Magnor et al. 2005]. The position of the next scattering event for a photon is then defined as:

$$\tilde{\mathbf{o}}_i = \tilde{\mathbf{o}}_{i-1} + (\tilde{\mathbf{d}} \cdot \nabla t) \tag{3.5}$$

If the dust density at $\tilde{\mathbf{o}}_{i-1}$ is below a given scattering-threshold the photon is simply traced forward along $\tilde{\mathbf{d}}$ until it either reaches a point with dust density above the threshold or exits the grid and it's flux (or power) is set to 0. Once a photon reaches a position where dust is present, a Russian roulette technique is used to determine whether the photon is scattered forward or absorbed. Russian roulette is a standard Monte Carlo technique introduced to speed up computations in particle physics and later applied to graphics [Jensen 2001].
It can be thought of as an importance-sampling technique where the probability distribution function is used to eliminate unimportant parts of the domain. Here the technique is used to determine whether a photon is scattered or absorbed. Here the scattering-albedo a is introduced which gives the probability of scattering in a given medium, in this case interstellar-dust. Scattering albedo of 0 would not give any scattering while an albedo of 1 would yield highly scattering materials. The Russian roulette algorithm that determines if a photon is scattered or absorbed is shown in Algorithm 1:

In case of scattering, a new direction is simply sampled from the Henyey-Greenstein phase function with the original photon direction and the constant

Algorithm 1 Russian roulette determining scattering events

$\xi \leftarrow$ random()
if $\xi < a$ **then**
  Scatter photon
else
  Absorb photon
end if

g as parameters:

$$\tilde{\mathbf{d}}_{i+1} = \text{sampleHG}\left(\tilde{\mathbf{d}}_i, g\right) \tag{3.6}$$

The energy, or the flux of the photons, emitted by the star into the nebula is a constant. As discussed in Ch. 6, this approach can be extended to accurately model the energy output and spectrum and of a star, given parameters like size, mass, temperature and composition. The photon tracing algorithm is explained in Algorithm 2

Algorithm 2 Photon tracing

Require: $\mathbf{D} \leftarrow$ Dust density lookup texture
Require: $w \leftarrow$ Band weight
  $\vec{o} \leftarrow \vec{o}_{star}$
  $\vec{d} \leftarrow$ sampleHG(g=0)
  while stored photons ¡ N **do**
    $\sigma_t \leftarrow \mathbf{D}(\vec{o})$
    $\nabla t \leftarrow \dfrac{-log(\xi_1) \cdot w}{\sigma_t}$
    $\vec{o} \leftarrow \vec{o} + \vec{d} \cdot \nabla t$
    if $\vec{o}$ is outside volume boundary **then**
      $\vec{o} \leftarrow \vec{o}_{star}$
      $\vec{d} \leftarrow$ sampleIsotropic()
      Store photon at $\vec{o}_i$ with 0 flux
    else
      Store photon at $\vec{o}_i$ with flux constant
      if $\xi_2 < a$ **then**
        $\vec{d} \leftarrow sampleHG(\vec{d}, g)$
      else
        $\vec{o} \leftarrow \vec{o}_{star}$
        $\vec{d} \leftarrow$ sampleIsotropic()
      end if
    end if
  end while

3.2.3 Convolution and gathering

After the photons have been traced into an array, a signal kernel **S** is filled with the flux of every photon according to the photon's position within the unit-grid. In the start of this chapter it was stated that the volume was discretized into voxels. Each voxel therefore represents a cubic boundary inside the volume and has the volume of $\frac{1}{N^3}$ since the whole volume has been divided into N^3 voxels. If a photon was stored at position $[0.5, 0.5, 0.5]$ and the dimension of the volume discretized into 128^3 voxels, the photon would add to the illumination of a voxel indexed in by the 3 integers: $[0.5{\cdot}128, \quad 0.5{\cdot}128, \quad 0.5{\cdot}128] = [64, 64, 64]$.

The algorithm for producing a radiative transfer solution for the all three channels is as follows:

Algorithm 3 Radiative transfer equation solved

Require: $\tilde{\mathbf{w}} \leftarrow$ Band weights
Require: $\mathbf{F} \leftarrow$ Cubic filter kernel
  Initialize result matrix $\mathbf{R}$
  for $i = 1$ to 3 **do**
    Trace Photons for using weight $\tilde{\mathbf{w}}_i$
    Create signal kernel $\mathbf{S}$ and illuminate with photon flux values
    $\mathbf{S} = \mathcal{F}^{-1}\{\mathcal{F}\{\mathbf{S}\} \cdot \mathcal{F}\{\mathbf{F}\}\}$
    $\mathbf{R}_i = \mathbf{S}$
  end for

A simplified version of the ray marching part of the algorithm is explained in Fig. 3.1, which is a modified version of Fig. 2.4 on page 14.

Since we are limited by a number of photons we can trace within acceptable time limit, we can't trace enough photons to represent the total illumination of the entire volume. A single photon may be the only one close to number of voxels. Since it transports only a fraction of the light source power, it cannot say how much light the surrounding region receives. Since every photon only radiates the exact voxel it *hits*, a method is needed to perform photon-density estimation for every voxel to get a radiance estimate based on the surrounding photons inside a given filter radius.
This is where convolution of the signal, storing the flux of the traced photons, and a filter kernel, comes into play. The convolution process using FFT is explained in detail in chapter 2.6.

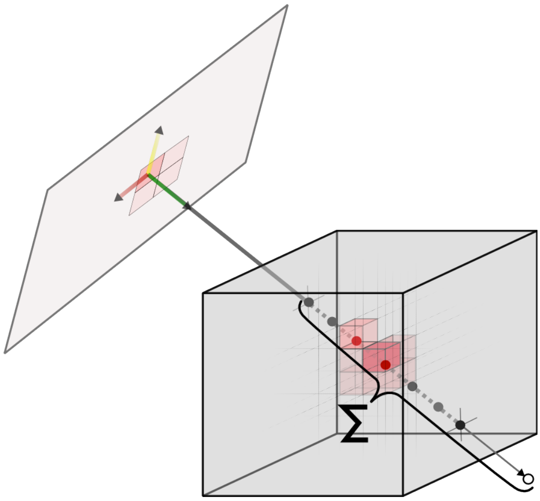

Figure 3.1: Simplified demonstration of the summation of samples taken along the ray, including a background-sample after the ray exits the volume.

3.2.4 Parallelization

Cuda threads are executed in parallel thread blocks that each consists of preferably the maximum amount of parallel threads the GPU can handle. The combined ID of a thread and it's parent thread-block gives a unique index into an array which each thread can safely write it's results into without risking a data-race condition with other threads. This is demonstrated in the Fig. 3.2

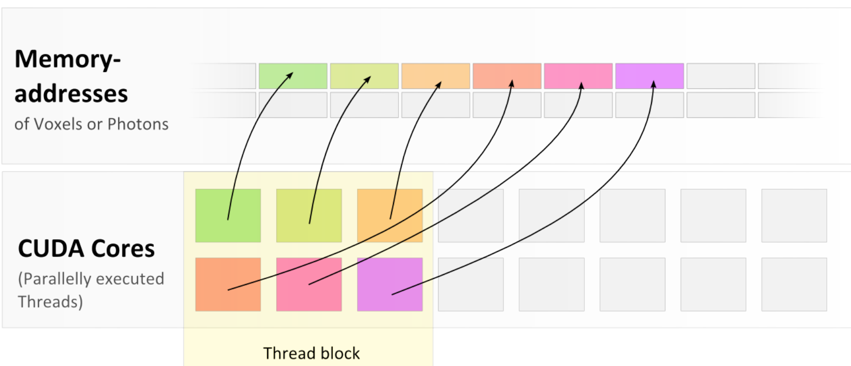

Figure 3.2: A diagram showing thread-blocks containing threads where each thread references a specific address of memory to work with.

Since threads are given specific indices or ranges to work with, there is no need for implementing mutex-based locking of addresses.

3.3 Real-time visualization

Once the pre-computation of the volume is complete, it is uploaded to the shaders as a 3D RGBA-texture. Shaders provide fast lookup-functions for textures, indexed by unit-vector where the textures are unit-boundaries. Additionally, cube-maps that are used for background-color lookup are uploaded to the shaders.
The implementation is a single-pass rendering algorithm that does not require any passes to be rendered into a frame-buffer as an input into a second pass. A single-pass algorithm requires less resources, eliminates the complexity of rendering to a FBO[2] and gives higher frame-rates.
The shader is applied to a very large cube which encloses the the camera so it completely fills the entire rendering canvas at all times. This is a proxy-geometry which forces the fragment shader to process the entire output image, not just the ones where the object it is applied to is visible. Since the volume is bounded by a unit cube, the shader can simply trace a ray for each pixel into the scene and test it's intersection with the boundaries of an imaginary unit-cube. If a ray intersects with the bounding-box then ray-marching is used to compute the color value of that pixel, otherwise the pixel is set to the value of a background function.

The theory behind ray marching is explained in chapter 2.5. Some preparations are needed before the ray marching algorithm can start. A point of origin must be determined along with the direction of the ray into the scene. The origin of the ray is the center of the camera and the direction of the ray needs to be translated by the position of the fragment on the image-plane. To perform this translation, we need a coordinate-system for the image-plane. We know the normal of the plane is the vector from the eye($\vec{e}$) to a given look-at point $\vec{p}$ in front of it. From these vectors, we can derive the other two that form the coordinate-system $\mathbf{V}$:

$$\begin{aligned} \mathbf{V}_{normal} &= |\vec{p} - \vec{e}| \\ \mathbf{V}_x &= |\mathbf{V}_{normal} \times \vec{up}| \\ \mathbf{V}_y &= |\mathbf{V}_x \times \mathbf{V}_{normal}| \end{aligned} \tag{3.7}$$

[2]Frame-Buffer Object

The origin and the direction of the ray are defined

$$\begin{aligned} \mathbf{r}_o = &\quad \vec{e} \\ \mathbf{r}_d = &\quad |\mathbf{V}_{normal} + \mathbf{V}_x \cdot f_x + \mathbf{V}_y \cdot f_y| \end{aligned} \tag{3.8}$$

where f is the two-dimensional normalized index of a fragment. Given a step-size ∇t everything is ready for the ray marching algorithm to proceed:

Algorithm 4 Volume ray marching

Require: $\mathbf{T} \leftarrow$ Volume lookup texture
Require: $\mathbf{B} \leftarrow$ Background lookup function
  if ray does **not** intersect with boundary **then**
    return $\mathbf{B}(\mathbf{r}_d)$
  end if
  $\mathbf{t}_{near}, \mathbf{t}_{far} \leftarrow$ Boundary intersection interval
  $\mathbf{t} = \mathbf{t}_{near} + (\xi \cdot \text{offsetScale})$
  $\mathbf{C} \leftarrow$ (accumulated color) initialize to 0
  $\mathbf{A} \leftarrow$ (accumulated alpha) initialize to 0
  for $i = 0$ to maxSteps **do**
    $\mathbf{C} \leftarrow \mathbf{C} + \mathbf{T}(\mathbf{r}_o)_{RGB} \cdot \mathbf{T}(\mathbf{r}_o)_A$
    $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{T}(\mathbf{r}_o)_A$
    $\mathbf{t} \leftarrow \mathbf{t} + \nabla t$
    $\mathbf{r}_o \leftarrow \mathbf{r}_o + \mathbf{r}_d \cdot \mathbf{t}$
    if $\mathbf{t} > \mathbf{t}_{far}$ **or** $A \geq 1$ **then**
      exit for
    end if
  end for
  return $(\mathbf{C} \cdot \mathbf{A}) + (\mathbf{B}(\mathbf{r}_d) \cdot (1 - \mathbf{A})$

As mentioned in Chapter 2.7.3, without randomly adjusting t_{near} for each pixel, the volume rendering algorithm suffers from aliasing. Aliasing is demonstrated on the left image of Fig. 3.3 and compared to an image rendered with the method described above, which removes or greatly reduces aliasing.

