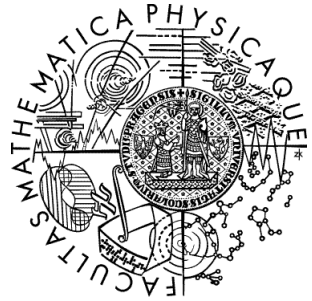
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Oskár Elek

Rendering Planetary Atmospheres in Real-Time

Department of Software and Computer Science Education

Thesis supervisor: Mgr. Petr Kmoch
Study program: Computer Science, Programming

2008

Here I would like to thank my supervisor, Mgr. Petr Kmoch, who aided me with picking the topic for my work and also advised me during the creation of both application and thesis. Moreover, I would like to thank my dear girlfriend, who helped me to overcome stress involved in the elaboration.

I declare that I have written my bachelor thesis independently and solely by using cited sources. I agree with lending of the thesis and its publishing.

In Prague, 05/30/2008 Oskár Elek

Contents

Název práce: Vykreslování planetárních atmosfér v reálním čase
Autor: Oskár Elek
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: Mgr. Petr Kmoch
e-mail vedoucího: petr.kmoch@mff.cuni.cz

Abstrakt: V oblasti fotorealistického vykreslování fyzikálních jevů hraje renderování atmosférického rozptylu světla velice důležitou roli. Vykreslování oblohy a atmosféry obecně je nezbytné pro většinu her, různých simulátorů, virtuálních světů či dokonce i animovaných filmů. Je to sice velice těžká úloha, ale díky rozvoji specializovaného počítačového hardware je dnes již zvládnutelná. V mé bakalářské práci představuji přesnou a zároveň rychlou metodu pro zobrazování planetárních atmosfér. Toho je dosaženo předpočítáním složitých rovnic primárního rozptylu světla do série vyhledávacích tabulek. Správná barva atmosféry je z nich poté vyzdvižena ve fragment shaderu. Prezentovaná metoda je implementována v počítačovém programu, který je schopen vykreslovat realistickou atmosféru rychlostí několika set snímků za sekundu.
Klíčová slova: fotorealistické zobrazování, atmosférický rozptyl světla, programování GPU, procedurální textury

Title: Rendering Planetary Atmospheres in Real-Time
Author: Oskár Elek
Department: Department of software and computer science education
Supervisor: Mgr. Petr Kmoch
Supervisor's e-mail address: petr.kmoch@mff.cuni.cz

Abstract: In the field of photorealistic rendering of physical phenomena, the rendering of atmospheric light scattering takes a very important place. Real-time rendering of sky and atmosphere in general is essential for all outdoor computer games, various simulators, virtual worlds or even for animated movies. It is a very difficult task, but thanks to the advancement of dedicated graphics hardware we can reach it today. In my thesis I present an accurate and fast method for real-time rendering of planetary atmospheres. This is achieved by precomputing complex single-scattering equations into a set of lookup tables. The correct atmospheric colour values are then fetched from these in the fragment shader. The presented method is then implemented in a program that is capable of rendering realistic atmosphere in hundreds of FPS.
Keywords: photorealistic rendering, atmospheric light scattering, GPU programming, procedural textures

Chapter 1

Introduction

One of the major reasons for founding 3D computer graphics was the desire for accurate reproduction of physical reality that surrounds us. Till today, 3D graphics diverged into many different directions, but the one which strives for the original purpose of this domain — photorealistic rendering — is perhaps the most important of them all. One of the most interesting parts of protorealistic rendering is, in my opinion, rendering of various physical phenomena. This work aims at rendering one such phenomenon, which is very well known to everyone, as it accompanies us in our everyday life — the atmospheric scattering responsible for light conditions which we live in.

1.1 Motivation — who needs realistic atmosphere?

Atmosphere is the layer of gases, aerosols and condensation cores that surrounds our planet. It is not very dense, but it still influences the manner in which light from the Sun propagates before it reaches the ground and also our retinas. It is responsible for perceived colour of the sky during the day or night (see Figure 1.1). Because of this, every application that renders any outdoor scene should care about beliveable appearance of the sky and also sunlight colour in certain time of day. Modern computer games, such as 3D shooters, flight or spacecraft simulators, but also various virtual worlds, strongly need to persuade their user about realism of the environment they are moving in. Moreover, training simulators for aircraft or spacecraft pilots have to create immersion of controlling the real machine in real conditions. Or for instance, some TV sessions such as weather forecasts may also need realistic atmosphere.

The key problem in these applications is that they have to immediately react for example to the user's movement, so all parts of the environment, including the atmosphere, have to be rendered in real-time. As we will se in forthcoming chapters, this is quite difficult to achieve.

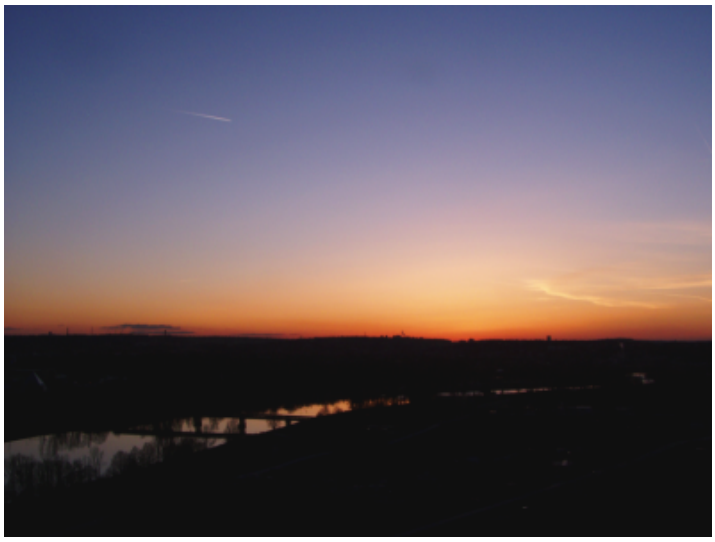


Figure 1.1: Real photo of an evening sky

1.2 Goals of the thesis

My work aims to reach these objectives:

1. Examine existing sources and methods which aim for real-time rendering of the atmosphere or other light scattering effects.

2. Pick one such method, implement it, analyze it and figure out the new method based on it, but with modifications allowing custom parameterization of both the rendered atmosphere and underlying planet and any corrections that I consider necessary.

3. To implement the new method in my own program with emphasis on strong GPU utilization and low CPU load.

Chapter 2

Atmosphere rendering

2.1 Physical basis of light scattering

First of all, I'd like to very shortly describe the physical mechanisms that run in the atmosphere. I think that their knowledge can help to better understand mathematical principles used for its visualization.

Earth's atmosphere is a very complex object. It is generally a mixture consisting of many chemical compounds. However, these can be divided into three groups: gases, aerosols and solid particles, such as dust or tiny ice crystals. In the climatological sense, the atmosphere is described by tens of properties — temperature, pressure, density, humidity and so on. However, these properties mainly describe the percentages, distribution and movement of these three basic types of compounds in the atmosphere.

This is important, because these compounds are directly involved in the influencing of the light propagation through the atmosphere. How? When a particular photon from the Sun (or any light source) hits a gas molecule or an aerosol particle, it will likely change the direction of movement (in fact, it may also be absorbed by it or another photon may be emitted from it, but I'll omit this process, as it belongs to quantum physics and thermodynamics). This phenomenon is called *light scattering* and is responsible for all characteristic colours of the sky. We call the compounds causing it a *participating media*.

We basically recognize two types of light scattering — Rayleigh and Mie scattering. Rayleigh scattering is the light scattering on gas molecules. These are of size comparable to the wavelength of visible light, so the shorter the wavelength of an incoming photon is, the more probably it gets scattered. This dependency is the main reason for the blue sky colour during the day and the reddish hues during the sunrise or sunset. Mie scattering is on the other hand the light scattering on aerosol and dust particles. As these are vastly bigger than visible light wavelength, the scattering on them is wavelength independent.

And here comes the main issue: a particular photon can hit and bounce from atmospheric particles many times until it reaches an observer[1]. It is called *multiple scattering*. This results in high-degree polynomial complexity, when one wants to simulate this behaviour. So we need a simplified mathematical model that is possible to implement algorithmically.

2.2 Statistical versus empirical approach

In designing a mathematical model that will be sufficently accurate for describing atmosphere colour and simple enough to be implemented in code, two different approaches can be taken: statistical and empirical.

Statistical - Thanks to gravitational force of the planetary body, the atmospheric mass is distributed unevenly in the sense of altitude. More precisely, about 50% of atmosphere's mass is located under 5.6km, 75% under 11.2km and so on. That means the atmospheric pressure halves each 5.6km [10]. Thanks to this knowledge, it is possible to create a probabilistic *distribution function* based on real physical principles. This is the preferred approach, as it is physically correct, despite the simplifications which have to be made. On the other hand, it can be very time-consuming, as we'll see in Section 3.1, where such model called *single-scattering* is described.

Empirical - This approach means that we don't try to create a physically based model, but we analyze desired behaviour (in this case the colour of the sky) and try to produce an arbitrary equation or algorithm that provides us results which are at least close to the desired ones. This *ad hoc* solution can be much faster than the previous one, but it is very difficult to find such empirical method to describe as complex an object as atmosphere. Moreover, such solution doesn't help to deeply understand any laws taking place in the particular phenomenon, it only tries to imitate the results. For this reason I don't consider this approach very scientific, so I've decided to follow the statistical approach everytime it will be possible, because it shows that despite its potential complexity, it can already be implemented in real-time.

2.3 Related work on atmospheric light scattering

There are many works dealing with general light scattering in various environments. Of course these principles are applicable also on atmospheric

[1]However, it loses some of its energy after each bounce, so the photon becomes infrared after a certain amount of bounces (but that can be quite high)

scattering, but dealing with it adds issues that are specific only to the atmosphere, such as its chemical composition, density fluctuation or largeness. For this reason I only focused on works about atmospheric light scattering. In spite of this there's still quite a lot of them, so I don't try to create their complete enumeration. Rather I present a list of works that influenced development of my method itself.

