RWTH Aachen University

Bachelor Thesis

Interactive Visualization of Parameterized Atmospheres in Virtual Reality

by

Peter Collienne

RWTH Aachen University

Bachelor Thesis

Interactive Visualization of Parameterized Atmospheres in Virtual Reality

for the degree of B.Sc. in Computational Engineering Science

by

Peter Collienne
Student Id.: 296711

Prof. Dr. Torsten Kuhlen
Virtual Reality Group

Prof. Dr. Leif Kobbelt
Computer Graphics Group

Supervisor: Robin Wolff, Ph.D.

Date of issue: April 22, 2013

Statement

Hiermit versichere ich, daß ich die vorliegende Arbeit selbständig im Rahmen der an der RWTH Aachen üblichen Betreuung angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

I guarantee herewith that this thesis has been done independently, with support of the Virtual Reality Group at the RWTH Aachen University and no other than the referenced sources were used.

Aachen, April 22, 2013 .....

Abstract

The *German Aerospace Center* (DLR) developed a scientific planetary renderer to display and analyse terrain data collected from past and ongoing planetary space exploration missions in a Virtual Reality (VR) environment. While this renderer supports large datasets (> 1TB) and interactive framerates it lacks a realistic atmosphere. Especially in VR, an atmosphere helps with perceiving distance and allows for judging the relative distance to objects, also known as *aerial perspective* [Kel77].
This thesis describes the implementation of a realistic atmosphere, fittable to different planets using multiple parameters. Due to certain approximations to render the atmosphere in real-time, the result is not completely accurate, but yields a plausible atmosphere. It is based on work by Bruneton [BN08] and while it is designed to work as an universal plugin, the implementation is based on the planetary renderer provided by the DLR. In addition to the previous work, the parameters are fit to resemble the atmosphere on Mars, as seen on photographs taken by recent Mars-missions. Also a *Bloom*-Filter is additionally implemented to imitate the behaviour of the human eye or a camera when looking into the sun. The HDR-implementation of Bruneton is extended with the use of another filmic tonemapping operator and further improved by rendering a realistic looking sun.
The here presented implementation offers a real-time, parameterizable atmosphere renderer, which is easy to integrate in any planetary rendering engine.

Contents

CHAPTER 1

INTRODUCTION

1.1 Motivation

Atmospheric scattering effects are widely used in computer graphic applications like video-games, special effects in movies or in scientific visualization. Especially in applications aiming to simulate a realistic experience in Virtual Reality (VR) on a rendered planet or terrain, visualizing an atmosphere is very important to archive a realistic look. Because the Earth's sky is perceived every day in a broad range of states, it has to be included in any rendering of the Earth and also in renderings of other planets. During the day, the sunlight is widely scattered inside the atmosphere, causing a bright blue sky. Depending on its consistency, the color of the sky differs from a clear blue to a dusty and dull white, because of different intensities of scattering particles. Also, when the sun rises or falls, the color of the sky changes to orange and red, while the overall brightness of the atmosphere decreases. This also allows for a brief judgement of daytime, depending on the color of the sky.
In human perception, estimating large distance in landscapes is partly based on the *aerial perspective* [Kel77], meaning the atmospheric attenuation between the observer and an object. Because light is scattered away on its way from object to observer, objects far away are, depending on the density of the atmosphere, often perceived dull and with a light blue hue. This effect allows for a relative judging of distances in a virtual environment. Also the color of the sky is an indication of the current daytime and sun position. This in combination with a planet visualized in a Virtual Reality allows for a greater presence in the rendered scene.

1.2 Challenges

Keeping the framerate on an interactive level (above 20 frames per second) is very important to maintain a good immersion in VR and is always a challenge in implementing computergrafic effects. Also the atmosphere should be adaptable to different planets and thus different atmosphere-compositions, since multiple planets are currently being watched and researched.

1.3 Goals

The goals of this work was to provide a completely independent, plugin-like atmosphere renderer, which is capable of displaying a realistic, although approximated atmosphere effect. Having as little impact as possible on the framerates of the underlying planetary renderer is very important, thus the atmosphere rendering has to be performed in real-time. Also, because the dataset used in this project is extracted from satellite images taken from Mars, the atmosphere has to be able to resemble the Martian atmosphere as seen on reference photos. To allow for a dynamic change of planets, the atmosphere has to be easy configurable to adapt to additional planets. It also has to support the usage of VR-Technology, like multi-pipe 3D displays, head tracking and should support the feeling of presence when used in VR.

CHAPTER 2

RELATED WORK

There have already been a number of approaches to explain how the color of the sky develops depending on the sun's position. Also there have been many different approaches and implementations to artificially generate such an effect using computer graphics. These effects have mostly been implemented for an earth-like atmosphere, which is then compared against photos to check for plausibility.

2.1 Atmosphere Visualization

Research on physical scattering of the atmosphere and also on its implementation has been done by Nishita et al. [NSTN93]. Their approach is based on a clear sky atmospheric model, meaning the atmosphere is assumed to only consist of air-molecules and aerosol-like particles. Despite using a precomputed 2D lookup table to improve performance on calculating the scattering-integral, their implementation is not suitable for realtime rendering. Their 2D table contained the attenuation due to scattering of light on its way through the atmosphere at each point. They also simplified the equations such that the direction to the sun can be considered constant because of the great distance between the earth and sun.
Using the same physical model, ONeil [ON05] improved the rendering speed using low sampling of the scattering integral on the GPU. Also he extended the usage of the 2D texture by additionally accounting for the sun-angle, thus completely precomputed one part of the scattering. He then analysed the generated texture and approximated it using a polynomial. With the use of this polynomial the performance was sufficient enough to calculate the remaining scattering integrals in realtime.