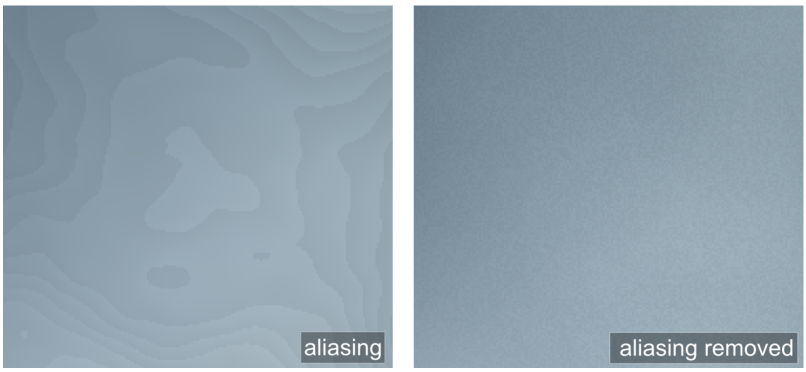

Figure 3.3: Aliasing demonstrated and removed with per-pixel random offset of t_{near}

3.4 Tools

The application is composed of a GUI[3] written in C# .NET, calculation modules written in C++ and CUDA and Shaders written in the Cg shading language. The role of the GUI is to acquire model input parameter from the user and handle realtime rendering of the resulting volume using the shaders. The architectural design is outlined in Fig. 3.4

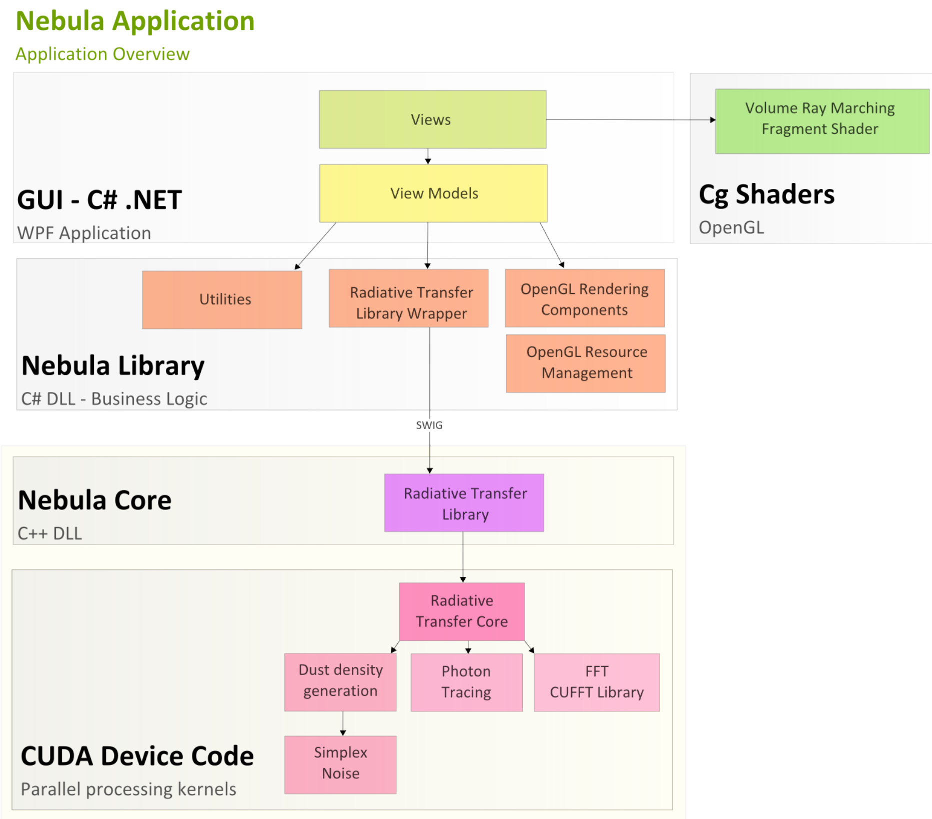

Figure 3.4: Nebula application overview shows the composition of different tiers and libraries.

[3]Graphical User Interface

3.4.1 GUI

The GUI is programmed in C# and uses WPF[4] which is a graphical subsystem for rendering next-generation user interfaces on Windows-based applications and is a part of the .NET framework 3.5. The main reason for using WPF is the power of it's data-binding capabilities and the clear separation of user interface and business logic. Combined with the recently established MVVM[5] pattern, the quality of the code behind the user-interface is greatly increased and all data communication, termed data binding, between the view and the view-model is defined in XML.
OpenGL rendering is made possible with the OpenTK library for .NET. Custom libraries were made for handling OpenGL textures, Cg shaders, scene rendering and resource management.

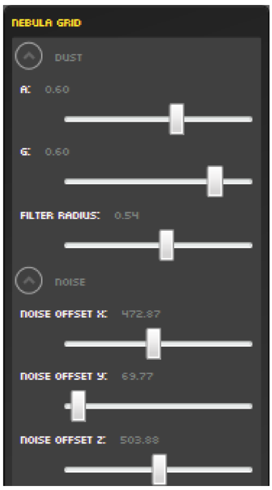

Figure 3.5: A part of the UI that controls a few of the model input-parameters.

Figure 3.5 shows one of the panels which handle model-input-parameters through data-binding. Through this specific panel the user can change the appearance of the dust region, the scattering albedo(a), the anisotropic-scattering factor (g) as well as the convolution-filter radius.

[4]Windows Presentation Foundation
[5]Model-View-ViewModel

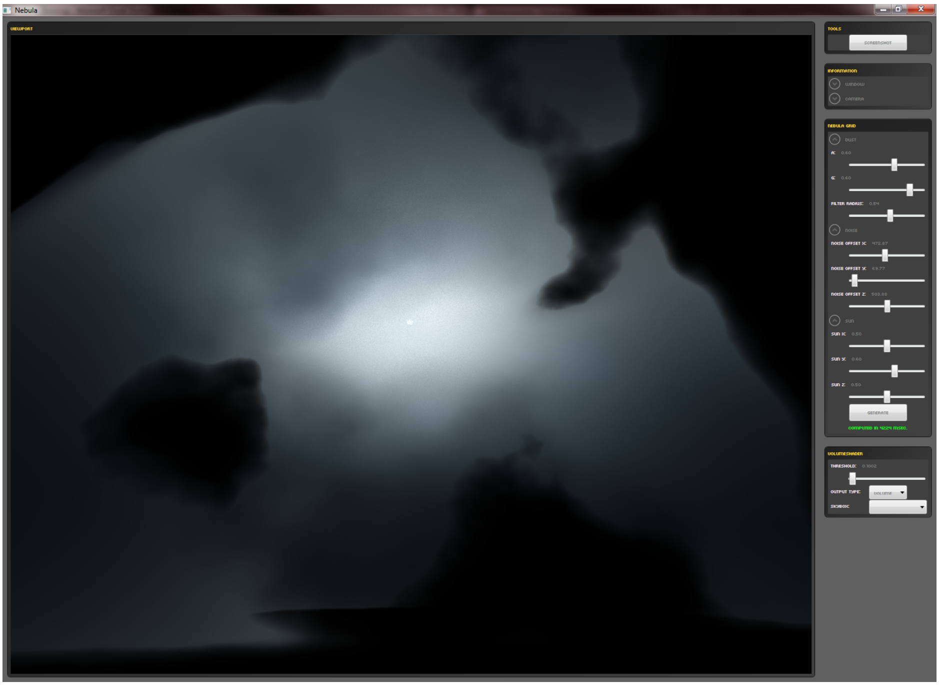

Figure 3.6: A screenshot of the entire application window.

3.4.2 Shaders

There are a few shader languages commonly in used in modern graphical applications, including games and scientific applications. I chose NVIDIA's Cg Shader framework for the following reasons:

1. They offer a great flexibility with their FX format which can be used to include texture definitions and to control the OpenGL state machine (blending, culling, etc.) using a simple script combined with the shader source.
2. They are cross-platform in terms of graphics API, in other words, they work for both OpenGL and Direct3D.
3. In the shader script, it is possible to select from wide variety of shader profiles, including GLSL. The CgFX format can thus mimic the syntax of other types of shader-languages.

3.4.3 Core library

The core library is composed of a C++ DLL which manages memory on the host, serial executions and calling the CUDA kernel functions. The CUDA kernel functions are defined in separate CUDA source files (.cu) and compiled with the CUDA compiler and then linked together with the C++ library. See figure 3.4 for an overview.

3.4.4 Utilities

A few utility programs were implemented to ease the development process. One was made to randomly sample the Henyey-Greenstein phase function and plot the outcome in 3D using different values of g and was used to create Fig. 2.1 on page 9. Another application was made to view the actual photon distribution as particles in a 3-dimensional bounding box.

3.4.5 Libraries used

The application depends on the following libraries

CUFFT: CUFFT is a library provided by CUDA and stands for, as the name suggests, CUDA FFT. It performs FFT and the IFFT[6] of any array of 1,2 and 3 dimensions.

OpenTK: Is a low-level OpenGL wrapper for .NET and enables the use of OpenGL in such applications. See `http://www.opentk.com/` for details

SWIG: SWIG stands for Simplified Wrapper and Interface Generator and was used to make a C# wrapper of the C++ library. With a single command, using the same scripts, wrappers for other languages can be created with ease, including Python, Java, R and Matlab/Octave. This means that the radiative transfer library itself can be used as a library in almost any programming language.

[6] Inverse FFT

Chapter 4

Results

The results of the implementation can be viewed from two separate angles. One being performance, or how fast the algorithm calculates the radiative transfer and the other being the actual visual output compared to expected results according to theory. These are discussed separately in the following sections.

4.1 Execution times

The specifications of the PC used to run the calculations were as follows: Intel Core i7 CPU, 4GB RAM, NVidia GeForce 250 GTS GPU with 128 CUDA cores. The volume was made of 128^3 voxels and $128^3 \approx 2.1M$ photons were traced for each color band for a total of approximately $6.3M$ photons. The total pre-computation time, for generating the volume, was $\approx 4013ms$ of which it took $3150ms$ to run the only part of the algorithm that was not parallelized. Figure 4.1 shows how much time was spent on each part of the radiative transfer algorithm.