In agreement with advantages and disadvantages of two possible approaches described in Section 2.2, the physically-based statistical approach is more frequent in endeavour for getting realistic atmosphere look. This is mainly possible thanks to very good knowledge of physical laws involved in light scattering and their algorithmic implementability. In fact all works I'm going to present are using basically the same *single-scattering* model (see Section 3.1 for details and mathematical equations). Then the main difference between them is not in the mathematical model they implement, but in the algorithmical approach they take for its implementation.

Display of The Earth Taking into Account Atmospheric Scattering - This I'd say almost classical paper of a Japanese research group from 1993 [5] is one of the first works in modern computer graphics that realistically deals with atmospheric scattering, at least as far as I'm aware. Nishita et al. presented here a single-scattering model suitable for algorithmic implementation. This model later became popular in modelling atmospheric scattering. As the word 'Display' from the title suggests, their algorithm was not capable of interactive rendering of the Earth's atmosphere[2] — they implemented a software ray-tracing that rendered one frame in several minutes. Although their scattering model was capable of calculating the atmosphere colour from any viewpoint under or above the upper atmosphere boundary, their algorithm was designed to display the realistic atmosphere only from outside of it (see Figure 2.1 for an example). They also designed a *pre-computed lookup table* for speeding up the rendering process a bit. In this 2D lookup table, they stored the *out-scattering* (see Section 3.1 for definition) from the Sun to a sample point in arbitrary height, by taking advantage of the fact that the Sun is so far away that any two light rays from it can be considered parallel.

Real-Time Atmospheric Scattering - In 2004, Sean O'Neil [6] implemented a real-time CPU based, vertex oriented algorithm by using the same scattering model presented by Nishita et al. [5]. He mainly improved the lookup table introduced by them, so he was able to store not only the out-scattering from the Sun, but also the out-scattering to the observer. However he hasn't succeeded in precomputing the

[2]Bear in mind it was the year 1993

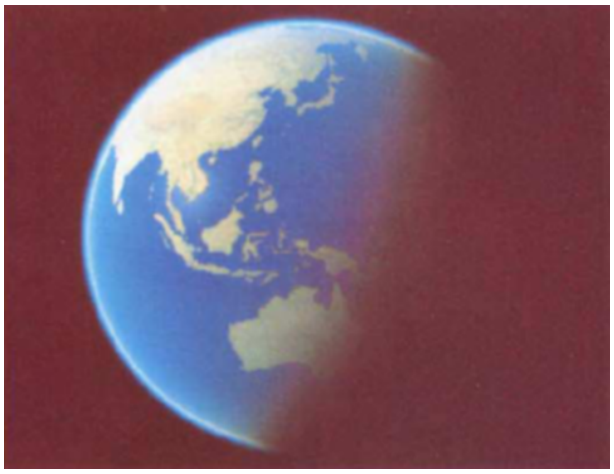


Figure 2.1: Ray-traced Earth (from Nishita et al. [5])

in-scattering, so he could afford only few integral samples for its computation, to keep the whole rendering process in real-time framerates. This, in combination with per-vertex calculations, resulted in quite rough accuracy. He was however aware that there must be a way to precompute both out- and in- scattering into a 3D lookup table, so he suggested further investigation of this possibility.

Accurate Atmospheric Scattering - A year later, in 2005, the same Sean O'Neil ported [7] his real-time solution onto the dedicated graphics hardware. This was an important deed as there was little perspective of improving the accuracy of his previous CPU version. However he still wasn't successful in precomputing both scattering integrals into any form of lookup texture, so he implemented the in-scattering integral in a vertex shader, as its real-time computation was too expensive for a fragment shader. He didn't want his implementation to require the Shader Model 3.0, so he didn't used the improved lookup texture from his previous implementation, because the *vertex texture* reads were expensive on the GeForce 6 series (according to Gerasimov et al. [4]). Instead he figured an analytical expression of both out-scattering integrals, by which he made a purely virtual algorithm without the need of any precomputed data.

Real-Time Rendering of Planets with Atmospheres - Finally in 2007, Schafhitzel et al. presented [8] an approach that already computed the atmospheric colour in a fragment shader. This was achieved by figuring out the paramaterization for the *3D lookup texture* (predicted by O'Neil [6]) — it was parameterized by camera height, angle to sun and angle to camera. Only coordinates (based on actual spatial parame-

ters) into this texture and several corrections had to be calculated in a fragment shader. This approach, as expected, was very fast, but the lookup table lacked one dimension — the azimuth from the Sun. This resulted in some incorrect colours mainly during the sunset and the inability to correctly display Mie scattering. I'll describe this method in the next chapter, as it forms the starting point for my work.

Chapter 3

Rendering of planetary atmospheres using precomputed 3D lookup table

In this chapter I'm going to present a method for real-time rendering of planetary atmospheres. Its mathematical model is adapted from [5] and the algorithmical part is based on [8]. However, I've made several improvements of this method, as it didn't produce correct results under certain conditions, which I'll describe later.

3.1 Mathematics of the single-scattering model

In this section I'm going to present the mathemtatical model of *single scattering* (hereinafter *SS*). SS is a simplified physical model in the sense that it omits all of a photon's multiple bounces in the participating medium. It calculates with only one deflection of the photon away from its original direction, so the light flux gets attenuated (out-scattering) and with the one deflection of photon flying in the arbitrary direction into our view path (in-scattering). Let's now discuss the statictical equations describing SS.

As explained in 2.1, we recognize two types of atmospheric light scattering — Rayleigh scattering and Mie scattering. Since their calculation is similar to each other, I'll first describe Rayleigh scattering and then I'll discuss differences between them.

The amount of light I_s scattered in sample point P in dependence on spectral wavelength λ and scattering angle θ (see Figure 3.1) is described by the Rayleigh scattering equation:

$$I_s(\lambda, \theta) = I_i(\lambda) K \rho(h) F_R(\theta) \frac{1}{\lambda^4} \tag{3.1}$$

where I_i is the intensity of incident spectral wavelength λ, $K = \frac{2\pi^2(n^2-1)^2}{3N_S}$ is the ratio for molecular density of atmosphere at sea level, n is index of

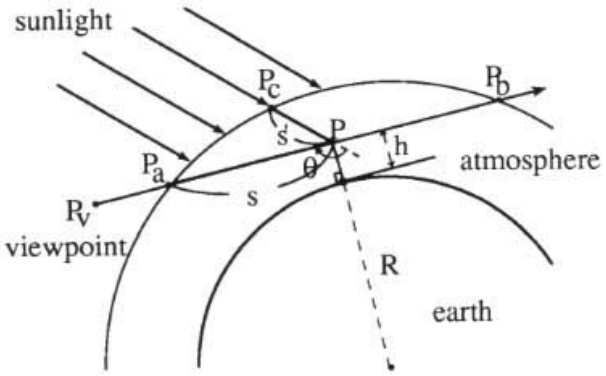


Figure 3.1: Schematic view of the atmosphere (redrawn from Nishita et al. [5])

refraction for air, N_S is the absolute molecular density at sea level, $\rho(h) = exp(-\frac{h}{H_R})$ stands for the density function (in dependence on height h of sample point P and Rayleight scale height[1] $H_R = 7994m$) and F_R is the Rayleigh phase function (for the complete list of used constants' values see Appendix C).

F_R represents the amount of scattered light in dependence on scattering angle θ between light ray and viewing ray. The standard Rayleigh phase function is denoted by $F_R(\theta) = \frac{3}{4}(1 + cos^2(\theta))$. Application of this function produces the largest amount of scattered light at 0° and 180° scattering angles and the smallest amount at 90° scattering angle (exactly half of 0°/180°). This is the correct property of SS, but results produced by this form of F_R do not match the real-world observations. This is caused by the fact that we are working with only SS whereas in the real observations, multiple scattering takes place. To resolve this problem I've derived an empirical Rayleigh phase function defined as

$$F_R(\theta) = \frac{7}{10}\left(\frac{7}{5} + \frac{1}{2}cos(\theta)\right) \tag{3.2}$$

This function produces the smallest values at 180° which gradually increase up to 0°, where the amount of scattered light is largest. There are approaches which solve this problem by adding an ambient term to the overall sky colour, but I find this kind of correction more appropriate — similarly to the SS, the amount of multiple-scattered light varies according to view angle and also daytime and thus cannot be approximated by a constant ambient term.

Let's now describe the calculation of *optical depth*, which is the attenuation coefficient for the light ray between two points in atmosphere due to

[1]Represents the altitude where an atmospheric pressure of Rayleigh-active molecules is halved

the out-scattering. Given the attenuation coefficient $\beta(\lambda) = \frac{4\pi K}{\lambda^4}$, the optical depth $t(S, \lambda)$ along path S is obtained by integrating $\beta(\lambda)$ weighted by the density function $\rho(h)$, that is

$$t(S, \lambda) = \int_0^S \beta(\lambda)\rho(s)ds \equiv \frac{4\pi K}{\lambda^4} \int_0^S \rho(s)ds \tag{3.3}$$

where the last equivalence is valid thanks to the assumption that the index of refraction for air n does not change with altitude (truly there's a small variation, but I neglect it for sake of simplicity).

Knowing the amount of light attenuated along some light path, we can finally obtain the amount of light that the observer situated in P_V receives. We achieve this by combining Equation 3.3 with Equation 3.1. The Rayleigh intensity I_V of spectral wavelength λ in P_V is then denoted by

$$I_V(\lambda) = I_i(\lambda)F_R(\theta)\frac{K}{\lambda^4}\int_{P_a}^{P_b} \rho(h)exp\left(-t(PP_c, \lambda) - t(PP_a, \lambda)\right)ds \tag{3.4}$$

where h is the height of sample point P.

Having defined the intensity of Rayleigh-scattered light, I can now describe the modifications needed to calculate Mie scattering. The whole system of equations stays, except for the $\frac{1}{\lambda^4}$ term, that has to be completely removed. This is caused by the nature of Mie scattering — the aerosol particles causing it are vastly larger than the wavelength of visible light, so this no longer influences the intensity of scattered light.