Schafhitzel [SFE07] et al. presented a method based even more on precomputation. Their approach is using a 3D texture parameterized by observer height, view-direction and angle of incident sunlight. During runtime the scattering integral is substituted by one texture-lookup per pixel. To emulate atmosphere and planet, two spheres are rendered, one for the planet using backface culling and one for the atmosphere using frontface culling.
Bruneton et al. [BN08] presented a method in which they account for complex scattering from any point in space. Using a 4D Texture, they include view- and sun-direction as well as a view-sun-angle as parameterization. Their goal was to implement a real-time implementation of a multiple-scattering approximation of the atmosphere, enhancing the degree of realism in the atmosphere rendering. Additionally, terrain features like mountains and hills are taken into account and they also present a way to implement light-shafts. Their documentation does however not state on how this effect can be implemented in an existing, arbitrary planetary rendering engine. Also, since the information in this publication is very compressed, a masters thesis is available which implements a similar effect, among other things [Spe11].
Based on the four-dimensional parametrization from Bruneton, although not using multi- but single-scattering, the implementation presented in this thesis is designed as a plugin-like structure to work with arbitrary rendering engines. Also, to adapt this atmosphere to the planet Mars, a new set of parameters is developed and cross-referenced with pictures taken from the Martian atmosphere.

2.2 Terrain Renderer

Most planetary terrain renderers are based on a transformation of heightmap data into a terrain surface. After extruding the geometry it is usually transformed onto a sphere to resemble the geometry of the respective planet. Typical challenges of these renderers are the high volume of data they have to be able to process in order to display a whole planet, and the performance to keep it interactive.
The planetary terrain renderer used as a basis for this atmosphere visualization was developed by Westerteiger [WGH+11] and is also using terrain heightmaps, provided by several Mars-missions. Being able to process huge amount of data is commonly done by organizing the given data into a quad- or octree structure, to improve the performance using a level-of-detail system. To create a very efficient quadtree, the terrain data is mapped onto a sphere using Nasa's HEALPix coordinates [GHB+05]. Using this coordinate system allows for a seamless coverage of the whole planet with equal sized patches, building an optimal quadtree. Also, to allow interactive framerates, this planetary rendering engine is using a level-of-detail system, scaling the observed terrain depending on the distance to the observer. The needed data is extracted from a terrain database and then converted and stored in a special file structure to optimize access speed for each level-of-detail.

Besides the pure height data, the renderer also displays texture data taken by the ESA Mars Express mission in 2004 [1]. These images have a resolution of 10m per pixel but only a partly coverage over the whole Mars is available at the time of writing.
To assist planetary scientist and geologists with analysing the planet's surface, the renderer provides tools to compute the volume of certain areas, slicing through ground or displaying height contours. Also it is possible to set waypoints to fly a certain path, e.g. the entrance route from different Mars-Landers.
Due to its very efficient scalability the implementation can run on modern VR-equipment as well as on normal desktop computers and laptops. An example picture of the planet Mars rendered in said planetary renderer is shown in figure 2.1, already shaded with basic lighting.

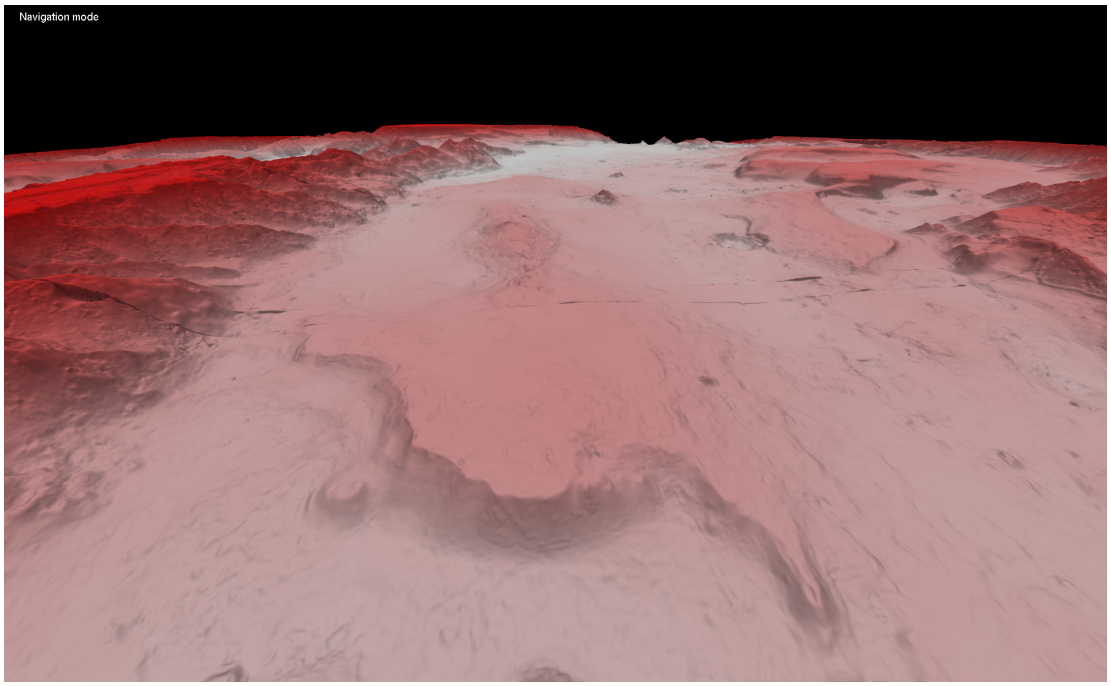

Figure 2.1: Mars as rendered by a planetary renderer without atmosphere

[1] http://www.esa.int/Our_Activities/Space_Science/Mars_Express

CHAPTER 3

BACKGROUND

3.1 Scattering Theory

When light travels through the atmosphere of a planet, it collides with small particles in it. Depending on the size of the particles and the light's wavelength, it has a certain chance of being reflected or scattered away. Because some of the particles the light collides with are air molecules and atoms, different wavelengths of light are affected differently. Short wavelengths, namely blue light, are more affected than longer wavelengths, e.g. red light. When the sun is high at the earth's sky, the light travels a short distance through the atmosphere. Along this short distance, mostly blue and green light is widely scattered in the atmosphere, resulting in an overall blue look of the sky. When looking directly into the sun, the light appears orange or red, because most of the blue and green light components are scattered out of that path into the rest of the atmosphere.
The same explanation holds for the color of the sky at dusk and dawn. Because the light travels its longest path through the atmosphere, blue and green light is almost completely scattered away from the view direction. When the sun rises, the distance of the light travelling through the atmosphere shortens and the color of the sky returns to a pale blue.
But the scattering of light is not just influenced by particles smaller than the wavelengths of the incident light. Bigger particles or aerosols which are about the same size as the wavelength, also influence the color of the sky. Examples for these are small water droplets, dust or smoke, are mostly found in lower regions of the atmosphere. Because their size is bigger than the particles affecting only the blue light, aerosols affect all colors of light about the same. When light e.g. travels through a cloud, all colors of

the light are scattered inside the cloud, making the light shining through very dull, but without a change in color. In bigger cities air pollution (see figure 3.1 [1]) also causes this dusty effect, resulting in a much less saturated view.