The right-side of the graph in Fig. 4.1 represents the largest part of the algorithm, which is executed in parallel on the GPU. The parallelized part of the radiative transfer algorithm thus only takes $863ms$ to compute for approximately $6.3M$ photons, including dust density generation, convolution and mem-

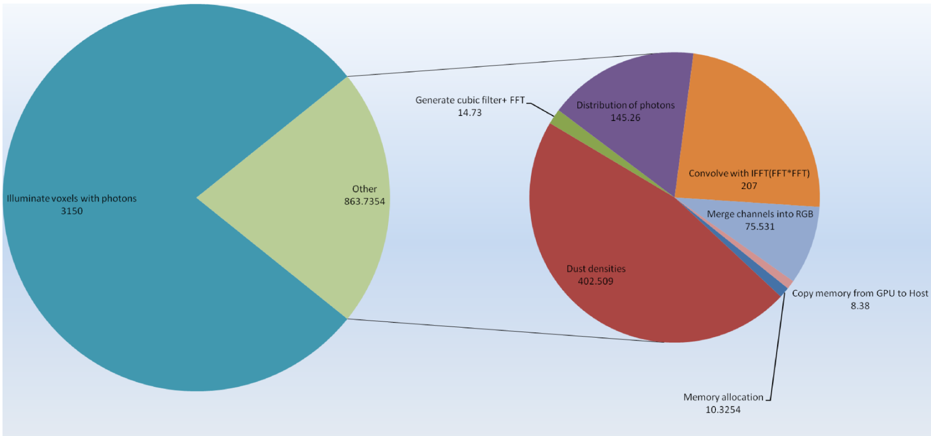

Figure 4.1: Execution times in milliseconds, dominated by the serial-process of illuminating voxels with the photons

ory transfers. These parts all fall under the *embarrassingly parallel* criteria as each individual task assigned to a thread is completely independent of the others. They therefore scale according to Amdahl's law of the maximum speed-up(S) expected by parallelizing portions of a serial program, as defined:

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{4.1}$$

where P is the fraction of the total serial execution time taken by the portion of the code that can be parallelized and N is the number of processors, or cores, over which the parallel portion of the code runs [NVIDIA Corporation 2009].

To demonstrate the power of CUDA compared to alternatives, I implemented an OpenMP version of the same dust-density generation function. OpenMP, Open Multi-Processing, or OMP is anAPI which adds multi-core programming functionality to programming languages like C, C++ and Fortran. It is easy to configure how many cores should be used for a given parallel execution. Figure 4.2 shows the huge difference in execution speeds between a single CPU core, 8 CPU cores and 128 CUDA cores respectively. Each one was used to calculate dust densities for 64,128 and 256 cubic-voxels.

As expected, they all show exponential increases in execution-times, following an exponential increase of the number of voxels. However, the CUDA code executes performs the execution considerably faster.

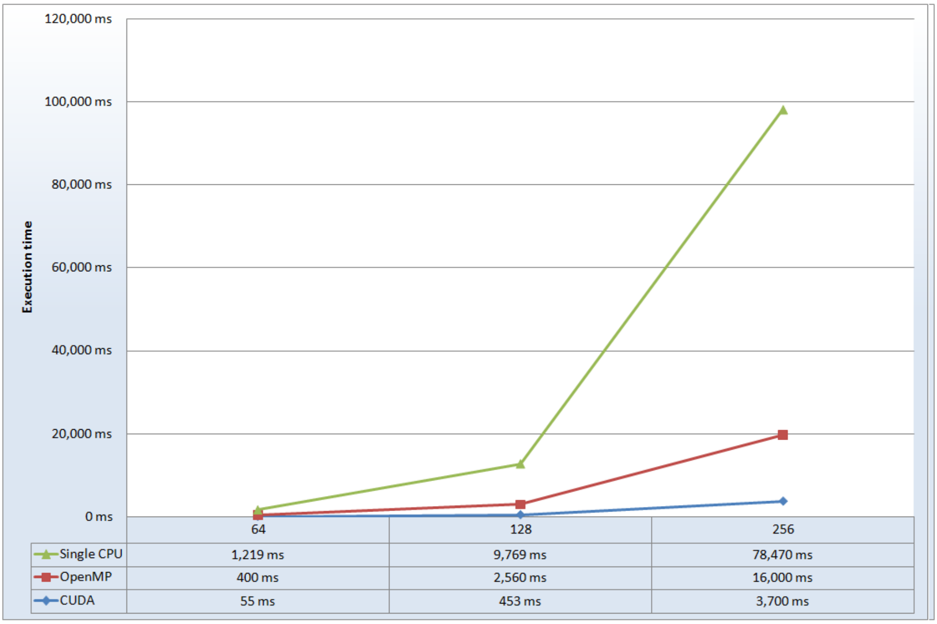

Figure 4.2: Comparison of execution times for generating N^3 voxels using a single cpu, OpenMP(8 CPU-cores) and CUDA(128 GPU-cores)

4.2 Visual results

In this section the actual visual results as rendered by the shaders are presented. Their change in appearance is described for different levels of scattering albedo which clearly shows the result from increased scattering. The scattering albedo is explained in Chapter 2. These images are actual frames exported from the real-time visualization framework.

Figure 4.3 show the radiative transfer in homogeneous dust density distribution, in other words, dust density is equal everywhere in the volume except around the nearest vicinity of the star. The illumination of the dust increases as the scattering albedo increases since photons are distributed further into the dust region. When there is little or no scattering present, it can be seen how blue light penetrates deeper into the dust away from the star, producing the blue halo as evident in the figure. When scattering increases, other color bands add to the total illumination of the dust.
Figures 4.4 and 4.5 show the same effect in modeled nebulae environment of heterogeneous dust density distributions.

Figure 4.3: Homogeneous dust distribution

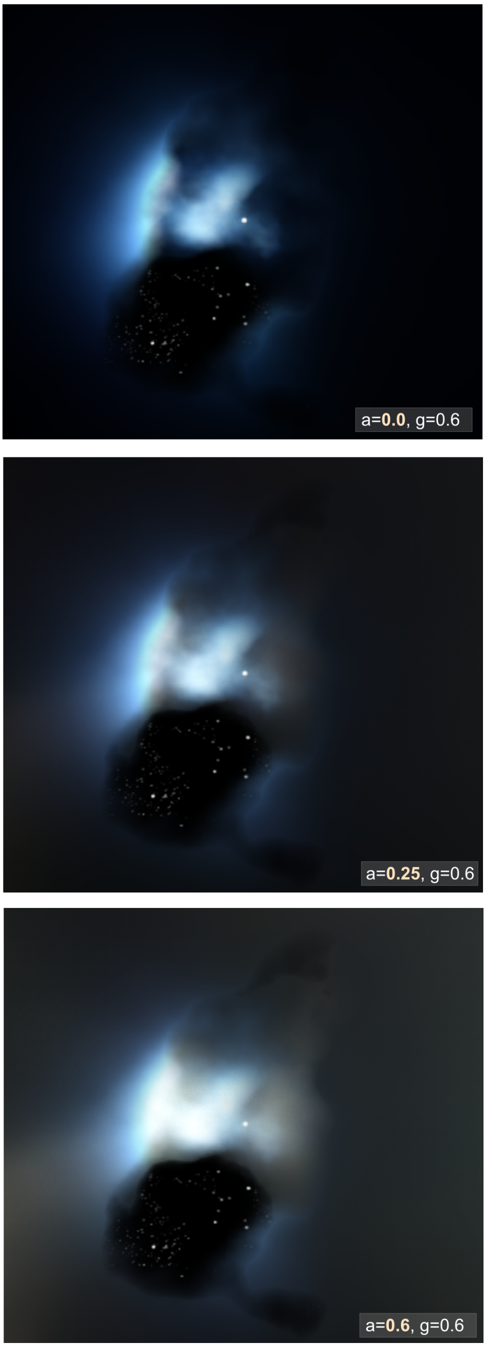

Figure 4.4: Random nebula environment showing changes in radiative transfer when the scattering-albedo is changed

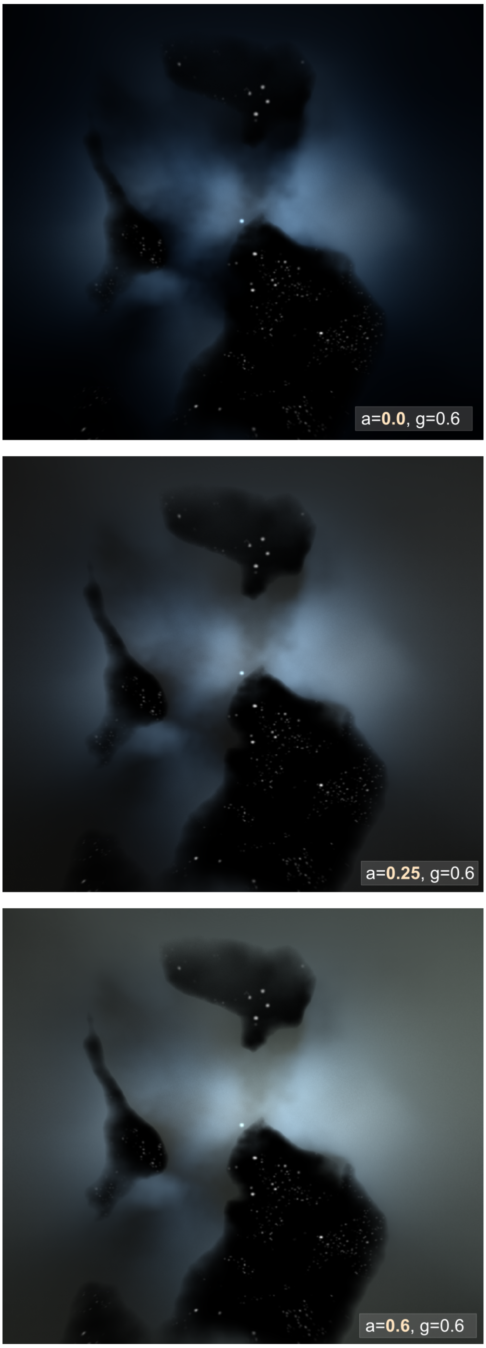

Figure 4.5: Same as in the previous figure, but nebula environment generated from a different seed

Chapter 5

Conclusion

I presented a method for calculating and visualizing radiative transfer in reflection nebulae, using photon mapping, extending previous implementation by e.g. [Magnor et al. 2005], using new recently established methods for efficiently solving radiative transfer in participating media.