We also have to use a different phase function. An often-used Henyey-Greenstein function has been improved by Cornette [2] to a physically more reasonable formulation of F_M:

$$F_M(\theta, g) = \frac{3(1-g^2)}{2(2+g^2)}\frac{(1+cos^2(\theta))}{(1+g^2-2gcos(\theta))^{3/2}} \tag{3.5}$$

where θ is the scattering angle and g stands for the asymetry factor between forward and backward scattering and $g \in (-1; -0.75\rangle$. It can be noticed that setting g to 0 gives us the original Rayleigh phase function.

To conclude, calculation of Mie scattering also requires modification of a few parameters. A different scale height for the density function must be used, as the aerosol and dust particles are located in much lower altitudes of the atmosphere due to their greater weight in comparasion with gas molecules. Thus the Mie scale height used is $H_M = 1200$m. Also the different molecular density N_S has to be used, because the absolute density of aerosol particles is much lower than the density of gas molecules.

3.2 The concept of precomputation

One of the most massively used concepts in computer graphics and especially in real-time graphics is *precomputation*. Precomputation is the typical tradeoff 'memory for speed' often seen in computer sciences. Suppose

we have some complicated function or integral equation (just like our SS model) which we need to evaluate for instance per-fragment to compute local lighting or any other property of particular graphical primitive (let's call it *critical function*, *CF*). This assumption implicates that we have to evaluate CF say a few million times to obtain one single frame. For the SS model, this is equivalent to several minutes of renderig.

Precomputation is the process when the CF is adequately sampled and the resulting values are stored in some data structure, from where they are fetched during the rendering[2]. The possibility of precomputation depends on two issues: dimensionality of CF and the desired precision. The first issue exist because it's quite difficult (and also memory consuming) to store values of function with dimension larger than 3 in the data structures common in today's real-time graphics — *lookup textures*[3]. The precision issue arises because the desired sampling frequency of CF may be larger than the biggest allowed size of textures or sampling of CF in this frequency could cause the size of the precomputed data to be unbearably large.

The importance of precomputation increased even more with coming of programmable graphics hardware. This is so because on modern GPUs the speed of a fragment texture fetch operation is extremely high. It is even reasonable to precompute a normalization cubemap and fetch the normalized version of vector instead of normalizing it directly in fragment shader [3]. Thus it is very convenient to figure how to precompute the SS model into such lookup texture. However as we'll see in the following section, it's not an easy task.

3.3 Precomputation of single scattering

Let's now discuss the possibility of precomputing the SS equations into a lookup table. Since I'm focusing on pure-GPU implementation (but precomputation can be handled by CPU, as this is a one-time batch operation), it's necessary to keep the dimensionality of a lookup table less or equal to 3, because higher-dimensional textures are not supported by hardware.

To completely precalculate the SS model, we basically have to evaluate Equation 3.4 for every possible observer position in the atmosphere looking in an arbitrary direction at arbitrary daytime (what means considering every possible sun direction). These are 3 parameters, but let's not forget that each of them is a 3-dimensional vector, what leaves us with 9(!) *degrees of freedom* (*DoF*). Even representing these vectors in polar coordinates leaves us with 6 DoF. This is then not the way to go. However, we can afford a few

[2]Of course we assume that the fetching operation lasts significantly shorter than calculation of CF

[3]A lookup texture is an arbitrary array of 1D, 2D or 3D data stored in primary or secondary memory

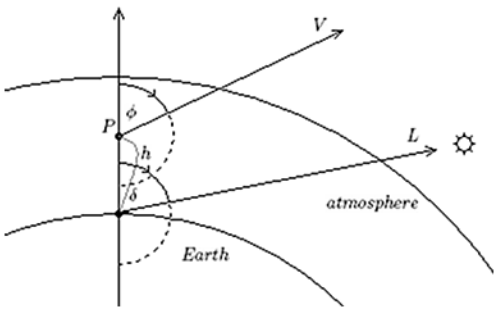


Figure 3.2: Scheme depicting $h\phi\delta$-parameterization

assumptions that will help us to achieve our intention:

(1) the Sun (light source) is so far away that all light rays from it can be considered parallel

(2) the Earth (planet) is perfectly spherical (neglecting terrain morphology for now)

(3) the density of an atmosphere changes according to altitude, but not according to latitude and longitude (this fits well with gaseous componds of atmosphere, but for aerosols there's a variation according to latitude that must be neglected)

Under these circumstances I adopted the parameterization proposed by Schafhitzel et al. [8]. This approach precomputes the scattering Equation 3.4 into a 3D lookup texture. This texture is parametrized by the altitude of the observer in the atmosphere $h \in \langle 0, H_{TOP} \rangle$ (where H_{TOP} is the altitude of atmosphere's upper boundary), the angle $\phi \in \langle 0, \pi \rangle$ between observer's position (in respect to the center of planetary sphere) and view direction and by the angle $\delta \in \langle 0, \pi \rangle$ between observer's position and light direction (see Figure 3.2). Thanks to the assumption (3) the position of observer in 3D space can be expressed without loss of generality as $\vec{P} = (0, h, 0)$. Since P lies on the y-axis, the view and light directions can be then expressed as $\vec{V} = (sin(\phi), cos(\phi), 0)$ and $\vec{L} = (sin(\delta), cos(\delta), 0)$, respectively. We can see that by using only 3 scalar values it's possible to express observer's position and both view and light direction and using them, we are able to precompute Equation 3.4 into a 3D lookup texture.

It can look a bit suspicious that even though we want to observe the atmosphere also from outside of it, we are calculating only with positions inside of it. But in fact, there's no light scattering in space, so for observer situated outside the atmosphere, the amount of scattered light is the same as if they were 'standing' on the upper atmosphere boundary.

The utilization of the precalculated dataset is not difficult. All we have to do is calculate the coordinates into the 3D texture during runtime for each fragment and fetch the appropriate atmosphere colour. In the fragment

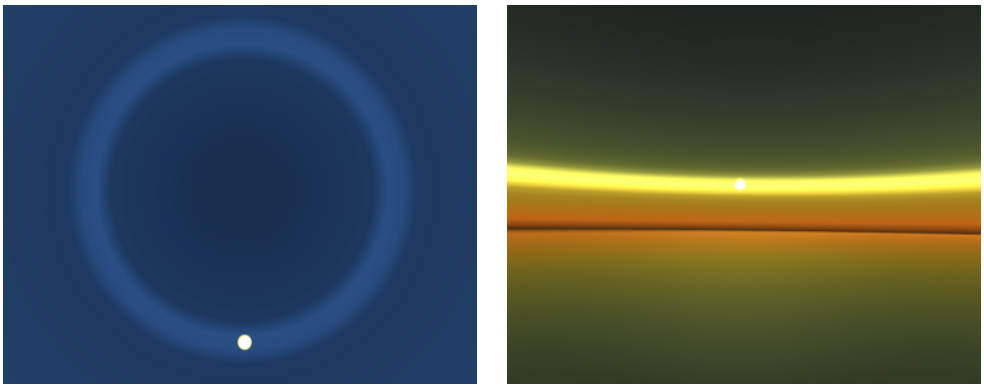


Figure 3.3: Errors in Mie scattering when using $h\phi\delta$-parameterization

shader, we have available all of $\vec{P}$, $\vec{V}$ and $\vec{L}$ and all that has to be done is to calculate the altitude of $\vec{P}$ over the ground and angles between $\vec{P}$, $\vec{V}$ and $\vec{P}$, $\vec{L}$ and use these values as coordinates into the lookup table (see [8] for the original fragment shader code).

This approach looks good, but soon I figured out that there's a lack of one dimension in such parameterization. Analyzing it I found out that the resulting lookup table stores the scattered light only for configurations when $\vec{V}$ is coplanar with $\vec{P}$ and $\vec{L}$. For any other situation, when the observer looks around, the table fails, because it completely neglects the *azimuth* (I will refer to it as ω) between $\vec{V}$ and $\vec{L}$. This causes uniformity of sky colour with respect to ω and nasty-looking 'rings' of Mie-scattered light centered around the zenith (see Figure 3.3 for examples of these errors).

This represents a problem, because the fourth dimension can't be simply squeezed into the texture. Thus, to get plausible results, I had to figure out the way to incorporate the remaining dimension into the whole concept. In the next section I'll show two different solutions of this problem.

3.4 Extended precomputation of single scattering

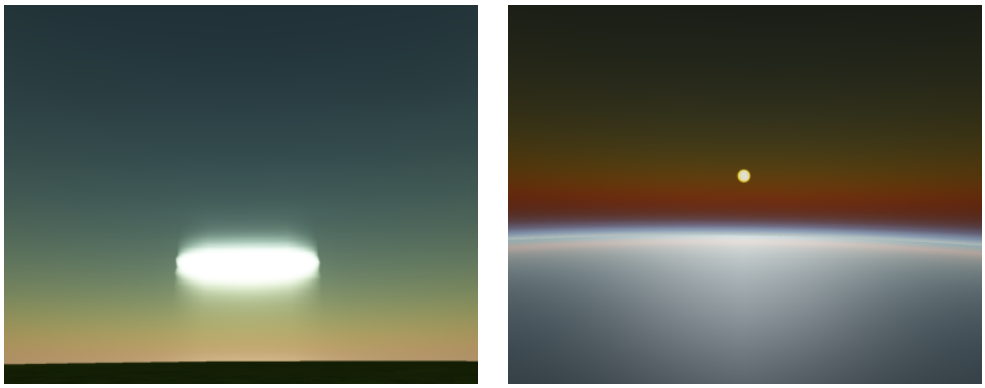
Despite the fact that hardware does not support 4D textures, it's theoretically possible to *emulate* it by a 3D texture[4]. However in practice, this is completely useless because of hugeness of the resulting data and also because the addressing in shader would be very awkward.