Figure 3.1: Air pollution in Beijing

This influence is computed using Mie scattering, which is mostly wavelength independent.
So the effect of light scattering inside an atmosphere is divided into the scattering due to small particles like molecules or atoms and scattering due to bigger particles, so called aerosols. The equations required to simulate a realistic atmosphere using both scattering theories are presented in the following sections.

3.1.1 Rayleigh- and Mie-Scattering

The so called *Rayleigh-Scattering* explains computation of scattering due to small particles (smaller than 10% of the incident lights wavelength [Ray99]). Because these particles have a very small cross-section, the wavelength λ has a strong impact on the resulting scattering. This is due to the higher probability of short wavelengths to collide with a given particle thus resulting in increased scattering for lower wavelengths. When computing the scattering-coefficients which describe the amount of reflection from incident light, the scattering parameter is divided in reflection coefficient $\beta_{R/M}$ (eq. 3.1) and the phase function, indicating the directional characteristic $\gamma_{R/M}$ (eq. 3.2).

[1] http://images.china.cn/attachement/jpg/site1007/20120612/0014222d98501140dd1001.jpg

The corresponding equations for the Rayleigh-Scattering are described in equation 3.1 for β_R and equation 3.2 for γ_R.

$$\beta_R(h, \lambda) = \frac{8\pi^3(n^2-1)^2}{3N_s\lambda^4}\rho(h) \tag{3.1}$$

$$\gamma_R(\theta) = \frac{3}{16\pi}(1 + cos^2(\theta)) \tag{3.2}$$

with

$$\rho(h) = e^{-\frac{h}{H_R}} \tag{3.3}$$

and h being defined as the observer height over ground $h = r - R_{\text{Ground}}$ with r denoting the distance of the observer to the planet's origin and R_{Ground} the altitude of the surface. H_R is the scale height or thickness of the atmosphere if it had uniform density and N_s represents the molecular number density of the standard atmosphere. n is the refraction index of the air, θ is the angle of the incident light and ρ is called the density ratio. As seen in β the smaller the wavelength λ the bigger is the impact on the scattering coefficient.
Mie-Scattering takes account for particles which are of equal or greater size than λ. These particles are commonly referred as aerosol, which cause a dusty blur on objects far away from the observer. Since their size is greater than the wavelength mixture in the daylight, all light gets scattered equally. This results in omitting the wavelength dependency in equation 3.1, resulting in equation 3.4. The phase function (equation 3.5) is a results from the Henyey-Greenstein phase function and was improved by Cornette [CS92].

$$\beta_M(h) = \frac{8\pi^3(n^2-1)^2}{3N_s}\rho(h) \tag{3.4}$$

$$\gamma_M(\theta) = \frac{3}{8\pi}\frac{(1-g^2)(1+cos^2(\theta))}{(2+g^2)(1+g^2-2gcos(\theta))^{\frac{3}{2}}} \tag{3.5}$$

If $g = 0$, this function is equal to *Rayleigh-Scattering*. The asymmetry factor $g(k)$ is given by equation 3.6.

$$g(k) = \frac{5u}{9} - (\frac{4}{3} - \frac{25u^2}{81})k^{-\frac{1}{3}} + k^{\frac{1}{3}}) \tag{3.6}$$

And $x(u)$ is then given as

$$k = \frac{5u}{9} + \frac{125u^3}{729} + (\frac{64}{27} - \frac{325u^2}{243} + \frac{1250u^4}{2187})^{\frac{1}{2}} \tag{3.7}$$

u depends on conditions like haze or dusty air and varies from 0.7 to 0.85 [CS92], assuming an earth's atmosphere. Density ratio also differs from air molecules, so the scale height H_0 has to be set accordingly [NSTN93].

3.1.2 Optical Depth

Since now both scattering equations have been presented, the computation of their influence on the light intensity is now discussed. With $\beta_{R,M}$ representing how much light intensity is lost due to the scattering, we can now calculate the attenuation of light inside the atmosphere. This attenuation a is independent of the light direction thus independent from $\gamma_{R/M}$ and just describes how much light is reaching the observer's eye from any point in the atmosphere. To compute a for an arbitrary ray inside the atmosphere, this view-vector is traced and the respective values of $\beta_{R,M}$ are summed up and integrated along the ray. Therefore, the attenuation over the distance d of the ray r is computed using equation 3.8 with $h(r)$ denoting the height at every point along the ray r.

$$a(\lambda, r) = \int_0^d \beta_R(\lambda, h(r))\beta_M(\lambda, h(r))\mathrm{dr} \tag{3.8}$$

Equation 3.9 converts the attenuation coefficient a into the extinction factor, here referred as transmittance t. This transmittance is defined as a percentage of incident light from the start point of the ray r, reaching the end of the ray.

$$t(\lambda, r) = e^{-a(\lambda,r)} \tag{3.9}$$

3.2 Visualization Models

Based on the Rayleigh- and Mie-Scattering now a formulation to compute the light intensity inside the atmosphere, also presented by Bruneton [BN08], is developed. This formulation then allows to compute the light intensity at any arbitrary but given point, depending on the parameters of the atmosphere and the incident light.
Starting with the total light intensity reaching an observer's eye when situated inside the atmosphere is described in equation 3.10.

$$I_{\text{total}}(x, v, d) = (I_{\text{incident}}(x, v, d) + I_{\text{reflected}}(x, v, d) + I_{\text{inscatter}}(x, v, d)) \tag{3.10}$$

An observer at an arbitrary point x inside the atmosphere looking in the direction v perceives the light intensity I_{total}. The view-ray r in the view-direction v travels the distance d through the atmosphere and either hits the boundary of the atmosphere at p or the planet at p' (see figure 3.2). The resulting intensity is a combination of the direct light from the light source, in this case the sun, reflected light from the surface of the planet and the intensity of inscattered light.