Given that the source of radiation and the participating media are static, volume photon tracing is an efficient and accurate method for solving radiative transfer in heterogeneous media such as interstellar dust. With the help of modern GPUs and HPC[1] frameworks such as CUDA, millions of photons can be traced, gathered and filtered into large three-dimensional texel-grids in matter of seconds. Using photon tracing to solve the radiative transfer equation can greatly enhance the quality of the final output, compared to other methods, as it captures multiple scattering events in great detail and depth. The algorithm is also easily extended to more complex models as discussed in chapter 6.

Pre-calculating the radiative transfer and storing the result in a volume has many advantages as the volume can be visualized and inspected in real-time on high frame-rates. The graphical user interface enables the researcher to enter input parameters into the model, such as star- and dust-properties and almost

[1] High-performance computing

instantly see the radiative transfer results for real-time visualization. Although speed was not a primary goal for this project, the calculation efficiency of GPUs by far exceeded expectations, compared to it's CPU counterpart.

CHAPTER 6

Discussion

In the following sections, limitations and improvements of presented methods are discussed.

6.1 Performance

As stated before and demonstrated in Fig. 4.1, 78% of the execution-time for the pre-computation is spent running over an array of photons to illuminate voxels corresponding to their positions inside the volume. The reasons for not parallelizing this part are the following:

Not easily parallelized: The position of every traced photon can not be predetermined due to the random nature of stochastic processes. Therefore, there is a high risk of data-race-conditions between parallel threads as threads can easily trace photons into the same *cell* at the same time.

Current limitation of CUDA: The GPU used for development did not support latest CUDA features, floating point atomic functions, which otherwise would have made the parallelization of this part easy.

Out of scope: Parallelizing this part was considered a nice extension if time permitted and doesn't contribute directly to the goals of this project.

There are 3 approaches I suggest for parallelizing this part.

CUDA or OpenMP: Implement custom parallel algorithms that involve mutex-based locking of voxel memory-addresses.

CUDA atomic functions: In the latest CUDA drivers, support has been added for floating-point atomic functions which have a built-in mechanism to handle locking of memory addresses and support. This will therefore not add any complexity to the existing algorithms.

KD-Trees: The use of a spatial-partitioning data-structure like a KD-Tree can partition the volume into smaller boundaries, each containing a number of photons. Each block can then be executed in parallel. This approach however, introduces various problems with how edges of boundaries are handled and has an overhead of building the actual KD-Tree that might be equal or higher than that of the current implementation.

6.2 Model extensions

The current implementation is very open to extensions. The radiative transfer model can be extended to account for more complex scenes:

Ionization: model ionization of gas particles to include emission of these particles.

Particle size distribution: instead of using a fixed size of dust-grains in reflection nebulae, more complex models could sample from particle-size distributions as described in [Andersen 2007; Gordon 2004].

Specifications of a star: the current approach uses a constant flux output of a given star. This can be extended with detailed models describing the type of the star and from that calculate the exact power-output and spectrum.

Appendix A

Code listings

Following are selected code segments which demonstrate some parts of the algorithm in detail.

A.1 CUDA

A.1.1 Density grid generation kernel function

Listing A.1: GenerateDensityGrid Kernel

```
__global__ void generateDensityGrid(float4* data, cudaExtent size, float3 sunPos,
                                    float oneOverX, float oneOverY, float oneOverZ,
                                    const float3 noiseOffset, float densityScale)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = 0, base = x*size.width*size.height + y*size.height;

    for(int z=0; z<size.depth; z++)
    {
```

```
            idx = base + z;
            float3 here = make_float3(x*oneOverX, y*oneOverY, z*oneOverZ);
            float sun = cubic( length(here - sunPos), 0.01f);
            float sct = sun + max(0.0f, turbulence(here + noiseOffset))*densityScale;
            float mag = sun;

            data[idx] = make_float4(mag, mag, mag,
                                    clamp(sct,0.0001f,1.0f)); //min/max dust density
        }
}
```

A.1.2 Filter generation kernel function

Listing A.2: GenerateFilter Kernel

```
__global__ void generateFilter(cufftReal* data, cudaExtent size, float r,
                               float3 center,
                               float oneOverX,float oneOverY,float oneOverZ )
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = 0, base = x*size.width*size.height + y*size.height;
    for(int z=0; z<size.depth; z++)
    {
        idx = base + z;
        float3 here = make_float3(x*oneOverX, y*oneOverY, z*oneOverZ);
        float d = length(here - center);
        data[idx] = cubic(d, r)/(4.0/3.0*M_PI*r*r*r);
    }

}
```

A.1.3 Band combination kernel function

This function combines individual bands into RGB.

Listing A.3: GenerateDensityGrid Kernel

```
__global__ void combine(float4* data, cufftReal* channelData, cudaExtent size, int band)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
```

```
    int idx = 0, base = x*size.width*size.height + y*size.height;

    for(int z=0; z<size.depth; z++)
    {
        idx = base + z;
        switch(band)
        {
        case BAND_RED:
            data[idx].x = channelData[idx];
            break;
        case BAND_GREEN:
            data[idx].y = channelData[idx];
            break;
        case BAND_BLUE:
            data[idx].z = channelData[idx];
            break;

        }
    }
}
```

A.1.4 Photon to light-field kernel function

This function illuminates designated voxels with the flux of the photons that hit them.

Listing A.4: PhotonToLightField Kernel

```
__global__ void computeRT(cufftReal* data, cudaExtent volumeSize, Photon* photons, cudaExtent photo
{
    const unsigned long psize = photonsSize.width*photonsSize.height*photonsSize.depth;
    for(unsigned long pid=0; pid<psize; pid++)
    {
        Photon* p = &photons[pid];

        unsigned int x = (int)(p->pos.x * volumeSize.width);
        unsigned int y = (int)(p->pos.y * volumeSize.width);
        unsigned int z = (int)(p->pos.z * volumeSize.width);
        unsigned int idx = x*volumeSize.width*volumeSize.height + y*volumeSize.height + z;

        data[idx] = p->flux;
    }
}
```

A.1.5 Stochastic Photon tracer

Listing A.5: DistributePhotons Kernel

```
__constant__ __device__ float band_weights[3] = {0.8f, 1.0f, 1.2f};
__global__ void distributePhotons(Photon* data, int dimension,
                                  float3 sunPos, float sunPhi,
                                  float a, float g, int band,
                                  float oneOverX, float oneOverY, float oneOverZ)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = 0, base = x*dimension*dimension + y*dimension;

    //Initialize random per-thread RNG states
    unsigned z1,z2,z3,z4;
    z1 = z2 = z3 = z4 = base + band;

    float s = oneOverX; //stepsize

    float3 o = sunPos; //origin
    float3 d = sampleHG(make_float3(0.f, 1.f, 0.f), 0.0f,
                         random(z1,z2,z3,z4), random(z1,z2,z3,z4)); //direction

    float T_aim;
    int depth = 1;
    while(idx < dimension )
    {
        float density = (idx == 0 ) ? 1.0f : tex3D(voxelsTex, o.x, o.y, o.z).w;

        float sigma_t = max(0.01f, density ) / s; //extinction coeff.
        T_aim = (-log(random(z1,z2,z3,z4)) / sigma_t) * band_weights[band];

        o = o + d*T_aim;
        //check for out of bounds
        if(o.x ≥1 || o.y ≥1 || o.z ≥1 || o.x < 0 || o.y < 0 || o.z < 0)
        {
            o = sunPos;
            d = sampleHG(make_float3(0.f, 1.f, 0.f), 0.0f,
                         random(z1,z2,z3,z4), random(z1,z2,z3,z4)); //direction
            data[base+idx] = Photon(o, 0.0f, depth);
            idx++;
            depth=1;
            continue;
        }

```

```
            if(density < 0.001f )
                continue;
            //store
            data[base+idx] = Photon(o, sunPhi, depth);
            idx++;

            //check if it scatters forward
            if(random(z1,z2,z3,z4) < a && depth < 10)
            {
                d = sampleHG(d, g, random(z1,z2,z3,z4), random(z1,z2,z3,z4)); //direction
                depth++;
            }
            else
            {
                o = sunPos;
                d = sampleHG(make_float3(0.f, 1.f, 0.f), 0.0f,
                           random(z1,z2,z3,z4), random(z1,z2,z3,z4)); //direction
                depth=1;
            }

        }

}
```

A.1.6 Cubic function

Listing A.6: Cubic function

```
__device__ float cubic(float d, float r)
{
    float ds = d*d;
    float rs = r*r;
    if (ds < rs)
    {
        float w = (1.0f - ds/rs);
        return w*w*w;
    }

    return 0.0f;
}
```

A.1.7 Element-wise multiplication of two kernel of complex numbers

Listing A.7: ElementWiseMult

```
__global__ void elementWiseMult3D(cufftComplex* a, cufftComplex* b, cudaExtent size)
{
    int x = blockIdx.x*blockDim.x + threadIdx.x;
    int y = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = 0, base = x*size.width*size.height + y*size.height;
    for(int z=0; z<size.depth; z++)
    {
        idx = base + z;
        a[idx] = complexMul(a[idx], b[idx]);
    }

}
```

A.1.8 Complex number multiplication

Listing A.8: ComplexMul

```
// Complex multiplication
static __device__ inline cufftComplex complexMul(const cufftComplex& a, const cufftComplex& b)
{
    cufftComplex c;
    c.x = a.x * b.x - a.y * b.y;
    c.y = a.x * b.y + a.y * b.x;
    return c;
}
```

A.1.9 Sample Henyey-Greenstein

From the book [Pharr and Humphreys 2004]

Listing A.9: SampleHG

```
// HG sample function, PBR page 713
//w: normal
//g: HGreenstein parameter
//u1: random
//u2: random
__device__ float3 sampleHG(const float3& w, float g, float u1, float u2)
{
    float costheta;
```

```
    if(fabsf(g) < 1e-3)
        costheta = 2.f * u1 -1.f;
    else
    {
        float tmp = (1.f - g*g)/(1.f - g + 2.f*g*u1);
        costheta = -1.f / (2.f * g) * (1.f + g*g - tmp*tmp);
    }

    float sintheta = sqrtf(fmaxf(0.f, 1.f-costheta*costheta));
    float phi = 2.f*M_PI*u2;

    float3 v1,v2;
    coordinateSystem(w, v1, v2);
    return normalize(sphericalDirection(sintheta, costheta, phi, w, v1, v2));
}
```

A.1.10 Compute direction vector given spherical coordinates

From the book [Pharr and Humphreys 2004]

Listing A.10: SphericalDirection

```
//Computes a spherical direction given the parameters and coordinate system
//PBR page 246
__device__ float3 sphericalDirection(float sintheta, float costheta, float phi,
                                     const float3& x, const float3& y, const float3& z)
{
    return sintheta * cosf(phi) * z + sintheta*sinf(phi) * y + costheta*x ;
}
```

A.1.11 Construct a coordinate system for a vector

Corrected version from the book [Pharr and Humphreys 2004]

Listing A.11: CoordinateSystem

```
__device__ void coordinateSystem(const float3& v1, float3& v2, float3& v3)
{
```

```
    if(fabsf(v1.x) > fabsf(v1.y))
    {
        float invLen = 1.f / sqrtf(v1.x*v1.x + v1.z*v1.z);
        v2 = make_float3(-v1.z*invLen, 0.f, v1.x*invLen);
    }
    else
    {
        float invLen = 1.f / sqrtf(v1.y*v1.y + v1.z*v1.z);
        v2 = make_float3(0.f, v1.z*invLen, -v1.y*invLen);
    }

    v3 = cross(v1,v2);
}
```

A.1.12 Main CUDA host function

This function combines individual bands into RGB.