For the first solution of the problem shown in Section 3.3 I've created a different parameterization of the 3D lookup texture. The angles ϕ and δ

[4]Similarly to emulating a 3D texture by a 2D texture by making the regular tiles on 2D plane that would represent slices of the emulated 3D texture

remain, but instead of h the third dimension is the azimuth ω. The precomputation is then performed as follows: first, an arbitrary altitude in the atmosphere is chosen, for example the sea level; then both $\vec{V}$ and $\vec{L}$ are calculated and $\vec{V}$ is aditionally rotated by ω around the y-axis. 3D lookup texture then stores correct values for all view and sun directions, but only for one altitude. To be able to display the atmosphere in different altitudes, a few of these textures must be calculated, each of them for different height[5]. Then in fragment shader, we have to fetch the atmosphere colour from both lookup textures adjacent to current observer's altitude and to linearly interpolate between them.

This is indeed a very simple and *brute force* approach. It's in fact the emulation of 4D lookup table with one dimension sampled very roughly. It would be suitable only for such applications where it's certain that the observer can be situated in only one or few height levels, because the linear interpolation produces quite disturbing visual transitions between them. It also produces unacceptable results for Mie-scattered light, because unlike the Rayleigh scattering, the Mie scattering has sharp colour transitions. The resolution of such lookup texture is simply insufficent and this results in very blocky-looking Mie-scattered light (see Figure 3.4).



(a) (b)

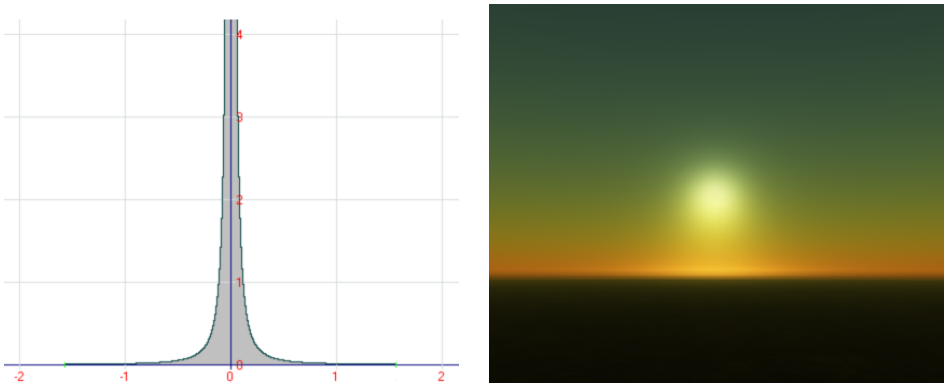
Figure 3.4: Errors in $\phi\delta\omega$-parameterization — blocky-looking Mie scattering (a) and visible interpolation between two height levels (b)

A more suitable approach would be, if one dimension could be excluded from the precomputation and computed during the rendering itself. At first I was considering elimination of h, but I wasn't successful in analytically expressing the light scattering for different altitudes from the lookup texture fixed in one height level. So I ended up with the solution explained in previous paragraphs. Then I realized it should be possible to exclude ω. The

[5]For example at ground level, in half of the atmosphere height and on the top of the atmosphere

reason of such possiblility is this: during the precomputation, the only usage of ω is to rotate the view direction $\vec{V}$ around the y-axis to include all possible viewing directions. However after such rotation, the distance between the observer's position $\vec{P}$ and the intersection point of $\vec{V}$ with outer atmosphere boundary stays the same, as the atmosphere is spherical and centered in the origin. Thus the path along which the light scattering is calculated remains the same, only the angle θ between $\vec{L}$ and $\vec{V}$ is altered. But θ only represents the parameter for the phase function which only multiplies the outermost in-scattering integral (see Equation 3.4).

This simple idea allowed me to exclude calculation of $F(\theta)$ from the precomputation step — for calculating the 3D lookup texture original $h\phi\delta$-parameterization is used, but the $F(\theta)$ term is excluded from it. Instead it's precomputed into a separate 1D lookup table parametrized only by θ and this is loaded into fragment shader along with the main 3D scattering texture. Then, the proper value of $F(\theta)$ is fetched according to $\vec{V}\vec{L}$ angle and multiplied with the previously fetched scattering colour to get the final colour of the sky.



(a) (b)

Figure 3.5: The graph of Cornette's F_M for $g = -0.95$ (a) and the correct sampling of Mie scattering with enhanced $h\phi\delta$-parameterization (b)

This solution removes both problems of $\phi\delta\omega$-parameterization. The problem with rough sampling of the altitude is now irrelevant, because the original $h\phi\delta$-parameterization with fine altitude sampling is used. The second problem — the blockiness of Mie scattering — is also removed, because the sharp colour transitions of Mie-scattered light are caused by the rapid rise of F_M in small scattering angles. In $\phi\delta\omega$-parameterization, the resolution of the scattering texture is simply insufficent to correctly catch this rise, but with real-time computation of F_M, this problem does not occur anymore as we have now the F_M sampled with per-fragment accuracy (see Figure 3.5 for details).

From now on, I'll be considering only the enhanced $h\phi\delta$-parameterization, as it's the only one that produces plausible results.

3.5 Rendering of planetary surface

Until now, I've been describing only rendering of the atmosphere. To be able to render the surface of the planet, a few more light contributions must be taken into account.

For surface shading, the Strauss local reflectance model [9] is used. I've chosen it because of its approximation of specular Fresnel terms (which can be precomputed in 1D lookup tables) and for the physically based parameterization of shaded surface. Its implementation is quite compact and is suitable for rendering realistically-looking surfaces in real-time.

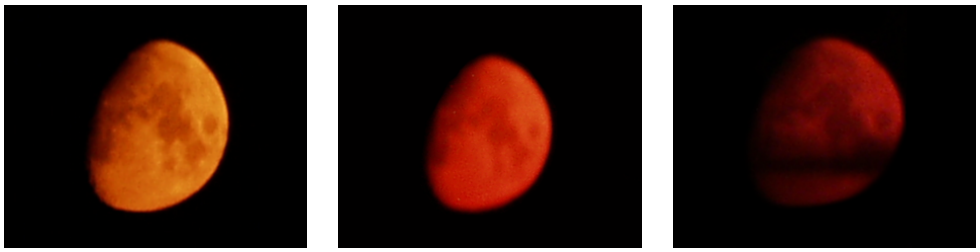


Figure 3.6: Real photos of the Moon approaching the horizon. It's clearly visible that the light gets weaker and reddish with the increasing layer of atmosphere that it has to pass through

Firstly, the light intensity coming from the light source changes after passing through the atmosphere. In particular the light gets attenuated and gains reddish hues before reaching planetary surface (see Figure 3.6). During this, some portion of photons gets scattered, and this loss is described by the optical depth t defined in Equation 3.3. The intensity of light reaching the planetary surface at point P_G is denoted by the equation

$$I_P(\lambda) = I_i(\lambda) exp(-t(P_G P_c, \lambda)) \tag{3.6}$$

where I_i is the intensity of incident light on the outer atmosphere boundary and P_c is the intersection between $\vec{L}$ and the upper atmosphere boundary. Since the length of the path from P_G to P_c is directly dependent on the angle between $\vec{P_G}$ and $\vec{L}$, it's possible to precompute the optical depth into a 2D lookup texture to avoid expensive real-time evaluation of it[6].

[6]The second dimension of this texture is represented by h — the altitude of P_G; we'll see later why it is needed

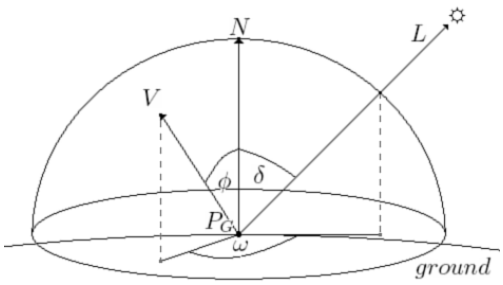


Figure 3.7: Schematic view of evaluation of Equation 3.7

Secondly, the ambient light in P_G must be accounted. Apart from the direct illumination discussed in the previous paragraph, the ambient illumination in P_G is caused by the intensity of scattered light coming from the sky segment that is visible from P_G. Assuming the perfectly spherical planetary surface, the ambient illumination at P_G can be calculated as

$$I_A(\lambda, \delta) = \int_{-\pi/2}^{\pi/2} cos(\phi) \int_0^{\pi} I_V(\lambda, \phi, \delta, \omega) d\omega d\phi \tag{3.7}$$

where I_V is calculated using the same method as in the $\phi\delta\omega$-parameterization. Since calculation of I_A is quite time-consuming, I'm also precomputing it into a 1D lookup table. This table is parameterized only by the sun angle δ between $\vec{P_G}$ and $\vec{L}$ because all points on the planetary surface with the same δ receive the same amount of ambient light (see Figure 3.7).

Thirdly, for a more realistic image of water surfaces, it's necessary to calculate the reflection of sky on them. Thanks to the nature of the 3D scattering lookup texture, it's also possible to use it as an environment texture (assuming we want only reflection of the sky). The intensity of the reflected light I_R at P_G, assuming P_G lies on the water surface, can be simply obtained as a 3D texture lookup with altered coordinates. The altitude is set to $h = 0$, ϕ is now the angle between $\vec{P_G}$ and $\vec{R}$ ($\vec{R}$ is the reflection vector at P_G) and δ has to be the angle between $\vec{P_G}$ and $\vec{L}$. Because the amount of reflected light generally depends on the angle under which the reflecting surface is observed, I use a very simple water reflection model based on this angle to obtain water surface colour, I_W:

$$I_W(\lambda) = \alpha I_S(\lambda) + (1 - \alpha) I_R(\lambda) \tag{3.8}$$

where $\alpha = max(\vec{N} \cdot \vec{V}, 0)$ and I_S denotes the intensity of shaded surface at P_G ($I_S = I_A + I_D + I_M$, I_D and I_M are the diffuse and specular terms calculated by the Strauss local reflectance model).