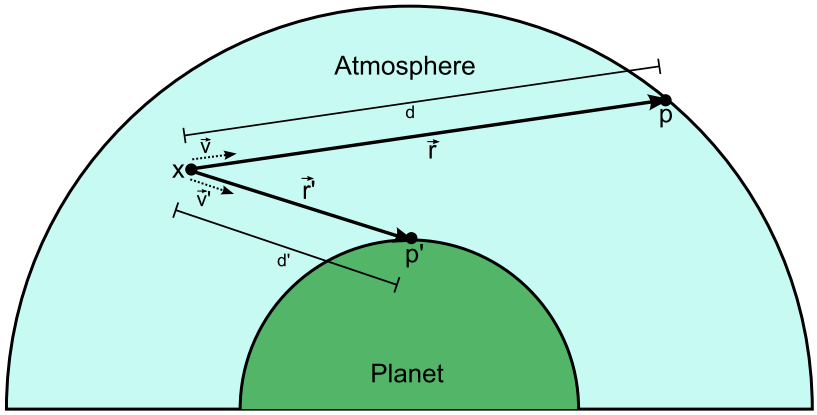

Figure 3.2: Planet - View-vector configurations from arbirtrary point x

Incident light is the direct illumination from the sun to the observer positioned at x. From the light's entry point at the top of the atmosphere to x it gets attenuated along its path. Inscattered light is a representation of light scattered into the view-ray r from other rays inside the atmosphere. Reflected light from the surface has only to be taken into account for the case that the view-ray is intersecting with the planet. Formulations for the three illumination factors are presented in the following sections.

3.2.1 Attenuation of incident Light

Incident light from the sun reaching the observer directly is the most straight forward computation. With the intensity I_0 when the light hits the top of the atmosphere, the intensity I_{incident} is calculated using the transmittance t.

$$I_{\text{incident}}(x, v, d) = t(x, v, d) \cdot I_0 \tag{3.11}$$

The wavelength λ dependency of the transmittance is constant and only depending on a choice at the start of rendering. To improve readability λ is henceforth ignored and not included explicitly in the parameters of β and all parameters depending on β. Is the sun occluded by e.g. the planet, thus no direct sunlight is reaching the observer, $I_{incident}$ is 0. Due to this computation being very simple, it can be applied at runtime and hence is not precomputed.

3.2.2 Reflected Light

Is the view-ray intersecting with the surface of the planet, the reflected light at that surface position has to be taken into account. Light reaching the planet's surface, commonly called *irradiance* is assumed to be reflected and not absorbed. The light reaching the intersection point p on the planet's surface can be calculated with an integration over the semi-circle around p as shown in figure 3.3.

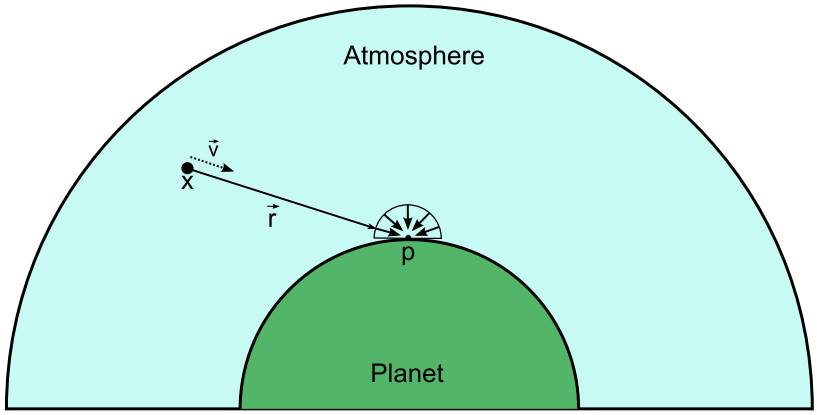

Figure 3.3: Reflected light at planet's surface

From there the reflected light is again attenuated until it reaches the start point of the view-ray, being the observer's eye (equation 3.12).

$$I_{\text{reflected}}(x, v, d) = t(x, v, d)\frac{\omega(x)}{\pi}\int_0^{2\pi} I_{\text{total}}(x, \theta, d) \cdot n(x)\mathrm{d}\theta \tag{3.12}$$

In this equation n is the normal vector at the surface point and $\cdot$ representing the scalar product. ω is the reflectance of the surface, described by a factor between $0 \ldots 1$ and dividing by π normalizes the integration over the semi-circle.
The integral can be simplified up unto the point where the only light source illuminating the planet is assumed to be the sun. Then the integral is completely omitted and the light intensity reflected from the planet's surface is a simple dot product between the normal of the surface point and the incident light direction multiplied with the lights intensity. Although this reduces the quality of the reflection, the simplified version can be calculated at runtime. This simplifcation is shown in figure 3.4. Depending on the

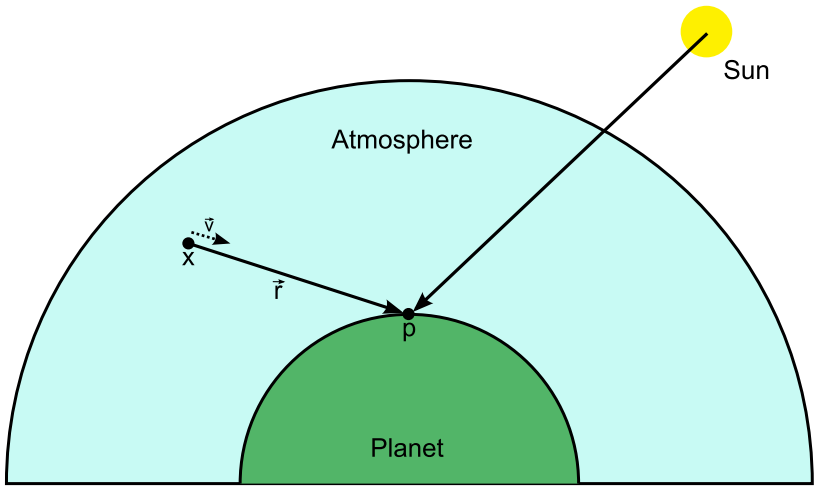

Figure 3.4: Simplification of the reflected light

planetary rendering engine providing the platform to the atmosphere visualization, the planet might already been shaded. This reduces the calculations of the reflection to two attenuation calculations and one texture lookup on the previously shaded planet. Else the normals of the planetary surface have to be either calculated in runtime or have to be provided by the planetary renderer.