Listing A.12: Main CUDA host function

```
__host__ void generateVolumeCuda(float4* data, const cudaExtent& volumeSize,
                                 Photon* photonArrayHost, const cudaExtent& photons,
                                 float3 sunPos, float sunPhi, float a, float g,
                                 float3 noiseOffset, float filterRadius,
                                 float densityScale )
{
    cufftReal* signal = NULL;
    cufftComplex* signalC = NULL;
    cufftReal* filter = NULL;
    cufftComplex* filterC = NULL;
    float4* gpuData = NULL;
    Photon* photonArray = NULL;
    float oneOverX = 1.0f / volumeSize.width;
    float oneOverY = 1.0f / volumeSize.height;
    float oneOverZ = 1.0f / volumeSize.depth;

    int N = volumeSize.width*volumeSize.height*volumeSize.depth;
    int PN = photons.width*photons.height*photons.depth;
    sunPhi /= N;


    cutilSafeCall(cudaMalloc((void**)&gpuData, N*sizeof(float4)));
    cutilSafeCall(cudaMalloc((void**)&signal, N*sizeof(cufftReal)));
    cutilSafeCall(cudaMalloc((void**)&signalC, N*sizeof(cufftComplex)));
```

```
	cutilSafeCall(cudaMalloc((void**)&filter, N*sizeof(cufftReal)));
	cutilSafeCall(cudaMalloc((void**)&filterC, N*sizeof(cufftComplex)));
	cutilSafeCall(cudaMalloc((void**)&photonArray, PN*sizeof(Photon)));
	cutilCheckMsg("Memory allocation");


	cudaArray* voxelsArray;
	cudaChannelFormatDesc desc = cudaCreateChannelDesc<float4>();
	cutilSafeCall(cudaMalloc3DArray(&voxelsArray, &desc, volumeSize));
	cutilCheckMsg("Texturearray allocation");



	dim3 block(8,8);
	dim3 grid(volumeSize.width/block.x,volumeSize.height/block.y);

	//#. Compute the scattering factors (density map)
	generateDensityGrid<<<grid, block>>>(gpuData, volumeSize, sunPos,
	                                    oneOverX, oneOverY, oneOverZ,
	                                               noiseOffset, densityScale);
	cutilCheckMsg("Density grid generation");

	//#. Copy into a 3D cudaArray and bind to a texture
	 cudaMemcpy3DParms copyParams = {0};
	 copyParams.srcPtr = make_cudaPitchedPtr((void*)gpuData,
	                                                volumeSize.width*sizeof(float4),
	                                                volumeSize.width, volumeSize.height);
	 copyParams.dstArray = voxelsArray;
	 copyParams.extent = volumeSize;
	 copyParams.kind = cudaMemcpyDeviceToDevice;
	 cutilSafeCall( cudaMemcpy3D(&copyParams) );

	//set texture parameters
	 voxelsTex.normalized = true; // access with normalized texture coordinates
	 voxelsTex.filterMode = cudaFilterModeLinear;
	 voxelsTex.addressMode[0] = cudaAddressModeClamp;
	 voxelsTex.addressMode[1] = cudaAddressModeClamp;

	 //bind array to 3D texture
	 cutilSafeCall(cudaBindTextureToArray(voxelsTex, voxelsArray, desc));
	cutilCheckMsg("Binding array to texture");

	//Initialize filter
	generateFilter<<<grid,block>>>(filter, volumeSize, filterRadius,
	                                sunPos, oneOverX, oneOverY, oneOverZ );
	cutilCheckMsg("Generating filter");
```

```
    dim3 pblock(8,8);
    dim3 pgrid(photons.width/pblock.x,photons.height/pblock.y);

    //create FFT plan
    cufftHandle planR2C;
    cufftSafeCall(cufftPlan3d(&planR2C, volumeSize.width, volumeSize.height,
                              volumeSize.depth, CUFFT_R2C));
    cutilCheckMsg("Creating plan R2C");

    cufftHandle planC2R;
    cufftSafeCall(cufftPlan3d(&planC2R, volumeSize.width, volumeSize.height,
                              volumeSize.depth, CUFFT_C2R));
    cutilCheckMsg("Creating plan C2R");


    //perform FFT on filter
    cufftSafeCall(cufftExecR2C(planR2C, filter, filterC));
    cutilCheckMsg("FFT Filter R2C");

    //#. Distribute photons
    for(int band=0; band<3; band++)
    //int band = 0;
    {
        //distribute photons
        distributePhotons<<<pgrid, pblock>>>(photonArray, photons.depth, sunPos, sunPhi,
                                             a, g, band, oneOverX, oneOverY, oneOverZ );
        cutilCheckMsg("Distributing Photons");
        //assign photons to 3d grid (signal)
        computeRT<<<1, 1>>>(signal, volumeSize, photonArray, photons, band);

        //Perform FFT
        cufftSafeCall(cufftExecR2C(planR2C, signal, signalC));
        cutilCheckMsg("Executing FFT R2C");

        elementWiseMult3D<<<grid, block>>>(signalC, filterC, volumeSize);
        cutilCheckMsg("Multiplying complex elements");

        //Perform FFT
        cufftSafeCall(cufftExecC2R(planC2R, signalC, signal ));
        cutilCheckMsg("Executing FFT C2R");


        combine<<<grid, block>>>(gpuData, signal, volumeSize, band);
        cudaMemset(signal, 0, N*sizeof(cufftReal));
    }

```

```
    cutilSafeCall( cudaMemcpy(photonArrayHost, photonArray,
                              photons.width*photons.height*photons.depth*sizeof(Photon),
                              cudaMemcpyDeviceToHost) );
    cutilCheckMsg("Copying photons to host memory");

    //#. Copy results to host memory
    cutilSafeCall( cudaMemcpy(data, gpuData, N*sizeof(float4), cudaMemcpyDeviceToHost) );
    cutilCheckMsg("Copying voxels to host memory");


    //#. Clean up
    cufftDestroy(planR2C);
    cufftDestroy(planC2R);
    cudaFree(gpuData);
    cudaFree(signal);
    cudaFree(signalC);
    cudaFree(filter);
    cudaFree(filterC);
    cudaFree(photonArray);
    cudaFreeArray(voxelsArray);


    //cudaThreadExit(); //just in case
}
```

A.2 Simplex implementation

This is an implementation of Simplex noise [Gustavson 2005] by Jeppe Revall Frisvad [Frisvad and Wyvill 2007] with minor changes to run on CUDA