The last thing before getting the correct planetary surface colour I'_S is calculation of the attenuation of light on its path from P_G to observer. This

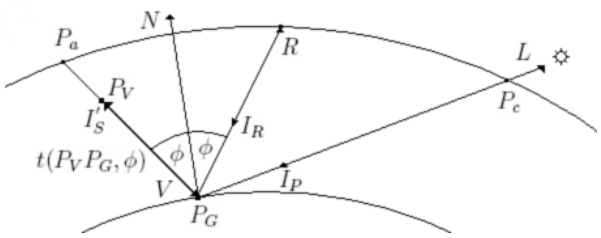


Figure 3.8: Schematic view of calculation of I_R, I_P and $t(P_V P_G, \phi)$

is again due to the light scattering on the atmosphere layer between P_G and observer. Thus, calculation of this attenuation is very similar to the calculation of I_P, except one thing. To get I_P, only the full optical depth from outer atmosphere boundary to ground must be accounted. However for I'_S, we must consider the fact that the observer can be situated anywhere inside the atmosphere. This situation is depicted in Figure 3.8. There's a little problem, because the 2D lookup table stores only the optical depth from top of the atmosphere to any altitude inside it. To solve this, two lookups have to be done - the first lookup for P_G gives us the optical depth from P_a to P_G, the second lookup for P_V gives us the optical depth from P_a to P_V (this is the reason why the optical depth lookup table must have 2 dimensions — the second dimension is h, the altitude of sample point). The correct optical depth $t(P_V P_G, \lambda)$ is then simply obtained as

$$t(P_V P_G, \lambda) = t(P_a P_G, \lambda) - t(P_a P_V, \lambda) \tag{3.9}$$

and the resulting colour of planetary surface (land or water) is calculated as

$$I'_S(\lambda) = I_{S|W}(\lambda) exp(-t(P_V P_G, \lambda)) \tag{3.10}$$

The definitive colour of planet is obtained as sum of I'_S and the previously calculated scattered light intensity from atmosphere.

There's yet one more note to make. The technique presented in this section correctly handles only a spherical planetary surface. The expansion necessary to render elevation data would need several corrections including correction of the amount of atmospheric scattered light, the optical depth and the amount of ambient light. Surface perturbations necessary for modelling water waves would also break the reflection model for some cases. Some of corrections treating the terrain morphology have been presented in Schafhitzel et al. [8], but these are insufficent for my extended approach. I've already started implementation of suitable corrections, but without a robust terrain renderer, which would exceed the extent of my thesis, they are useless.

3.6 Technique summary

The amount of steps necessary for plausible real-time rendering of the planet with atmosphere may seem overwhelming, even though this was only a brief explanation of the whole algorithm. So I feel that a little summarization could help the reader to sort all facts into a better-knit image of presented techniques.

The behaviour of light in the conditions of planetary atmosphere is approximated by the single-scattering model. As this is severely complicated, for run-time applications it is essential to precompute as many equations as possible into various lookup textures. The resolution of these textures directly influences the final accuracy of output pictures.

The colour of the sky is determined by gathering the scattered light along the view ray. This is stored in the main 3D lookup texture for every possible viewpoint and directly fetched in the fragment shader. The only correct parameterization of this texture is $h\phi\delta$-parameterization with deferred evaluation of phase function in fragment shader.

For rendering the planetary surface, the light from the source star has to be attenuated during its pass through atmosphere. The part of the scattered light during this pass reaches the ground as the ambient light gathered from visible segment of sky dome. The colour of water areas is also influenced by the reflected sky light, the contribution of which depends on the angle under which the water surface is observed. The final colour of the planetary surface is obtained by attenuating the surface light along its path to the observer again due to the light scattering and adding contribution of scattered light from the atmosphere.

Chapter 4

Implementation

The aim of this part of my work was implementation of two standalone applications, whose conjunction would allow the user to display an arbitrary planet with atmosphere in real-time framerates. The emphasis should be on finding a reasonable tradeoff between image realism and rendering speed.

The first application was intended to be a **batch-based highly configurable precomputing utility**. This application should allow user (with knowledge of the topic) to set numerous physical parameters of both planet and its atmosphere and then generate whole dataset containing atmospheric data for utilization by:

The second application, a **realistic 3D real-time planet renderer**. By using precomputed atmospheric data, this application should render a fully textured planet with its atmosphere. It should furthermore allow the user to navigate freely around the planet, to use various visual features and also to change numerous parameters of the atmosphere that are not hardwired into the precomputed dataset.

4.1 Environment and libraries

The programming language chosen for implementation of both applications is C++, because of the emphasis on application speed (as in most cases of graphical applications) and also because of broad spectrum of freely available libraries and utilities. HLSL was chosen for shaders' implementation. The target operating system is Microsoft Windows XP or higher, the development was done on XP with SP2.

For the implementation of both applications, the following libraries are used:

Win32 API - GUI library for all MS Windows systems. Since the accent is on speed, the user interaction is very simple and is limited to keyboard shortcuts and mouse navigation. For this purpose, WINAPI is more than sufficent.

Microsoft Direct3D - 3D graphics API/library. I'm using it because of my previous positive experience with it and richness of included functions and utilities. Moreover, its distribution is monolithic and therefore very compact, including all needed libraries and also very useful sample applications (with source codes).

TinyXML - Simple and compact *.xml* files parser[1]. Used for loading both applications' settings. Thanks to Lee Thomason for the free source code distribution.

Boost C++ Libraries - Huge library distribution for C++[2]. I use only the CONVERSION module for conversions between numerical types and strings.

4.2 TextureCreator

TextureCreator is the precomputing application. Its purpose is the creation of lookup textures that will be later utilized by the rendering application. The precomputing itself is a batch task and can last several hours, depending on desired lookup textures' resolutions and quality.

The source code structure is very simple with accent on functionality and transparency. The application consists of a main module that creates the window and Direct3D device and manages lookup textures' creation and storage on HDD, and the namepace `generator` that contains the set of callback functions that are used for lookup textures' calculation.

The precomputation is parameterized from a single *.xml* file. This contains the generator settings such as resolutions and sampling rates and also many physical parameters that participate in the calculation process. All settings are loaded before the precomputation starts and thus can't be changed during the calculations.

The output of TextureCreator depends on settings specified before the execution, but in the default settings, it produces the full set of lookup textures needed for atmosphere rendering. All textures are generated using $64b$ floating-point texel format (`D3DFMT_A16B16G16R16F`) and *.dds* file format[3]. This is essential, because the physical nature of generated data implicates high dynamic range of resulting intensities. Using such format, the resulting textures may become quite large, mainly the 3D scattering texture(s). So let's take a look on the possible memory consumption. Suppose we want the main scattering texture to have resolution of 256^3, what is quite reasonable; then the size of the resulting texture is simply calculated

[1] http://www.grinninglizard.com/tinyxml/index.html

[2] http://www.boost.org

[3] Even more suitable would be to use an IEEE $32b$ float per-component, but not even high-end hardware yet commonly supports filtering of $128b$ texel formats

as $2^8 * 2^8 * 2^8 * 8b = 128MB$. Multiply this by 2, because we need to calculate both Rayleigh and Mie scattering. The resulting size of $256MB$ is too high, because along with the main scattering textures, the rest of the lookup textures and also planetary surface textures must be placed in texture memory. I've tried to DXT-compress them — this shrank both textures to 1/8 of original size, but the resulting quality was very poor and no filtering helped to improve this.

So I had to tweak the main lookup texture a bit. It can be noticed, that thanks to the fact that Mie scattering is λ-independent (see Section 3.1), it can be stored in only one texture channel instead of 3. Therefore I store both Rayleigh and Mie scattering in only one lookup texture — Rayleigh in RGB components and Mie in α component. This reduces the size of 3D lookup texture back to $128MB$, which is an acceptable value[4].

For the evaluation of integrals in equations 3.3, 3.4 and 3.7 I use the trapezoid evaluation rule with uniform sampling interval. It could be more suitable to use adaptive sampling with weight function that would prefer smaller sampling intervals for lower altitudes (the scattering function changes more dynamically in them), but this would complicate the solution and as I found out, uniform sampling works well if used in the trapezoid rule. The sample number of 15 produces yet acceptable results and above 50, the convergence of the approximation is already very slow.

All equations discussed in Chapter 3 use λ for the light colour representation. To be able to display precomputed scattering values (that fundamentally must be calculated using wavelenghts), the corresponding wavelenghts must be converted to RGB colour representation. I've implemented two different approaches for this:

RGB equivalent wavelenghts - the basic approach. During the calculation three different wavelenghts are used (configurable by the user), each corresponding to one colour component. No conversion is then needed, as the intensity of the concrete colour component is equal to scattered light intensity for its corresponding λ.

Multifrequency scattering calculation - in this extended, physically more correct approach I'm calculating the light scattering for 191 wavelenghts ranging from 350nm to 730nm with the step of 2nm. For converting wavelenghts to RGB values I use a simple algorithm by Dan Burton [1]. The final RGB colour value is obtained as average of converted values weighted by the corresponding intensities of scattered light and also by intensities of blackbody radiation (in current implementation, these constants are hardcoded in the application and

[4]Using a card with $256MB$ video memory on PCIe bus and Direct3D's `D3DPOOL_MANAGED` memory pool will allow us to use also very large textures for planetary surface representation thanks to fast transfers between video and system memory

correspond to the incoming radiation from Sun on the top of Earth's atmosphere). It may seem this enhancement will drastically elongate the precomputation time, but in fact all time-consuming integrations are λ-independent — only the β constant in Equation 3.3 and $\frac{1}{\lambda^4}$ term in Equation 3.4 contain λ, and as these are involved only in multiplications with integrals, the real computational time growth is only few percent.

4.3 AtmoVision

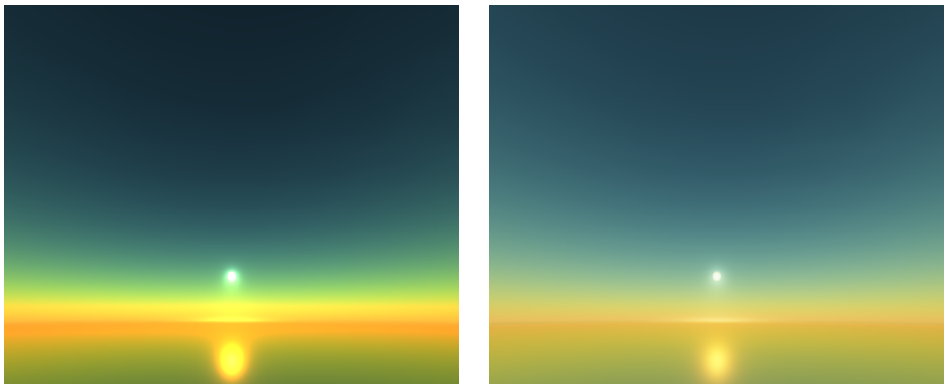
AtmoVision is the main rendering application. It's capable of real-time rendering of one planet with atmosphere, free camera navigation around the planet and setting various graphical and physical parameters.