3.2.3 Inscattered Light

The light from the sun to the observer is not just scattered away from the view-ray but also scattered into the view-ray. This inscattered light can be computed at each point of the view ray using an integral over a sphere around that point (figure 3.5). Solving this integral yields the total light intensity reaching this single point in the atmosphere due to inscattering from all possible directions. To get the total light intensity from the view-ray, we have to integrate over the different intensities at each point along the ray. After proper attenuation, the light intensity reaching the observer's eye is a result of equation 3.13 and 3.14 .

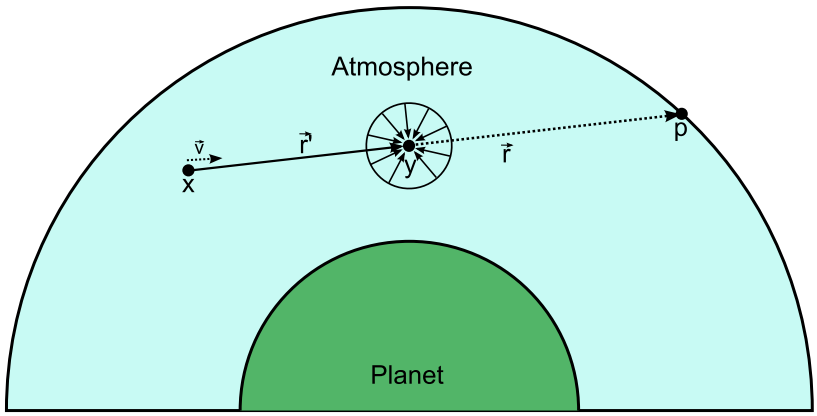

Figure 3.5: Inscattered light at arbitrary point

$$I_{\text{inscatter}} = \int_x^p t(\lambda, r')j(y, v)\text{dy} \tag{3.13}$$

With r' being the vector from x to y.

$$j(y, v) = \int_0^{4\pi} (\beta_R(y)\gamma_R(v \bullet \theta) + \beta_M(y)\gamma_M(v \bullet \theta)) \cdot I_{\text{total}}(y, v', d')\text{d}\theta \tag{3.14}$$

The total light intensity $I_{\text{total}}(y, v', d')$ is a result of sunlight reaching y after passing the distance d' through the atmosphere. Computing the integral over the sphere can again be simplified, just like the reflection of the planet's surface, with only accounting for direct sunlight (figure 3.6). So instead of integrating over the whole sphere around an arbitrary point, just the direct sunlight reaching each point is taken into account. This again results in omitting the integral and hence in loss of brightness along the ray.

3.2.4 Multiple Scattering

As already mentioned in the equations for reflection and inscattering, computation of the light intensities can be simplified by omitting the integration around the sample points. The reduced formulation is called single-scattering, the one including the spherical integration is called multiple-scattering. Bruneton [BN08] presents one of the first realtime multiple-scattering implementations with the usage of special precomputation. This involves solving the inscatter and reflection computations multiple times to obtain the correct values. Typical about three iterations over the scattering equations suffice for good results. Multiple-Scattering is most noticeable during twilight. It brightens up the sky due to scattered light not being discarded but accounted for in computation. During the day, the distances of the sun are already very short in comparison to the distances

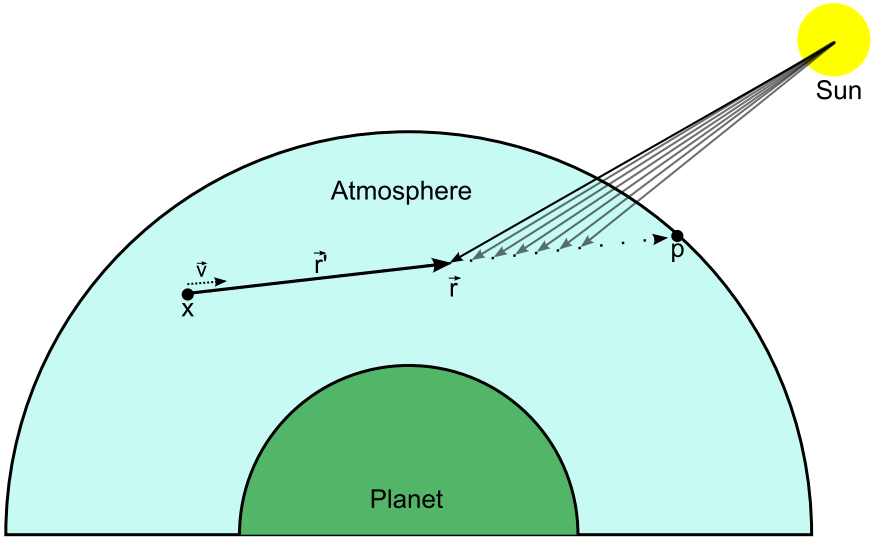

Figure 3.6: Simplification of inscattered light

at dusk and dawn, so the difference is barely noticeable. Multiple-scattering is not included in this project although the implementation can easily be extended by adding the iterative scattering equations after the already implemented single-scattering.