Listing A.13: Main CUDA host function

```
/*****************************************************
 *
 * This is a C++ version of Stefan Gustavson's implementation
 * of improved Perlin noise and Perlin's simplex noise.
 * See: http://webstaff.itn.liu.se/~stegu/simplexnoise/
 *
 * Code written by Jeppe Revall Frisvad
 * Copyright (c) DTU Informatics, 2009
 *
 *****************************************************/
```

```

#include <cmath>
using namespace std;

namespace SimplexCUDA // Simplex noise in 2D, 3D and 4D
{
  __device__ int grad3[12][3] = {{1,1,0},{-1,1,0},{1,-1,0},{-1,-1,0},
                  {1,0,1},{-1,0,1},{1,0,-1},{-1,0,-1},
                  {0,1,1},{0,-1,1},{0,1,-1},{0,-1,-1}};
  __device__ int grad4[32][4]= {{0,1,1,1}, {0,1,1,-1}, {0,1,-1,1}, {0,1,-1,-1},
                 {0,-1,1,1}, {0,-1,1,-1}, {0,-1,-1,1}, {0,-1,-1,-1},
                 {1,0,1,1}, {1,0,1,-1}, {1,0,-1,1}, {1,0,-1,-1},
                 {-1,0,1,1}, {-1,0,1,-1}, {-1,0,-1,1}, {-1,0,-1,-1},
                 {1,1,0,1}, {1,1,0,-1}, {1,-1,0,1}, {1,-1,0,-1},
                 {-1,1,0,1}, {-1,1,0,-1}, {-1,-1,0,1}, {-1,-1,0,-1},
                 {1,1,1,0}, {1,1,-1,0}, {1,-1,1,0}, {1,-1,-1,0},
                 {-1,1,1,0}, {-1,1,-1,0}, {-1,-1,1,0}, {-1,-1,-1,0}};
  __device__ int perm[512] = {151,160,137,91,90,15,
        131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
        190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
        88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
        77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
        102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
        135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
        5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
        223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
        129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
        251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
        49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
        138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180,
             151,160,137,91,90,15,
        131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
        190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,
        88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,
        77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
        102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,
        135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
        5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,
        223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
        129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
        251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
        49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
        138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
};

 // A lookup table to traverse the simplex around a given point in 4D.
 // Details can be found where this table is used, in the 4D noise method.
```

```
 __device__ int simplex[64][4] = {
   {0,1,2,3},{0,1,3,2},{0,0,0,0},{0,2,3,1},{0,0,0,0},{0,0,0,0},{0,0,0,0},{1,2,3,0},
   {0,2,1,3},{0,0,0,0},{0,3,1,2},{0,3,2,1},{0,0,0,0},{0,0,0,0},{0,0,0,0},{1,3,2,0},
   {0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},
   {1,2,0,3},{0,0,0,0},{1,3,0,2},{0,0,0,0},{0,0,0,0},{0,0,0,0},{2,3,0,1},{2,3,1,0},
   {1,0,2,3},{1,0,3,2},{0,0,0,0},{0,0,0,0},{0,0,0,0},{2,0,3,1},{0,0,0,0},{2,1,3,0},
   {0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},{0,0,0,0},
   {2,0,1,3},{0,0,0,0},{0,0,0,0},{0,0,0,0},{3,0,1,2},{3,0,2,1},{0,0,0,0},{3,1,2,0},
   {2,1,0,3},{0,0,0,0},{0,0,0,0},{0,0,0,0},{3,1,0,2},{0,0,0,0},{3,2,0,1},{3,2,1,0}};

// This method is a *lot* faster than using (int)Math.floor(x)
__device__ int fastfloor(float x) {
  return x>0 ? (int)x : (int)x-1;
}

__device__ float dot(int g[], float x, float y) {
  return g[0]*x + g[1]*y;
}

__device__ float dot(int g[], float x, float y, float z) {
  return g[0]*x + g[1]*y + g[2]*z;
}

__device__ float dot(int g[], float x, float y, float z, float w) {
  return g[0]*x + g[1]*y + g[2]*z + g[3]*w;
}

 // 2D simplex noise
 __device__ float noise(float xin, float yin)
 {
   float n0, n1, n2; // Noise contributions from the three corners
   // Skew the input space to determine which simplex cell we're in
   const float F2 = 0.5*(sqrt(3.0)-1.0);
   float s = (xin+yin)*F2; // Hairy factor for 2D
   int i = fastfloor(xin+s);
   int j = fastfloor(yin+s);
   const float G2 = (3.0-sqrt(3.0))/6.0;
   float t = (i+j)*G2;
   float X0 = i-t; // Unskew the cell origin back to (x,y) space
   float Y0 = j-t;
   float x0 = xin-X0; // The x,y distances from the cell origin
   float y0 = yin-Y0;
   // For the 2D case, the simplex shape is an equilateral triangle.
   // Determine which simplex we are in.
   int i1, j1; // Offsets for second (middle) corner of simplex in (i,j) coords
   if(x0>y0) {i1=1; j1=0;} // lower triangle, XY order: (0,0)->(1,0)->(1,1)
   else {i1=0; j1=1;} // upper triangle, YX order: (0,0)->(0,1)->(1,1)
```

```
    // A step of (1,0) in (i,j) means a step of (1-c,-c) in (x,y), and
    // a step of (0,1) in (i,j) means a step of (-c,1-c) in (x,y), where
    // c = (3-sqrt(3))/6
    float x1 = x0 - i1 + G2; // Offsets for middle corner in (x,y) unskewed coords
    float y1 = y0 - j1 + G2;
    float x2 = x0 - 1.0 + 2.0 * G2; // Offsets for last corner in (x,y) unskewed coords
    float y2 = y0 - 1.0 + 2.0 * G2;
    // Work out the hashed gradient indices of the three simplex corners
    int ii = i & 255;
    int jj = j & 255;
    int gi0 = perm[ii+perm[jj]] % 12;
    int gi1 = perm[ii+i1+perm[jj+j1]] % 12;
    int gi2 = perm[ii+1+perm[jj+1]] % 12;
    // Calculate the contribution from the three corners
    float t0 = 0.5 - x0*x0-y0*y0;
    if(t0<0) n0 = 0.0;
    else {
      t0 *= t0;
      n0 = t0 * t0 * dot(grad3[gi0], x0, y0); // (x,y) of grad3 used for 2D gradient
    }
    float t1 = 0.5 - x1*x1-y1*y1;
    if(t1<0) n1 = 0.0;
    else {
      t1 *= t1;
      n1 = t1 * t1 * dot(grad3[gi1], x1, y1);
    }
    float t2 = 0.5 - x2*x2-y2*y2;
    if(t2<0) n2 = 0.0;
    else {
      t2 *= t2;
      n2 = t2 * t2 * dot(grad3[gi2], x2, y2);
    }
    // Add contributions from each corner to get the final noise value.
    // The result is scaled to return values in the interval [-1,1].
    return 70.0 * (n0 + n1 + n2);
  }


  // 3D simplex noise
  __device__ float noise(float xin, float yin, float zin)
  {
    float n0, n1, n2, n3; // Noise contributions from the four corners
    // Skew the input space to determine which simplex cell we're in
    const float F3 = 1.0/3.0;
    float s = (xin+yin+zin)*F3; // Very nice and simple skew factor for 3D
    int i = fastfloor(xin+s);
    int j = fastfloor(yin+s);
```

```
int k = fastfloor(zin+s);
const float G3 = 1.0/6.0; // Very nice and simple unskew factor, too
float t = (i+j+k)*G3;
float X0 = i-t; // Unskew the cell origin back to (x,y,z) space
float Y0 = j-t;
float Z0 = k-t;
float x0 = xin-X0; // The x,y,z distances from the cell origin
float y0 = yin-Y0;
float z0 = zin-Z0;
// For the 3D case, the simplex shape is a slightly irregular tetrahedron.
// Determine which simplex we are in.
int i1, j1, k1; // Offsets for second corner of simplex in (i,j,k) coords
int i2, j2, k2; // Offsets for third corner of simplex in (i,j,k) coords
if(x0>=y0) {
  if(y0>=z0)
    { i1=1; j1=0; k1=0; i2=1; j2=1; k2=0; } // X Y Z order
    else if(x0>=z0) { i1=1; j1=0; k1=0; i2=1; j2=0; k2=1; } // X Z Y order
    else { i1=0; j1=0; k1=1; i2=1; j2=0; k2=1; } // Z X Y order
  }
else { // x0<y0
  if(y0<z0) { i1=0; j1=0; k1=1; i2=0; j2=1; k2=1; } // Z Y X order
  else if(x0<z0) { i1=0; j1=1; k1=0; i2=0; j2=1; k2=1; } // Y Z X order
  else { i1=0; j1=1; k1=0; i2=1; j2=1; k2=0; } // Y X Z order
}
// A step of (1,0,0) in (i,j,k) means a step of (1-c,-c,-c) in (x,y,z),
// a step of (0,1,0) in (i,j,k) means a step of (-c,1-c,-c) in (x,y,z), and
// a step of (0,0,1) in (i,j,k) means a step of (-c,-c,1-c) in (x,y,z), where
// c = 1/6.
float x1 = x0 - i1 + G3; // Offsets for second corner in (x,y,z) coords
float y1 = y0 - j1 + G3;
float z1 = z0 - k1 + G3;
float x2 = x0 - i2 + 2.0*G3; // Offsets for third corner in (x,y,z) coords
float y2 = y0 - j2 + 2.0*G3;
float z2 = z0 - k2 + 2.0*G3;
float x3 = x0 - 1.0 + 3.0*G3; // Offsets for last corner in (x,y,z) coords
float y3 = y0 - 1.0 + 3.0*G3;
float z3 = z0 - 1.0 + 3.0*G3;
// Work out the hashed gradient indices of the four simplex corners
int ii = i & 255;
int jj = j & 255;
int kk = k & 255;
int gi0 = perm[ii+perm[jj+perm[kk]]] % 12;
int gi1 = perm[ii+i1+perm[jj+j1+perm[kk+k1]]] % 12;
int gi2 = perm[ii+i2+perm[jj+j2+perm[kk+k2]]] % 12;
int gi3 = perm[ii+1+perm[jj+1+perm[kk+1]]] % 12;
// Calculate the contribution from the four corners
float t0 = 0.6 - x0*x0 - y0*y0 - z0*z0;
```

```
    if(t0<0) n0 = 0.0;
    else {
      t0 *= t0;
      n0 = t0 * t0 * dot(grad3[gi0], x0, y0, z0);
    }
    float t1 = 0.6 - x1*x1 - y1*y1 - z1*z1;
    if(t1<0) n1 = 0.0;
    else {
      t1 *= t1;
      n1 = t1 * t1 * dot(grad3[gi1], x1, y1, z1);
    }
    float t2 = 0.6 - x2*x2 - y2*y2 - z2*z2;
    if(t2<0) n2 = 0.0;
    else {
      t2 *= t2;
      n2 = t2 * t2 * dot(grad3[gi2], x2, y2, z2);
    }
    float t3 = 0.6 - x3*x3 - y3*y3 - z3*z3;
    if(t3<0) n3 = 0.0;
    else {
      t3 *= t3;
      n3 = t3 * t3 * dot(grad3[gi3], x3, y3, z3);
    }
    // Add contributions from each corner to get the final noise value.
    // The result is scaled to stay just inside [-1,1]
    return 32.0*(n0 + n1 + n2 + n3);
  }

  // 4D simplex noise
  __device__ float noise(float x, float y, float z, float w)
  {
    // The skewing and unskewing factors are hairy again for the 4D case
    const float F4 = (sqrt(5.0)-1.0)/4.0;
    const float G4 = (5.0-sqrt(5.0))/20.0;

    float n0, n1, n2, n3, n4; // Noise contributions from the five corners

    // Skew the (x,y,z,w) space to determine which cell of 24 simplices we're in
    float s = (x + y + z + w) * F4; // Factor for 4D skewing
    int i = fastfloor(x + s);
    int j = fastfloor(y + s);
    int k = fastfloor(z + s);
    int l = fastfloor(w + s);

    float t = (i + j + k + l) * G4; // Factor for 4D unskewing

    float X0 = i - t; // Unskew the cell origin back to (x,y,z,w) space
```

```
float Y0 = j - t;
float Z0 = k - t;
float W0 = l - t;

float x0 = x - X0; // The x,y,z,w distances from the cell origin
float y0 = y - Y0;
float z0 = z - Z0;
float w0 = w - W0;

// For the 4D case, the simplex is a 4D shape I won't even try to describe.
// To find out which of the 24 possible simplices we're in, we need to
// determine the magnitude ordering of x0, y0, z0 and w0.
// The method below is a good way of finding the ordering of x,y,z,w and
// then find the correct traversal order for the simplex we...
// First, six pair-wise comparisons are performed between each possible pair
// of the four coordinates, and the results are used to add up binary bits
// for an integer index.
int c1 = (x0 > y0) ? 32 : 0;
int c2 = (x0 > z0) ? 16 : 0;
int c3 = (y0 > z0) ? 8 : 0;
int c4 = (x0 > w0) ? 4 : 0;
int c5 = (y0 > w0) ? 2 : 0;
int c6 = (z0 > w0) ? 1 : 0;
int c = c1 + c2 + c3 + c4 + c5 + c6;

int i1, j1, k1, l1; // The integer offsets for the second simplex corner
int i2, j2, k2, l2; // The integer offsets for the third simplex corner
int i3, j3, k3, l3; // The integer offsets for the fourth simplex corner

// simplex[c] is a 4-vector with the numbers 0, 1, 2 and 3 in some order.
// Many values of c will never occur, since e.g. x>y>z>w makes x<z, y<w and x<w
// impossible. Only the 24 indices which have non-zero entries make any sense.
// We use a thresholding to set the coordinates in turn from the largest magnitude.

// The number 3 in the "simplex" array is at the position of the largest coordinate.
i1 = simplex[c][0]>=3 ? 1 : 0;
j1 = simplex[c][1]>=3 ? 1 : 0;
k1 = simplex[c][2]>=3 ? 1 : 0;
l1 = simplex[c][3]>=3 ? 1 : 0;

// The number 2 in the "simplex" array is at the second largest coordinate.
i2 = simplex[c][0]>=2 ? 1 : 0;
j2 = simplex[c][1]>=2 ? 1 : 0;
k2 = simplex[c][2]>=2 ? 1 : 0;
l2 = simplex[c][3]>=2 ? 1 : 0;

// The number 1 in the "simplex" array is at the second smallest coordinate.
```

```
    i3 = simplex[c][0]>=1 ? 1 : 0;
    j3 = simplex[c][1]>=1 ? 1 : 0;
    k3 = simplex[c][2]>=1 ? 1 : 0;
    l3 = simplex[c][3]>=1 ? 1 : 0;

    // The fifth corner has all coordinate offsets = 1, so no need to look that up.
    float x1 = x0 - i1 + G4; // Offsets for second corner in (x,y,z,w) coords
    float y1 = y0 - j1 + G4;
    float z1 = z0 - k1 + G4;
    float w1 = w0 - l1 + G4;
    float x2 = x0 - i2 + 2.0*G4; // Offsets for third corner in (x,y,z,w) coords
    float y2 = y0 - j2 + 2.0*G4;
    float z2 = z0 - k2 + 2.0*G4;
    float w2 = w0 - l2 + 2.0*G4;
    float x3 = x0 - i3 + 3.0*G4; // Offsets for fourth corner in (x,y,z,w) coords
    float y3 = y0 - j3 + 3.0*G4;
    float z3 = z0 - k3 + 3.0*G4;
    float w3 = w0 - l3 + 3.0*G4;
    float x4 = x0 - 1.0 + 4.0*G4; // Offsets for last corner in (x,y,z,w) coords
    float y4 = y0 - 1.0 + 4.0*G4;
    float z4 = z0 - 1.0 + 4.0*G4;
    float w4 = w0 - 1.0 + 4.0*G4;

    // Work out the hashed gradient indices of the five simplex corners
    int ii = i & 255;
    int jj = j & 255;
    int kk = k & 255;
    int ll = l & 255;
    int gi0 = perm[ii+perm[jj+perm[kk+perm[ll]]]] % 32;
    int gi1 = perm[ii+i1+perm[jj+j1+perm[kk+k1+perm[ll+l1]]]] % 32;
    int gi2 = perm[ii+i2+perm[jj+j2+perm[kk+k2+perm[ll+l2]]]] % 32;
    int gi3 = perm[ii+i3+perm[jj+j3+perm[kk+k3+perm[ll+l3]]]] % 32;
    int gi4 = perm[ii+1+perm[jj+1+perm[kk+1+perm[ll+1]]]] % 32;

    // Calculate the contribution from the five corners
    float t0 = 0.6 - x0*x0 - y0*y0 - z0*z0 - w0*w0;
    if(t0<0) n0 = 0.0;
    else {
      t0 *= t0;
      n0 = t0 * t0 * dot(grad4[gi0], x0, y0, z0, w0);
    }
    float t1 = 0.6 - x1*x1 - y1*y1 - z1*z1 - w1*w1;
    if(t1<0) n1 = 0.0;
    else {
      t1 *= t1;
      n1 = t1 * t1 * dot(grad4[gi1], x1, y1, z1, w1);
    }
```

```
        float t2 = 0.6 - x2*x2 - y2*y2 - z2*z2 - w2*w2;
        if(t2<0) n2 = 0.0;
        else {
          t2 *= t2;
          n2 = t2 * t2 * dot(grad4[gi2], x2, y2, z2, w2);
        }
        float t3 = 0.6 - x3*x3 - y3*y3 - z3*z3 - w3*w3;
        if(t3<0) n3 = 0.0;
        else {
          t3 *= t3;
          n3 = t3 * t3 * dot(grad4[gi3], x3, y3, z3, w3);
        }
        float t4 = 0.6 - x4*x4 - y4*y4 - z4*z4 - w4*w4;
        if(t4<0) n4 = 0.0;
        else {
          t4 *= t4;
          n4 = t4 * t4 * dot(grad4[gi4], x4, y4, z4, w4);
        }
        // Sum up and scale the result to cover the range [-1,1]
        return 27.0 * (n0 + n1 + n2 + n3 + n4);
    }
}
```