The structure of source code is fairly simple. It contains the main application and 4 classes. The main application does the usual work — creates and destroys application window, device and all classes, contains the rendering routine and handles keyboard and mouse interactions. The `DataNode` class contains public data members representing settings parameters loaded from configuration *.xml* file and is available for main application and the rest of classes. This class is also used by TextureCreator, because they use the same settings file. It's very important for AtmoVision to use the same settings (except graphical) as were used for dataset generation, otherwise the planet rendering will likely get spoiled. The `AtmosphericPlanetEx` class represents the atmospheric planet itself. It implements functionality connected with the planet or the atmosphere including their creation/destruction, features' functionality and also rendering. The `Camera` class represents the observer and implements the functionality connected with looking and movement in space. Finally, the `SunSprite` class represents the billboard with star texture mapped on. It represents the direction to light source and moves around the camera to create the illusion of being infinitely far.

As indicated in Chapter 3, purely the programmable pipeline is used for the whole rendering. Because the size of vertex and fragment programs' source code is quite large, I'm using the DIRECT3D's *.fx* effect files for wrapping all samplers, vertex/fragment programs and techniques. AtmoVision uses 4 *.fx* files — one for each of the atmosphere, planet, star and offscreen surface rendering.

The geometry of both planet and atmosphere is represented by 2 uniform grid meshes that are programmatically wrapped around the corresponding spherical limit surfaces[5]. I don't use any external models, instead I create them procedurally during the application initialization (as I found out, the

[5]Note that the atmosphere does not need any additional sampling spheres often used by some of real-time light scattering rendering techniques



(a) (b)

Figure 4.1: Example of scene with oversaturated (down) and undersaturated (up) areas (a) and the same scene with software HDR correction (b)

time needed for the mesh generation is minor in comparasion with the time consumed by the loading of all textures into memory).

The rendering of both objects is done primarily by the fragment shaders and follows the technique described in Chapter 3. It's good to say that the planet is rendered with backface culling while the atmosphere with frontface culling. The reason for this is the fact that during the rendering of planetary surface, also the scattered light from the atmosphere is accounted. This implies that if the camera is situated above the upper atmosphere boundary, only the shining atmospheric 'halo' has to be rendered and if the camera is situated inside the atmosphere volume, the sky must be visible. The planet is rendered first and thanks to DepthStencil test, only needed parts of the atmosphere that exceed the planet are rendered. The atmosphere has to be rendered with alpha blending turned on because we want any object behind the upper atmosphere boundary to be visible, such as the Sun in my case (`DestBlend` and `SrcBlend` arguments should be both set to `One`).

Rendering of the natural phenomena related to light always implies the high dynamic range of incoming radiation intensities. This effect is very strong in atmosphere rendering. To avoid undersaturated or oversaturated scenes, it's very convenient to implement at least software HDR rendering. As I previously said, all lookup textures are in floating point format, so the only thing that must be done is to create an offscreen floating point render target and draw both planet and atmosphere into it. Then this surface is mapped as a texture onto a screen-sized quad, which is rendered using the following formula (taken from O'Neil [7])

$$Colour_{New} = 1.0 - exp(-cExposure \times Colour_{Old}) \tag{4.1}$$

in the fragment shader. `cExposure` constant denotes the software equivalent of camera exposure time. Usage of this formula provides a normalization of

all colours to interval $\langle 0.0, 1.0 \rangle$ and flattens the sharp colour transitions, as shown on Figure 4.1.

4.4 Tests and results

Before the testing itself, I feel it's reasonable to analyze the technique and both applications, as it can help us to make some expectations and understand the measured results.

TextureCreator is a batch application and it was written in this manner from the beginning. The emphasis is firstly on precision and physical validity. I've made many optimizations to aviod repetitive calculation of same equations or expressions, but no speed-ups that would decrease the precision of computation have been made. For manipulating all values (except counters) the `long double` native type is used. The only way for the user to affect the precomputation time is to change the sampling rate or the textures' resolution. The precomputation itself runs purely on CPU and consumes 100% of one core's time.

AtmoVision is based on the technique described in Section 3, which is designed with accent on minimal CPU load and maximal utilization of GPU. As has been said, most of the rendering work related to atmosphere is done in fragment shader. Fixed pipeline is not used at all and vertex shaders are not particulary complicated, they do only transformations to clip space and to world space (or in case of enabled normal mapping, to tangent space). I also do not use any *LoD* technique in the moment and because both planet and atmosphere are modelled by single spheres, there's no possibility of frustrum clipping (except the case when the entire object is not visible) and thus all geomerty is always rendered. It's then quite obvious that the *bottleneck* of the whole algorithm are fragment shaders[6] and the framerate will be directly dependent on the amount of rendered fragments. Moreover, the fragment shader for rendering planetary surface is much more complicated than the one for sky dome rendering (see section 3.5 for details). So it's expected that if the observer is situated inside the atmosphere, the more sky fragments and the less planetary surface fragments constitute the current view, the higher framerates will be encountered. Let's then proceed to the tests now.

The configuration used for testing was a desktop PC with *Intel Core 2 Duo @ 1866MHz* CPU, *2048MB @ 800MHz DDR2* RAM and *NVIDIA PCI-E GeForce 8800GT* graphics adapter with *512MB GDDR3* VRAM and *112 unified shader units*.

I used this testing concept: the camera starts at 1km over planetary surface looking straight down. Then the camera automatically starts pitching

[6] Assuming the reasonable amount of vertices in scene

at the rate of 6 degrees per second upwards for 30 seconds, ending in looking straight up. During this procedure, the FPS is measured every second. Then I take control of camera and am flying around the planet with effort to catch different viewing conditions around the planet. During this free flight, only the minimal and maximal framerates are measured and stored. I'm also measuring the amount of available texture memory.

I've made 5 tests of this kind but with certain differences. I'll call them TT_1–TT_5:

Test	**Screen Res**	**Size of textures**	**Atmosphere**	**# Vertices**
TT_1	800×600	full	off	low
TT_2	800×600	full	on	low
TT_3	800×600	reduced	on	low
TT_4	800×600	full	on	high
TT_5	2560×2048	full	on	low

TT_1 was performed to test the speed of vertex transformations and Strauss local reflectance model without rendering of the atmosphere. TT_3 was performed to find out if the size of used textures and potential transfers between video and AGP memory do or do not decrease the overall performance. The full-size set of textures uses 4096×2048 textures for planetary surface and 256^3 3D scattering lookup texture and the reduced-size set consists of 512×256 textures for planetary surface and 128^3 3D scattering texture. TT_4 was intended to examine how the highly increased number of vertices influences overall performance — the used graphics adapter works with unified shader architecture, which implies that the more computational power will be used in vertex transformations, the less will remain for fragment processing. In case of low number of vertices, the scene consists of 65536 vertices (equally divided between planet and atmospheric shell) and in case of high number, the scene consists of 1048576 vertices. TT_5 has been performed in the windowed mode to enable creation of such large offscreen buffer, because the used display has the resolution only 1280×1024. The purpose of this last test was to verify the dependence between the amount of rendered fragments and the overall performance.

The results of all five tests are shown on the graph in Figure 4.2. The remaining measurements are written in the table below:

Test	**Min/Max/Avg FPS**	**Init/Run TexMem**	**MPx/Frame**
TT_1	786/1227/1079.74	$736MB/549MB$	0.48
TT_2	570/1266/864.13	$736MB/399MB$	0.48
TT_3	520/1288/869.10	$736MB/621MB$	0.48
TT_4	68/166/132.91	$736MB/344MB$	0.48
TT_5	62/342/123.26	$643MB/305MB$	5.24

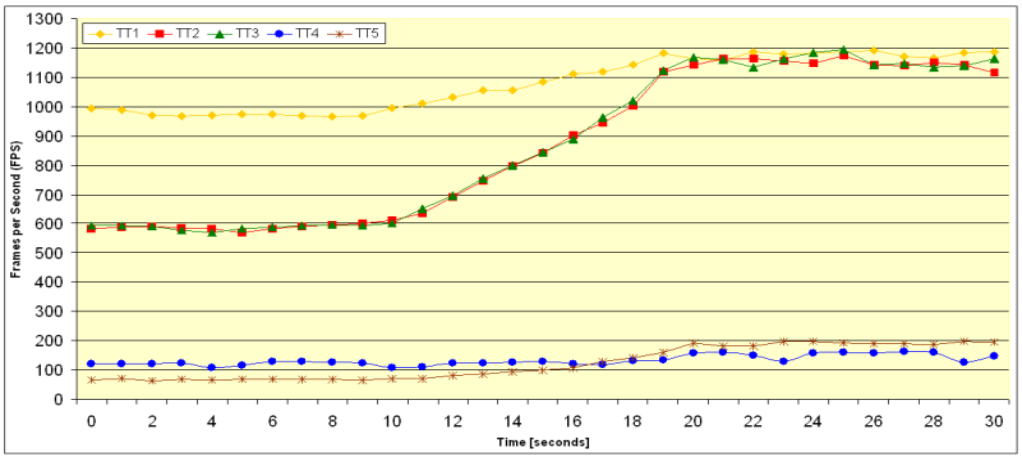


Figure 4.2: Results of tests TT_1–TT_5 — the dependence of FPS (y-axis) on time (x-axis)

In the table, Min and Max FPS refer to limit framerates during the whole runtime[7], while the Avg FPS refers to the average framerate value only during the automatic camera action. Init and Run TexMem refer to the amount of free texture memory immediately after device creation and during the application runtime, respectively. MPx/Frame represents the peak amount of fragments that had to be rendered each frame.