CHAPTER 4

IMPLEMENTATION

With the physical aspects of atmospheric scattering covered, the remaining question is on how to implement this effect. Assuming a scenario using an already implemented terrain rendering engine, where the observer is situated somewhere inside the potential atmosphere looking in an arbitrary direction. When rendering that scenario, all equations can be solved by obtaining what is already known in that scene: the observer position, its view-angle and the angle to the sun. So a straight forward implementation would now just shoot a ray into the scene, sampling and calculating equation 3.10 at each sample point. While this would yield a correct result, due to the ray casting it is very expensive to compute, especially for high sample density and high screen resolutions. Because one of the goals is to maintain interactivity, this straight-forward approach is not viable in this scenario.
As already mentioned, there have been implementations with sampling at runtime by ONeil by simplifying the optical depth with an analytical formulation [ON05]. Another way, allowing for realtime rendering of said atmosphere was presented by Schafhitzel et al. [SFE07]. Their approach was using a precomputed scattering table to relieve the realtime computation from solving the needed integrals for each frame and each view-direction. However, their implementation was based on rendering additional geometry, spheres, into the scene. Because a goal of the work presented in this thesis is to create an universal extension to any already existing planetary rendering, drawing additional objects into the scene is not an option. Inspired by Bruneton's example code [BN08] which is only using an aligned quad in screen space to visualize the atmosphere, we present an implementation on how this effect can be set up as a post-processing effect. Also we use a similar approach on how to generate the lookup textures using the GPU and multiple texture layers.
Acessing the data from a previously rendered scene, on which we assume to not have any influence, is done with the usage of *deferred rendering*.

4.1 Deferred Rendering

Deferred rendering, as used e.g. in modern computer games engines [1] in the form of *deferred lighting* or *deferred shading*, is based on a concept of splitting the shading process into two steps [ST90]. In the first step, all information of the scene is computed, collected and saved into several buffers or textures. The collected information can be anything from surface normals and depth information to basic colors and effects like blur or bloom. The second step then performs the deferred shading, using only the previously saved information to shade the entire scene. This decouples operations like lighting from the scene itself and allows for e.g. expensive calculations of multiple light-sources. Also it simplifies the access to vertex and object data like normals and the z-Buffer of the scene, while at the same time reducing the amount of computation because only visible data is processed. The usage of deferred shading in this atmosphere visualization is based on rendering into *Framebuffer-objects* [2] and therefore requires Shadermodel 3.0.

4.2 Atmospheric Scattering as a post-processing Effect

The first preparation to be made is setting up the precalculation and rendering all scattering evaluations into lookup textures. Data needed for the precomputation is very minimal and mostly independent of the rendering beforehand. Parameters to choose are both scale heights $H_{R,M}$, planet and atmosphere radius $R_{planet,atmosphere}$, $\beta_{M,R}$ and $\gamma_{M,R}$. Only planet and atmosphere radius have a connection to the already rendered planet, so they have to be fit accordingly. Thus the precomputation can already be performed at this point, without the need for any additional information.
For the rendering of the atmosphere however, these informations only suffice when the planet is assumed to be a perfect sphere. When the planet has terrain features like mountains, hills or canyons, this data has to be passed into the atmosphere lookup shader. As already described in section 4.1, this information can be accessed using the techniques used in deferred rendering. Both color and depth information of the previously rendered scene of the planet can be copied into a buffer or texture. The two resulting textures then hold all information needed to reconstruct the terrain geometry of the planet during the atmosphere rendering.
To take advantage of fast GPU's, both the atmosphere precomputation and rendering is done inside a fragment shader. How the shaders operate to provide the corresponding lookup textures for the final rendering is described in the following sections.

[1] http://en.wikipedia.org/wiki/Deferred_shading#Deferred_shading_in_commercial_games
[2] http://www.opengl.org/wiki/Framebuffer_Object

4.3 Generating the lookup Textures

As mentioned above, the calculation of all scattering and ray integrals during runtime is too expensive to allow for interactive framerates. Precomputation solves this performance issue and since it has to be run just once, has no impact on the framerate at runtime. But to allow precomputation, simplifications have to be made, to reduce memory storage. Without reducing storage complexity, every point in the entire atmosphere and all possible view-directions for each point would have to be computed and saved. Even with a coarse sampling and good interpolation between sample points, this would produce too much data volume to fit onto the GPU. Also the size of the needed data would increase with the size of planet and atmosphere.
A very efficient way to reduce data volume is the simplification to approximate the planet with a perfect sphere. Due to the rotation symmetry, a point can now just be referred by its altitude. Also the view-angles reduce to a 2D problem, again because of rotational symmetry. Of course, when visualizing a planet which is not a perfect sphere, this approximation produces certain errors which have to be fixed in runtime.
Given this assumption, precomputation becomes straight forward due to the very little parameters needed. With a set of possible observer position parameterized by an altitude and a 2D view-direction, a 2D texture can directly be generated. As with the physical representation itself, the precomputation is split into three different shaders : *transmittance*,*irradiance* and *inscatter*. Before discussing their implementation, a brief explanation of the used ray-sphere intersection algorithm is given.

4.3.1 Ray-Sphere Intersection

For generation of the different textures, a simple ray-sphere intersection test is necessary to compute the distance that the light travels through the atmosphere. Due to the rotation symmetry of a perfect sphere, the intersection algorithm can be projected into 2D space. The here presented algorithm is published by [AMMH02]. Figure 4.1 shows the configuration for the 2D circle-ray intersection.

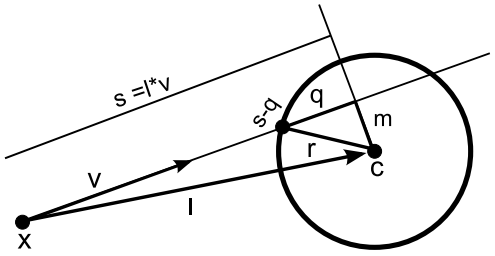

Figure 4.1: 2D circle-ray intersection

Information given to compute whether or not an intersection appears, contains viewer

position x , the view-direction v and the planet's radius R_{ground}. First, the distance l from x to the center of the sphere c is calculated for the case that c is not lying in the origin. Then the projection s from l onto v is computed using equation 4.1.

$$s = l \bullet v; \quad l = x - c; \tag{4.1}$$

Is $s < 0$ and x is outside the sphere then the view-ray is not intersecting the sphere. Otherwise the distance from the projection to the center of the sphere can be computed using the Pythagorean theorem (eq. 4.2)

$$m^2 = l^2 - s^2 \tag{4.2}$$

If $m^2 > r^2$, the ray is missing the sphere, else it is guaranteed to hit it.
Then the two intersection points with the sphere can be computed using equation 4.3 with q being the distance from m to the sphere's radius.

$$q^2 = r^2 - m^2 \tag{4.3}$$

The square root of q^2 can be computed because of m being smaller or equal than r. So the two intersection points are given with $t = s \pm q$. Applied on this scenario, the smaller $t = t_1$ yields the distance to the atmosphere and $dt = t_2 - t_1$ the distance travelled inside the atmosphere until the ray hits the outside again.