A.2.1 Turbulence function

Listing A.14: Turbulence

```
#define FSIZE 7
__constant__ __device__ float turb_frequencies[FSIZE] = {0.05f, 2.0f, 4.0f, 6.0f, 12.0f, 8.0f};

__device__ float turbulence(float3 v)
{
    float sum = 0.0f;
    for(int i=0; i<FSIZE; i++)
    {
        float f = __powf(2.0, turb_frequencies[i]);
        sum += (SimplexCUDA::noise(v.x*f,v.y*f,v.z*f, (float)i)) / f ;
    }

    return sum;
}
```

A.3 OpenMP

A.3.1 OpenMP implementations

Listing A.15: Dust density generation function (OMP)

```
void generateVolumeOMP(float4* data, int xsize, int ysize, int zsize)
{
    float oneOverX = 1.0f / xsize;
    float oneOverY = 1.0f / ysize;
    float oneOverZ = 1.0f / zsize;
    int x =0, y=0, z=0;
    const float scale = 1000.0f;
    const float3 sunPos = make_float3(0.5f);
    const float3 noiseOffset = make_float3(0.5f);
    const float scale2 = 2.0f;
    int idx;
    #pragma omp parallel for default(none), \
                              private(x,y,z,idx), \
                              shared(xsize,ysize,zsize,data,oneOverX,oneOverY,oneOverZ)
    for (x = 0; x < xsize; x++ )
        for (y = 0; y < ysize; y++ )
            for(int z=0; z<zsize; z++)
            {
                idx = x*xsize*ysize + y*ysize + z;
                float3 here = make_float3(x*oneOverX, y*oneOverY, z*oneOverZ);
                float sun = cubic( length(here - sunPos), 0.01f);
                float sct = sun + max(0.0f, Simplex::turbulence(here + noiseOffset))*0.15f;
                float mag = sun;

                data[idx] = make_float4(mag, mag, mag,clamp(sct,0.0001f,1.0f));
            }
}
```

A.4 C# .NET

A.4.1 Example datatemplate showing databinding

```
<Expander Header="Sun" Foreground="{StaticResource FontColor}" IsExpanded="True">
 <StackPanel>
    <StackPanel Orientation="Horizontal">
```

```
            <TextBlock Style="{StaticResource TextBlockTitle}">Sun Phi (exp):</TextBlock>
            <TextBlock Text="{Binding SunPhi, Mode=OneWay, StringFormat=N2}" />
        </StackPanel>
        <Slider Minimum="-10.0" Maximum="10.0" Value="{Binding SunPhi, Mode=TwoWay}"/>
        <StackPanel Orientation="Horizontal">
            <TextBlock Style="{StaticResource TextBlockTitle}">Sun X:</TextBlock>
            <TextBlock Text="{Binding SunX, Mode=OneWay, StringFormat=N2}" />
        </StackPanel>
        <Slider Minimum="0.0" Maximum="1.0" Value="{Binding SunX, Mode=TwoWay}" />

        <StackPanel Orientation="Horizontal">
            <TextBlock Style="{StaticResource TextBlockTitle}">Sun Y:</TextBlock>
            <TextBlock Text="{Binding SunY, Mode=OneWay, StringFormat=N2}" />
        </StackPanel>
        <Slider Minimum="0.0" Maximum="1.0" Value="{Binding SunY, Mode=TwoWay}"/>

        <StackPanel Orientation="Horizontal">
            <TextBlock Style="{StaticResource TextBlockTitle}">Sun Z:</TextBlock>
            <TextBlock Text="{Binding SunZ, Mode=OneWay, StringFormat=N2}" />
        </StackPanel>
        <Slider Minimum="0.0" Maximum="1.0" Value="{Binding SunZ, Mode=TwoWay}"/>
    </StackPanel>
</Expander>
```

A.4.2 Procedural Nebula Texture

The following code segment is a class that is bound to xml as seen in the previous code listing. This class calls the nebula core C++ DLL and handles binding the results to OpenGL textures.

Listing A.16: CodeExample

```
namespace Nebula.Rendering.Textures
{
    public class ProceduralNebulaTexture : Texture
    {
        #region Fields
        private float _sunX;
        private float _sunY;
        private float _sunZ;
        private ICommand _generateCommand;
        private float _a;
        private float _g;
        private float _noiseOffsetX;
        private float _noiseOffsetY;
```

```
        private float _noiseOffsetZ;
        private string _message;
        private float _filterRadius;
        private float _sunPhi;
        private float _densityScale;

        #endregion

        #region CTors

        public ProceduralNebulaTexture()
        {
            A = 0.6f;
            G = 0.6f;
            FilterRadius = 0.9f;
            SunX = 0.5f;
            SunY = 0.5f;
            SunZ = 0.5f;
            DensityScale = 0.015f;
        }

        #endregion

        #region Properties

        public float DensityScale
        {
            get { return _densityScale; }
            set { _densityScale = value; OnPropertyChanged("DensityScale"); }

        }


        public float FilterRadius
        {
            get { return _filterRadius; }
            set { _filterRadius = value; OnPropertyChanged("FilterRadius"); }

        }


        public float SunPhi
        {
            get { return _sunPhi; }
            set
            {
                if (value == _sunPhi)
```

```
                    return;

                _sunPhi = value; OnPropertyChanged("SunPhi");
            }
        }

        public float SunX
        {
            get { return _sunX; }
            set
            {
                if (value == _sunX)
                    return;

                _sunX = value;OnPropertyChanged("SunX");
            }
        }

        public float SunY
        {
            get { return _sunY; }
            set
            {
                if (value == _sunY)
                    return;

                _sunY = value; OnPropertyChanged("SunY");
            }
        }


        public float SunZ
        {
            get { return _sunZ; }
            set
            {
                if (value == _sunZ)
                    return;

                _sunZ = value; OnPropertyChanged("SunZ");
            }
        }

        public string Message
        {
            get { return _message; }
            set { _message = value; OnPropertyChanged("Message"); }
```

```
        }

        public float NoiseOffsetZ
        {
            get { return _noiseOffsetZ; }
            set { _noiseOffsetZ = value; OnPropertyChanged("NoiseOffsetZ");}
        }

        public float NoiseOffsetY
        {
            get { return _noiseOffsetY; }
            set{ _noiseOffsetY = value; OnPropertyChanged("NoiseOffsetY"); }
        }

        public float NoiseOffsetX
        {
            get { return _noiseOffsetX; }
            set { _noiseOffsetX = value; OnPropertyChanged("NoiseOffsetX"); }
        }

        public float G
        {
            get { return _g; }
            set { _g = value; OnPropertyChanged("G"); }
        }

        public float A
        {
            get { return _a; }
            set { _a = value; OnPropertyChanged("A"); }
        }


        public ICommand GenerateCommand
        {
            get
            {
                if (_generateCommand == null)
                    _generateCommand = new DelegateCommand(n => Generate(),
                                                           n => CanGenerate());
                return _generateCommand;
            }
        }

        #endregion

```

```
        #region Functions
        protected override System.Drawing.Bitmap GeneratePreview()
        {
            return null;
        }

        public override void Initialize()
        {
            _textureID = GL.GenTexture();
        }

        public bool CanGenerate()
        {
            return !_backgroundWorker.IsBusy;
        }

        public void Generate()
        {
            GL.Enable(EnableCap.Texture3DExt);
            GL.PixelStore(PixelStoreParameter.UnpackAlignment, 1);
            GL.BindTexture(TextureTarget.Texture3D, _textureID);

            Mouse.SetCursor(Cursors.Wait);
            Stopwatch t = new Stopwatch();
            t.Start();
            NebulaCore.generateVolume(SunX, SunY, SunZ, (float) Math.Exp(SunPhi),
                                      A, G, NoiseOffsetX,
                                      NoiseOffsetY, NoiseOffsetZ, FilterRadius,
                                      DensityScale);
            t.Stop();
            Message = string.Format("Computed in {0} msec.", t.ElapsedMilliseconds);
            Mouse.SetCursor(Cursors.Arrow);
        }

        #region Implementation of Texture

        public override void Enable()
        {
            if (TextureID == 0)
                throw new Exception("Texture has not been initialized!");
            GL.Enable(EnableCap.Texture3DExt);
            GL.BindTexture(TextureTarget.Texture3D, TextureID);
        }

        public override void Disable()
        {
            GL.Disable(EnableCap.Texture3DExt);
```

```
            }

            #endregion

            #endregion
        }
}
```