Let's now analyze the obtained results. It can be seen that measurements fit quite well with the predictions.

TT_1 (yellow diamonded line) shows the performance of vertex shaders and fragment shaders when rendering only the planet using Strauss reflectance model. No scattering is calculated yet. This is the kind of reference values from which I'll derive the performance of my technique.

TT_2 (red squared line) shows the performance with atmosphere turned on. The memory consumption in comparasion to TT_1 grew about $150MB$, which is exactly the size of all lookup textures with generated *mipmaps*. The pattern suggested in the beginning of this section has been confirmed — in around the half of the automated test, the performance starts to grow and ends up in double framerates compared to the beginning. This growth of course corresponds to the transition between the ground and the sky, when the camera view passes through horizon.

[7]Since the second phase of all tests consists of free flight, the Max FPS value very likely corresponds to the case when the camera looks into empty space; this could be understood as the performance when rendering purely the geometry, because the planet is behind the camera and thus no fragments are generated

TT_3 (green triangled line) represents the performance when using small textures. We see that in spite of the fact that the memory consumption decreased about 3 times comparing to TT_2 the performance hasn't risen at all. This proves that thanks to the size of available video memory and the bandwidth of PCIe BUS, the performance is not dependent on the size of lookup textures. This implies that we can use lookup textures of an arbitrary precision without a performance penalty, if these can fit into the texture memory.

TT_4 (blue circled line) the performance when using high amount of polygons for representing the planet and the atmosphere. It's clear that the high number of polygons eats up too much of shading units, leaving too little computational power for fragment shading.

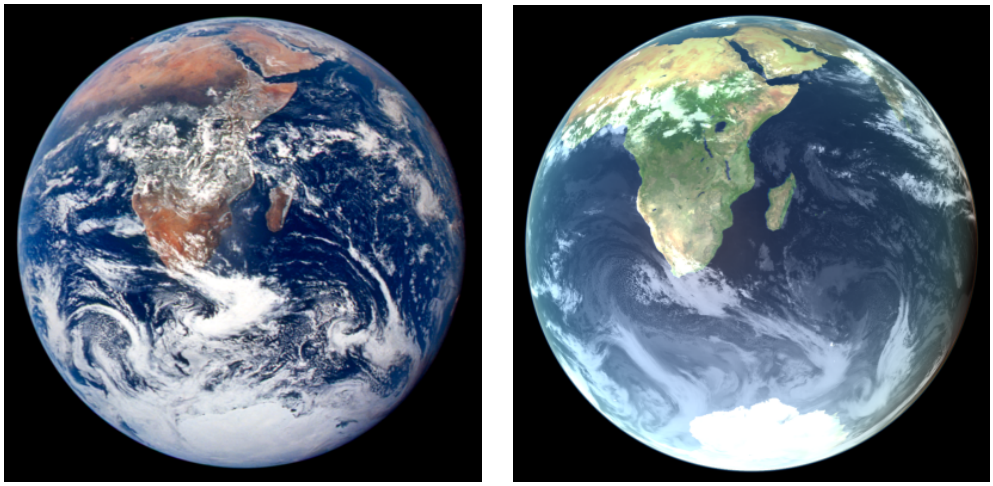
TT_5 (brown crossed line) proves that the performance is very strongly dependent on the amount of rendered fragments. This test has also proven the power of this method, because even in such high resolution the framerates stayed around 120 FPS and never dropped under the desired 60 FPS. I assume that in combination with some reasonable LoD technique for terrain rendering and multiple graphics adapters, this technique could be used also for simulators running on very large displays in high framerates.

So the majority of tests agree well with the predictions made in the beginning of this section. The slight exception is my underestimation of performance cost of vertices' transformations. Comparing the results of TT_4 and TT_5, it shows up that if the number of vertices in the scene closes on the number of fragments, the performance drops down quite rapidly. This implies that the implementation of some robust terrain rendering method is strongly needed for this type of applications.

In all tests I've been using my enhanced $h\phi\delta$-parameterization. I skipped tests of the $\phi\delta\omega$-parameterization, because the code is very similar to the previous one, the only additional thing is to compute the azimuth between $\vec{V}$ and $\vec{L}$ and one more 3D lookup texture fetch. In average case, this would represent only a few % of all computations.

The last thing to mention is the performance cost of HDR rendering. In my experience, this is only $2-3$ frames on 800×600 screen resolution and even less on higher ones, because in DIRECT3D there's only a neglectable preformance penalty for rendering to offscreen frame buffer, and the rendering of the screen-sized quad represents only a small fragment of instructions that are needed for computation of the scattering itself.

Some of the results of my technique can be found on Figures 4.3, 4.4 and 4.5.



(a) (b)

Figure 4.3: Real photo of Earth from Apollo spaceship (a) and screenshot from AtmoVision (b) (the cloud layer has been added to increase realism). The different colour of the land is determined by the used surface texture

(a) (b) (c)

Figure 4.4: Orbital view of the horizon: real photograph from space shuttle (courtesy of NASA) (a), ray traced image (b) and screenshot from AtmoVision (high resolution) (c). (a) and (b) are taken from Nishita et al. [5], reddish tones instead of black are caused by the compression

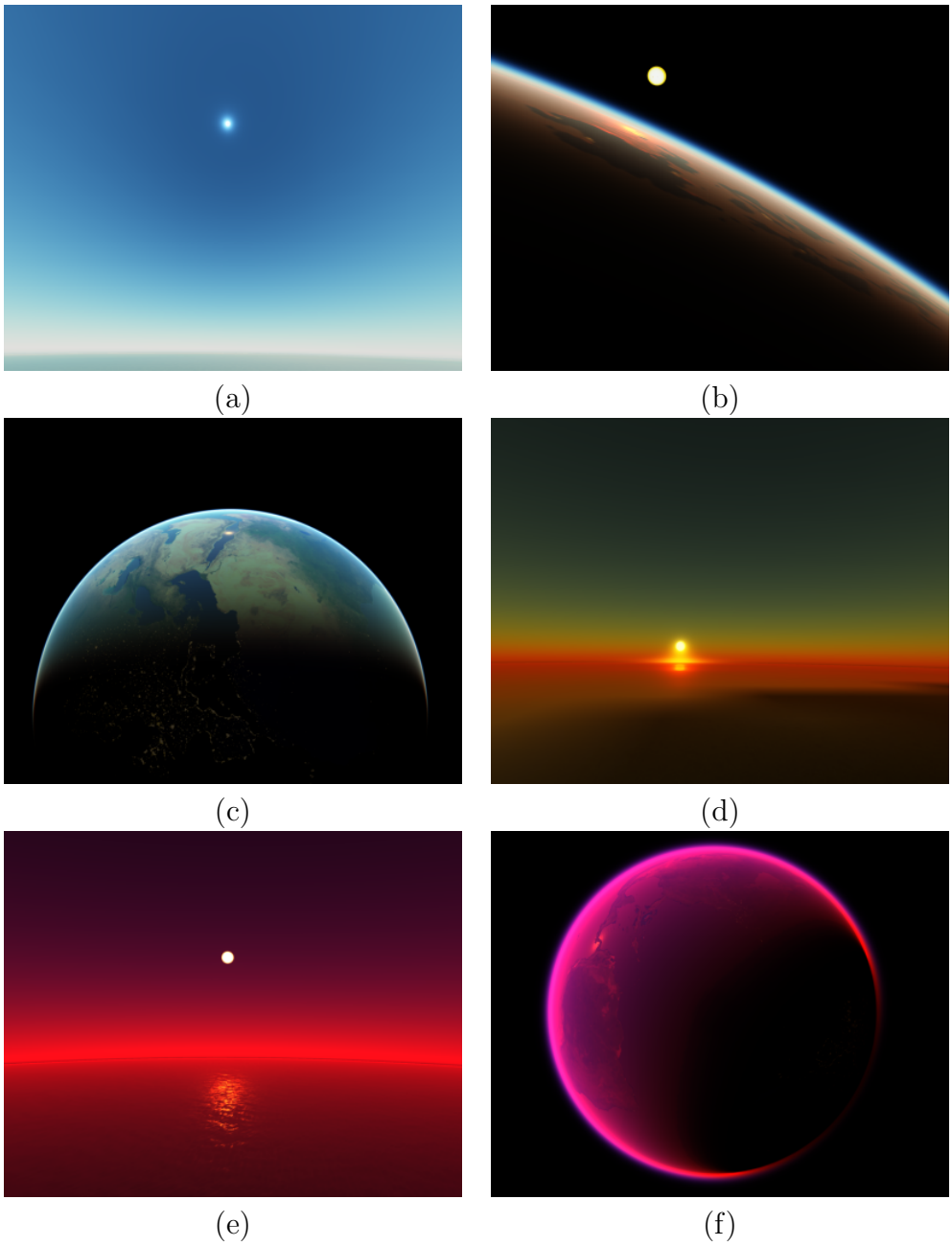


Figure 4.5: Screenshots from AtmoVision: the 'fisheye' projection of daytime sky (a); sight on the terminator over Indonesia (b); orbital view over the Middle-East (c); sunset at Spain (d); examples how different parameterization can radically change the overall look of the planet (e) and (f)

Chapter 5

Conclusion

5.1 Summary

In the first chapter I've opened the topic of rendering the physical phenomena and explained why I think the realistic rendering of the atmosphere as the phenomenon of light scattering is important and what applications may need it. I've also presented aims of my work.

In the second chapter I have briefly presented the physical fundaments of light scattering and shown how such complex problematique can be described by a mathematical model. Then I've presented a few works dealing with realistic atmosphere rendering.

The third chapter presents the method for real-time rendering of planetary atmospheres based on previously listed works. At first the mathematical model of single scattering is described. Secondly I explain how such complicated mathematical equations can be implemented in a real-time rendering algorithm. Finally, two such methods based on similar idea are presented along with specialities involved in the rendering of realistic planetary terrain.