4.3.2 Transmittance Texture

Recall the formulation of the *transmittance* (equation 3.9). Transmittance is the percentage of light reaching the observer's eye after travelling a certain distance through the atmosphere. This information is needed for any light intensity calculation and thus rendered into its own lookup texture.
All relevant angles for the generation of the textures are shown in figure 4.2. To reduce computation in the later stages, the angles are not stored directly but as their corresponding cosine:

$$\mu_{viewZenith(vz)} = \cos(\alpha); \quad \mu_{sunZenith(sz)} = \cos(\omega); \quad \mu_{viewSun(vs)} = \cos(\theta); \tag{4.4}$$

For the transmittance only the distance travelled through the atmosphere is relevant, thus we can take advantage of the simplification mentioned above. Due to the nature of a perfect sphere, a ray shot from an arbitrary altitude with a fixed angle to the zenith (90° up) has always the same distance to the outer boundary. So to precompute a lookup

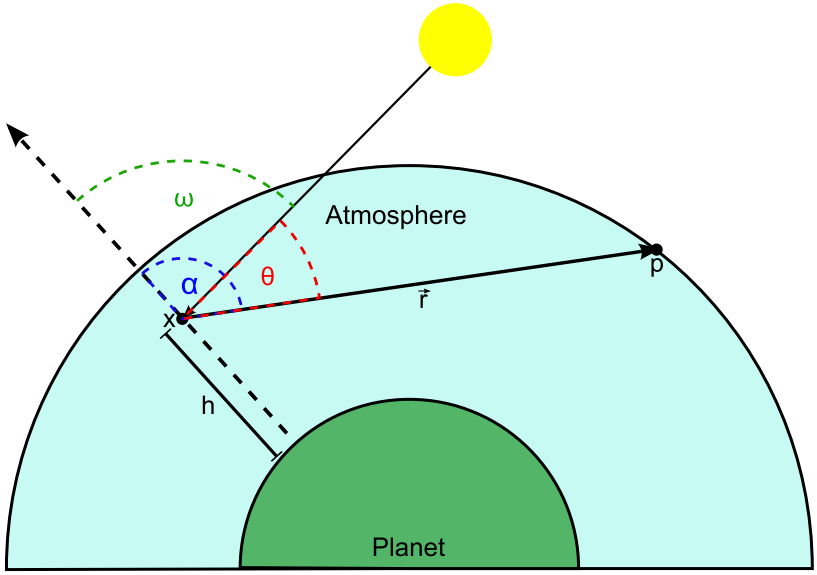

Figure 4.2: The three angles needed for precomputation and rendering

texture for the transmittance, its value has to be computed for each altitude from planet radius to atmosphere radius and each view-angle to the zenith.
Solving the equation, involves approximating the integral and the computing of $\beta(\lambda, h)$ at each point along the ray. λ is previously defined and h as the observer's altitude above ground (figure 4.2) is given for the starting point of the ray and has then to be recomputed for each sample (see equation 4.5).

$$h_x = \sqrt{h^2 + x^2 + 2 \cdot x \cdot h \cdot \cos(\alpha)} \tag{4.5}$$

x is the current distance to the starting point and α is the angle from ray-direction to the zenith at the current sample point. With the thus computed $\beta(\lambda, h)$ for any sample point along the ray, the integral can be solved using trapezoidal integration.

Solving the integral is done with trapezoidal integration (equation 4.6) [DR06]

$$\int_a^b f(x)dx \approx (b-a)\frac{f(a)-f(b)}{2} \tag{4.6}$$

To increase performance, β can be split up into the constant part depending only on λ and ρ depending on the height. Computing the transmittance for a point x looking in direction v is done by first calculating the distance to the outer boundary of the atmosphere using the ray-circle intersection algorithm in 4.3.1. That distance is then divided by the number of desired sample points and the result used as a step width dx. After computing β at the starting position, we loop over the number of sample points, moving along the ray with dx. Is the last sample point reached, we multiply the result with the constant part of β and add the computation result of the corresponding Mie-scattering. Mie-scattering is computed exactly the same way but with other parameters in β.

Now that we can calculate the transmittance for any arbitrary point and view-direction, this has to be done for all possible altitudes above the planet looking into all directions. To take the advantage of the high computation speed of modern GPU's, these calculations are entirely performed in the fragment shader. Because the fragment shader is called for each pixel on the screen, it already provides us with a discretization of the altitude and the view-angle, given as the current pixel coordinate. The pixel position can be normalized into the range of $0 \ldots 1$ and can then be transformed into a respective height or angle. This can either be done using a straight forward linear mapping such as equation 4.7 for the altitude or equation 4.8 for the cosine of the view-zenith angle or in a nonlinear mapping.

$$r(v) = R_{\text{Ground}} + (v) \cdot (R_{\text{Atmosphere}} - R_{\text{Ground}}) \tag{4.7}$$

$$\mu(u) = -0.15 + u \cdot (1 + 0.15) \tag{4.8}$$

u and v are the respective normalized pixel coordinates. Translating the cosine of the view-angle by 0.15 radians (8°) into negative direction allows for α up to 98°. An angle greater than 90 degrees is possible when the observer is situated above the planet and looking at the side of the planet, as displayed in figure 4.3. Is the angle even greater than the used 98°, the attenuation coefficient doesn't change as much as it does between smaller angles (figure 4.5). So it is sufficient to clamp the texture at a certain angle and use a constant value for greater angles.