A.4.3 Texture base-class

Listing A.17: Texture base

```
namespace Nebula.Lib.Graphics.OpenGL
{
    public abstract class Texture : INotifyPropertyChanged
    {

        protected int _textureID;
        private string _name;
        protected Bitmap _preview;

        public virtual int TextureID
        {
            get
            {
                if (_textureID == 0)
                    Initialize();
                return _textureID;
            }
        }

        public string Name
        {
            get { return _name; }
            set { _name = value; OnPropertyChanged("Name"); }
        }

        public abstract void Enable();
        public abstract void Disable();

        public abstract void Initialize();


        public Bitmap PreviewImage
```

```
        {
            get
            {
                if (_preview == null)
                {
                    _preview = GeneratePreview();
                    OnPropertyChanged("PreviewImage");
                }

                return _preview;
            }
        }

        protected abstract Bitmap GeneratePreview();

        #region INotifyPropertyChanged
        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged(string name)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(name));

        }
        #endregion
    }
}
```

A.5 Shaders

A.5.1 CG Shader for volume ray marching

```
struct VSData
{
    float4 Position : POSITION;
    float4 Normal : NORMAL;
    float4 Color : COLOR0;
    float4 TexCoord : TEXCOORD0;
};

struct FSData
{
    float4 Position : POSITION;
    float4 Color : COLOR0;
    float4 TexCoord : TEXCOORD0;
```

```
    float4 PosMVP : TEXCOORD5;
    float4 FragCoord : WPOS;
    float3 Eye : TEXCOORD4;

};



sampler2D BackfaceTexture;


sampler3D VolumeTexture = sampler_state {
    MinFilter = Linear;
    MagFilter = Linear;
    WrapS = Clamp;
    WrapT = Clamp;
    WrapR = Clamp;
};

samplerCube SkyboxTexture = sampler_state {
    MinFilter = Linear;
    MagFilter = Linear;
    WrapS = ClampToEdge;
    WrapT = ClampToEdge;
    WrapR = ClampToEdge;
};


float3 Eye;
float3 LookAt;
float3 Up;

float Threshold;
int Steps;
bool AlphaOnly;
bool EmptyBackground;
int OutputType;


// http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm
int intersectBox(float4 r_o, float4 r_d, float4 boxmin, float4 boxmax,
                 out float tnear, out float tfar)
{
    // compute intersection of ray with all six bbox planes
    float4 invR = float4(1.0) / r_d;
    float4 tbot = invR * (boxmin - r_o);
    float4 ttop = invR * (boxmax - r_o);
```

```
    // re-order intersections to find smallest and largest on each axis
    float4 tmin = min(ttop, tbot);
    float4 tmax = max(ttop, tbot);

    // find the largest tmin and the smallest tmax
    float largest_tmin = max(max(tmin.x, tmin.y), max(tmin.x, tmin.z));
    float smallest_tmax = min(min(tmax.x, tmax.y), min(tmax.x, tmax.z));

    tnear = largest_tmin;
    tfar = smallest_tmax;

    return smallest_tmax > largest_tmin;
}


float random(float2 co)
{
    return fract(sin(dot(co.xy ,float2(12.9898,78.233))) * 43758.5453);
}

FSData mainVS(VSData IN)
{
    FSData OUT;
    OUT.Position = mul(glstate.matrix.mvp, IN.Position);
    OUT.PosMVP = OUT.Position;

    OUT.TexCoord = IN.TexCoord;
    OUT.Color = IN.Color;
    OUT.Eye = Eye;
    return OUT;
}

float4 mainFS(FSData IN) : COLOR
{
    float2 fragCoord = (IN.PosMVP.xy / IN.PosMVP.w);
    float rnd = random(fragCoord);


    //calculate camera vectors
    float3 vp_normal = normalize(LookAt-Eye);
    float3 vp_axisX = normalize(cross(vp_normal, Up));
    float3 vp_axisY = normalize(cross(vp_axisX, vp_normal));

    float3 r_dir = normalize(vp_normal + vp_axisX*fragCoord.x + vp_axisY*fragCoord.y);
    float3 r_orig = Eye;
    const float4 boxMin = float4(-1.0f, -1.0f, -1.0f,0.0f);
```

```
    const float4 boxMax = float4( 1.0f, 1.0f, 1.0f, 0.0f);

    float4 background = EmptyBackground ? float4(0.0f, 0.0f, 0.0f, 1.0f)
                                        : float4(texCUBE(SkyboxTexture, r_dir).rgb, 1.0f);

   float tnear=0.0;
   float tfar=0.0;

   //This is only false when tnear ≠tfar as this shader only applies to the box geometry
   if(intersectBox(float4(r_orig,1.0f), float4(r_dir,1.0f),
                   boxMin, boxMax, tnear, tfar) ≤0)
       return background;

   //if we are inside the volume, tnear is 0 so starting point will be the eye
   tnear = max(0.0f, tnear);
   tnear += 0.1f * rnd * Threshold;
   float3 col_acc = float3(0.0f);
   float alpha_acc = 0.0f;
   float4 color_sample;
   float alpha_sample;



   // march along ray from back to front, accumulating color
    float t = tnear;
   float tstep = 0.01f;
   const int maxSteps = 250;
   for(uint i=0; i<maxSteps; i++)
   {
       float3 pos = r_orig + r_dir*t;
       pos = pos*0.5 + 0.5; // map position to [0, 1] coordinates

       color_sample = (tex3D(VolumeTexture,pos));
       alpha_sample = color_sample.a;

       //col_acc = lerp(col_acc, color_sample,alpha_sample);
       col_acc += color_sample.xyz * color_sample.a;
       alpha_acc += alpha_sample;
       t += tstep;
       if(t > tfar || alpha_acc ≥1.0)
           break;

   }

   if(OutputType == 1)
       return float4((r_orig + r_dir*tnear)*0.5 + 0.5 , 1.0);
   if(OutputType == 2)
```

```
        return float4((r_orig + r_dir*tfar)*0.5 + 0.5, 1.0);

    return alpha_acc ≥1.0f ?
            float4(col_acc.xyz, 1.0) :
            lerp(float4(col_acc.xyz, 1.0), background, 1.0f − alpha_acc);
}



technique technique0
{
    pass pre
    {
        CullFaceEnable = true;
        FrontFace = CW;
        VertexProgram = compile vp40 mainVS();
        FragmentProgram = compile fp40 mainFS();
    }

}
```

Bibliography

ANDERSEN, A. C. 2007. Dust from AGB Stars. In *Why Galaxies Care About AGB Stars: Their Importance as Actors and Probes*, F. Kerschbaum, C. Charbonnel, & R. F. Wing, Ed., vol. 378 of *Astronomical Society of the Pacific Conference Series*, 170–+.

ARS TECHNICA, 2008. Gaming expected to be a $68 billion business by 2012. `http://arstechnica.com/gaming/news/2008/06/gaming-expected-to-be-a-68-billion-business-by-2012.ars`, June.

BARKER, E., BARKER, W., BURR, W., POLK, W., AND SMID, M. 2005. Recommendation for key management - part 1: General.

BOHREN, C. F., AND HUFFMAN, D. R. 1983. *Absorption and scattering of light by small particles.* John Wiley and Sons.

COMPUTERWORLD, 2008. Thanks to gamers, the desktop supercomputer arrives. `http://www.computerworld.com/s/article/9120741/Thanks_to_gamers_the_desktop_supercomputer_arrives`, November.

FOSTER, I. 1995. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

FRISVAD, J. R., AND WYVILL, G. 2007. Fast high-quality noise. In *Proceedings of GRAPHITE 2007*, ACM, 243–248+315.

GORDON, K. D. 2004. Interstellar dust scattering properties. *Astrophysics of Dust, ASP Conference Series 309*, 77–91.

GUSTAVSON, S. 2005. Simplex noise demystified. Tech. rep., Linkoping University, Sweden (stegu@itn.liu.se), March.

GUTIERREZ, D., JENSEN, H. W., JAROSZ, W., AND DONNER, C. 2009. Scattering. In *SIGGRAPH ASIA '09: ACM SIGGRAPH ASIA 2009 Courses*, ACM, New York, NY, USA, 1–620.

HENYEY, L. G., AND GREENSTEIN, J. L. 1938. The Theory of the Colors of Reflection Nebulae. *Astrophysical Journal 88* (Dec.), 580–+.

HENYEY, L. G., AND GREENSTEIN, J. L. 1940. Diffuse radiation in the Galaxy. *Astrophysical Journal 3* (Jan.), 117–+.

HOWES, L., AND THOMAS, D. 2007. *Gpu gems 3*. Addison-Wesley Professional.

JENSEN, H. W., 2000. Images demonstrating the use of photon mapping. `http://graphics.ucsd.edu/~henrik/images/`.

JENSEN, H. W. 2001. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA.

JUN, B., AND KOCHER, P., 1999. The intel random number generator. White paper, Cryptography Research Inc., April.

KAY, T. L., AND KAJIYA, J. T. 1986. Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 269–278.

LEVOY, M. 1990. Efficient ray tracing of volume data. *ACM Trans. Graph. 9*, 3, 245–261.

LINTU, A., HOFFMAN, L., MAGNOR, M. A., LENSCH, H. P. A., AND SEIDEL, H.-P. 2007. 3d reconstruction of reflection nebulae from a single image. In *VMV*, Aka GmbH, H. P. A. Lensch, B. Rosenhahn, H.-P. Seidel, P. Slusallek, and J. Weickert, Eds., 109–116.

MAGNOR, M. A., HILDEBRAND, K., AND INFORMATIK, M. 2005. Reflection nebula visualization. In *In Proc.of IEEE Visualization*, IEEE, 255–262.

NASA, H. S. T. C., 2000. The reflection nebula in orion. `http://www.nasaimages.org/luna/servlet/detail/nasaNAS~5~5~23359~127347:The-Reflection-Nebula-in-Orion`.

NVIDIA CORPORATION, 2009. Nvidia cuda c programming best practices guide. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf`, July.

PERLIN, K. 1985. An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3, 287–296.

Pharr, M., and Humphreys, G. 2004. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Sen, S.K., S. T., and Reese, A. 2006. Quasi- versus pseudo-random generators: Discrepancy, complexity and integration-error based comparison. *International Journal of Innovative Computing, Information and Control 2*, 3, 621–651.