The fourth chapter describes the pair of applications implementing techniques from Chapter 3. At first the internal structure of both of them is stated. Then the rest of the chapter devotes to testing these applications and exploring their abilities and weaknesses.

5.2 Fulfillment of goals

In my work, I have succeeded in fulfilling most of the objectives I've demarcated in first chapter:

1. I have examined 4 works on the field of atmospheric scattering. All of them are aiming for real-time implementation and thus have been great knowledge bases for my thesis.

2. For my implementation I chose the work by Schafhitzel et al. [8]. In this work they are calculating the atmospheric scattering per fragment from data precomputed in a 3D lookup texture. I've implemented this method, analyzed the results and figured numerous modifications that I considered necessary for increased fidelity of the rendered atmosphere and planetary surface. For the correction of missing fourth dimension of 3D lookup texture, I've deferred evaluation of the phase function, resulting in correct Mie scattering and anisotropy of Rayleigh scattering in respect to $\vec{V}\vec{L}$ azimuth. For terrain rendering I'm calculating light attenuation due to out-scattered photons, an ambient term described by Nishita et al. [5] and also the water reflections.

3. I've implemented the enhanced technique in two programs. The first, batch-based, performs the precomputation step and generates all lookup textures for the second application, which uses these textures for real-time computation and rendering of atmospheric light scattering. Thereby the method proved itself suitable for the task by its capability of rendering the realistic atmosphere even in hundreds of FPS. Also the majority of artefacts resulting from this method has been removed by various corrections. I've also reached the goal of high utilization of GPU and low consumption of CPU time.

5.3 Discussion and future directions

When I was starting this project, my knowledge of computer graphics was very poor. I'm glad for this topic, because I learned a lot of techniques during the time I invested into this work. I also realised that the rendering of light scattering and other physical phenomena is very broad and inherently interesting topic and I'd like to investigate it even more. I won't go into details of the work itself, because it's full of them and I have discussed them anytime I felt it necessary.

In the text I mentioned a few times the desperate need for sophisticated terrain rendering. The atmospheric scattering constitutes the overall colour visage of the planet, but it is terrain that adds the details. And these are equally important to create a sensation of looking on some huge object such as a planet, because without them, it looks dull. Moreover, the reasonable LoD technique would also decrease the amount of rendered geometry primitives, leaving more computational power for computation of light scattering.

Even though I've neglected the rendering of clouds, these are also a very important element in realistic atmosphere rendering. There is plenty of

techniques aiming on this problem, but the problematique itself is so large that it is beyond the extent of this work.

I also think that rendering of realistic water deserves attention. Again, many works dealing with this exist, but I'm not aware of any that would combine physical modelling of waves, realistic reflections of the sky and light scattering on water molecules. Because of this, I plan to further examine this topic in detail.

Bibliography

[1] Burton, D.: *Approximate RGB values for Visible Wavelengths*, http://www.physics.sfasu.edu/astro/color/spectra.html

[2] Cornette, W. M., Shanks, J. G.: *Physical Reasonable Analytic Expression for The Single-Scattering Phase Function*, Applied Optics Vol. 31, No. 16, 3152-3160, 1992

[3] Fernando, R. and Kilgard M. J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley, 205-207, 2003

[4] Gerasimov, P., Fernando, R. and Green, S.: *Shader Model 3.0: Using Vertex Textures*, NVIDIA white paper DA-01373-001_v00, 2004

[5] Nishita, T., Sirai, T., Tadamura, K., Nakamae, E.: *Display of The Earth Taking into Account Atmospheric Scattering*, Siggraph '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, 175-182, 1993

[6] O'Neil, S.: *Real-Time Atmospehric Scattering*, http://www.gamedev.net/reference/articles/article2093.asp, 2004

[7] O'Neil, S.: *Accurate Atmospheric Scattering*, Addison-Wesley, GPU Gems 2, 253-268, 2005

[8] Schafhitzel, T., Falk, M. and Ertl, T.: *Real-Time Rendering of Planets with Atmospheres*, Journal of WSCG, Vol. 15, 2007

[9] Strauss, P. S.: *A Realistic Lighting Model for Computer Animators*, IEEE Computer Graphics and Applications, Volume 10, Issue 6, 56 - 64, 1990

[10] Wikipedia The Free Encyclopedia: *Earth's Atmosphere*, http://en.wikipedia.org/wiki/Earth%27s_atmosphere, 2008

Appendix A

Contents of DVD

The accompanying DVD is organized as follows:

- `/Data Sets` - a few precomputed datasets with configuration files used for their generation
- `/Documents`
  - `/Documentation` - contains the both applications' documentation
  - `/Others` - miscellaneous documents
  - `/Tests` - *.xls* file containing performance tests' results
  - `/Thesis` - contains thesis' *.pdf* file and LaTeX source code
- `/Installation` - both (*Full* and *Lite*) installation packages are located here, plus free *.dds* texture viewer
- `/Pictures and Screenshots` - contains screenshots from AtmoVision and also a few real images of Earth
- `/Source Code` - AtmoVision's and TextureCreator's complete source code

Appendix B

TextureCreator and AtmoVision quick reference

B.1 System requirements

This is the minimal recommended configuration for running AtmoVision and TextureCreator:

- CPU 1.2 GHz single-core
- Operational memory 512 MB
- VGA graphics adapter with support of DirectX 9.0c (TextureCreator)
- VGA graphics adapter with support of DirectX 9.0c, Shader Model 3.0 and 128MB VRAM (AtmoVision)
- HDD space 350 MB
- OS MS Windows 2000/XP

It's possible that they will run also on slower systems, but low performance and/or decreased precission due to insufficent texture memory may occur.

B.2 Installation

The installation is very simple and straightforward. Simply launch either *AtmoVision_Full.exe* or *AtmoVision_Lite.exe* installation package located in the `/Installation` directory and the wizard will guide you through the whole process. The difference between them is that *Full* version already contains a full set of precomputed data, resulting in a much larger installation. The *Lite* version doesn't, so it's necessary to perform the precomputational step. Some series of precomputed lookup textures can also be found on source CD in the `/Data Sets` directory.

B.3 TextureCreator

TextureCreator is launched by executing either *TextureCreator_RGB.exe* or *TextureCreator_MFQ.exe*. These two builds are almost identical, except the wavelength colour representation. The first of them precomputes lookup textures using 3 RGB-equivalent wavelengths, the second uses 191 wavelengths through the whole spectrum (see Section 4.2 for details).

All settings for TextureCreator are located in common configuration file *Resources/Dbs.xml*. By default, the parameters are set to generate whole dataset in appropriate resolution and sufficent quality. However, this will take a few hours depending on the speed of your CPU. Some of the settings are described in the user documentation located in *Documentation* directory, the rest of them should be comprehensible from the descriptions provided in the settings *.xml* file.

TextureCreator contains no GUI, it consist only of one small window. The generation can be stopped anytime by pressing the *Esc* key, although the termination may not be immediate. The progress of precomputation process is displayed on window's title — there's a percentual information about how much of currently generated lookup texture is done and absolute number representing the order of currently generated texture (this will range from 1 to 12 in case that all textures are generated).

After succesfully finished execution, the following lookup textures will be located in *Data* directory:

tex_AmbSc_1D.dds	*tex_F_1D.dds*	*tex_G_1D.dds*
tex_HTD_3D.dds	*tex_OptDpta_2D.dds*	*tex_OptDptp_2D.dds*
tex_OTDL0a_3D.dds	*tex_OTDL0p_3D.dds*	*tex_OTDL1a_3D.dds*
tex_OTDL1p_3D.dds	*tex_OTDL2b_3D.dds*	*tex_ScPh_1D.dds*

B.4 AtmoVision

AtmoVision is launched by running the *AtmoVision.exe* from installation directory or one of the shortcuts created by the installator. Before execution, it's good to check the common settings file *Resources/Dbs.xml*, if the graphical settings suit your machine. It's necessary that you use the same settings file which was used for the dataset creation and modify only parameters related to the runtime, such as graphical settings, detailness of models or the desired parameterization. Also, paths to surface textures are specified here.

Besides the settings file, the input for AtmoVision are lookup textures generated by TextureCreator. AtmoVision expects them in the *Data* directory, the place where TextureCreator will store them. Not all textures are used at once — $h\phi\delta$- and enhanced $h\phi\delta$-parameterizations don't need those

with prefix *tex_OTD* and $\phi\delta\omega$-parameterization doesn't need *tex_HTD_3D.dds* texture.

After loading all textures into memory, the application starts with camera on Earth's high orbit and you can immediately work with it. All controls are described in the user documentation and also briefly on the yellow tooltip on the left side of screen. The application is terminated by pressing the *Esc* key.

Appendix C

Physical constants and parameters

Such large and complex object as the planetary atmosphere has to be described by many parameters. Despite I'm using a simplified model for its simulation, the amount of necessary constants and equations is still quite high and it took me a few days in all to find the data that would produce correct results. So I've decided to present some of them here in one comprendious table.

Constant	**Value[units]**	**Description**
g	$\langle -1; -0.75 \rangle$	Mie phase function parameter
H_R	7994[m]	Rayleigh scale height
H_M	1200[m]	Mie scale height
H_{TOP}	100000[m]	atmosphere upper boundary altitude
I_r	1.0	relative incoming light intensity (red)
I_g	0.960784	relative incoming light intensity (green)
I_b	0.949019	relative incoming light intensity (blue)
λ_r	705×10^{-9}[m]	wavelength (red light)
λ_g	530×10^{-9}[m]	wavelength (green light)
λ_b	440×10^{-9}[m]	wavelength (blue light)
n_r	1.000271287	refractive index (red light)
n_g	1.000274307	refractive index (green light)
n_b	1.000275319	refractive index (blue light)
N_{S_R}	2.653×10^{25}[p/m^3]	molecular density (Rayleigh particles)
N_{S_M}	1.5×10^{10}[p/m^3]	particle density (Mie particles)
R	6372797[m]	mean Earth radius
T	5778[K]	Sun's photosphere temperature

T is the value that is used only for multifrequency calculation (see Section 4.2). The used intensity values of blackbody radiation correspond to this temperature.