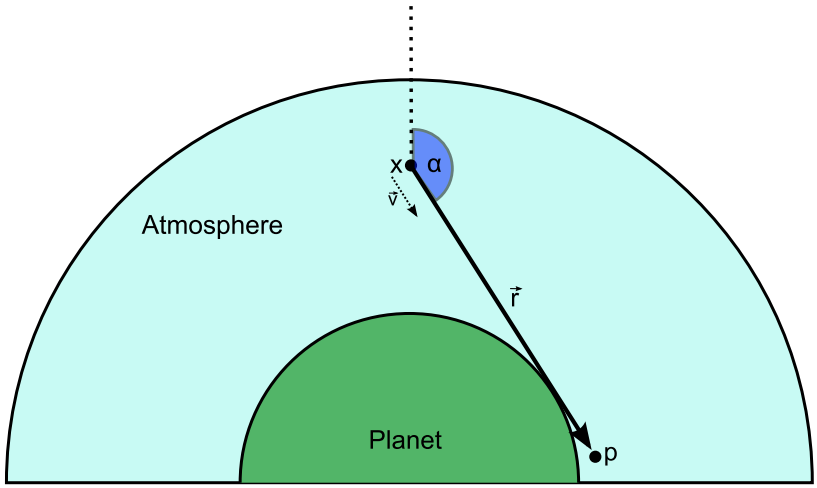

Figure 4.3: Example for an angle greater than 90°

Although this already yields good results, Bruneton [BN08] presents a nonlinear mapping, to increase precision in areas where the change in transmittance is most noticeable. The change of transmittance directly above the observer with α close to 0 can be sampled coarse, due to the short distance the light travels through the atmosphere. This

observation results in an alternative parameterization for the altitude from eq. 4.9 and the cosine of the angle from eq. 4.10. The arccos of both the linear and nonlinear parameterization are shown in figure 4.4

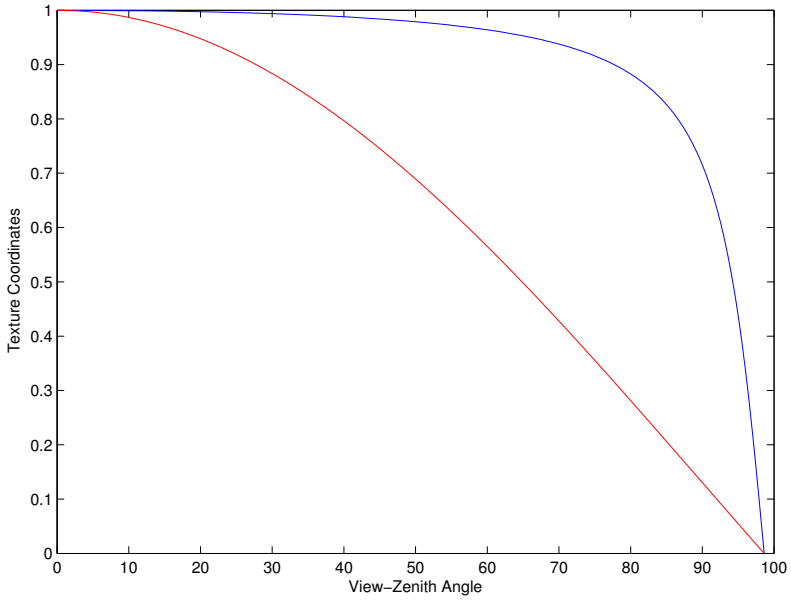

Figure 4.4: Arccos of the two linearizations: Red: linear, Blue: nonlinear

$$r(v) = R_{\text{Ground}} + (v^2) \cdot (R_{\text{Atmosphere}} - R_{\text{Ground}}) \tag{4.9}$$

$$\mu(u) = -0.15 + \tan 1.5 \cdot u \cdot \frac{1 + 0.15}{\tan(1.5)} \tag{4.10}$$

Due to the fact that the change in transmittance can be neglected for angles close to the zenith, α is clamped after reaching 1.5 rad. Using this fragment shader results in a texture which can be used for all further calculations and for the realtime rendering. To look up the correct transmittance value in that texture requires to apply the inverse of this parameterization on the actual height or angle to receive the normalized pixel coordinate. With that the texture lookup can be performed and the value can be read. Because β is set as a three dimensional vector, a lookup will return this vector and thus the attenuation of the light. The size of the texture can be chosen arbitrary but a texture size of 256x64 as used in [BN08] is sufficient. Figure 4.5 shows the lookup texture for the transmittance, after it's been computed.

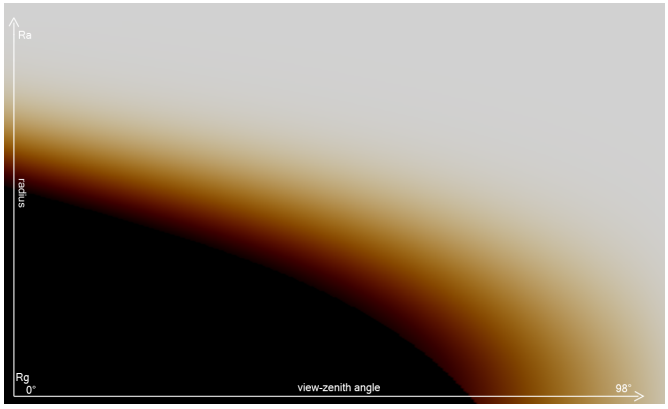

Figure 4.5: Transmittance lookup texture

4.3.3 Irradiance Texture

Irradiance is caused by reflected light from the ground's surface. Is the sun the only light source in the scene, hence only single scattering is taken into account, the irradiance formulation can be simplified. Instead of integrating over the hemisphere at each surface point, the reflected light can directly be calculated by a single dot product. This dot product between the light- or sun-direction and the surface normal can be calculated in realtime. After the reflected light is computed it still has to be attenuated by the atmosphere.
Since the given terrain of the planetary rendering engine does not include normals and the planet is already shaded, the irradiance in this case is equal to the surface texture. Due to deferred shading, the surface of the planet is already shaded and available in a texture. The color values at each pixel have to be attenuated from their position to the observer, as seen in figure 4.6 in contrast to figure 4.7, where the surface is not attenuated.

4.3.4 Inscatter Texture

Recall the inscatter coefficient, representing the light at each sample point scattered into the current view-ray. This is also a fixed value for each sample point along said view-ray and thus can be precomputed as previously done with the transmittance. Recall the inscatter equation 3.13 and the integral formulation from equation 3.14. The spherical integral around each sample point is then omitted when only taking single-scattering into account. Simplifying the equation results in a new formulation for the light intensity at point x looking into direction v:

$$j(x, v) = \sum_{i \in R,M} \beta_i(x)\gamma_i(\theta)I_{\text{total}} \tag{4.11}$$

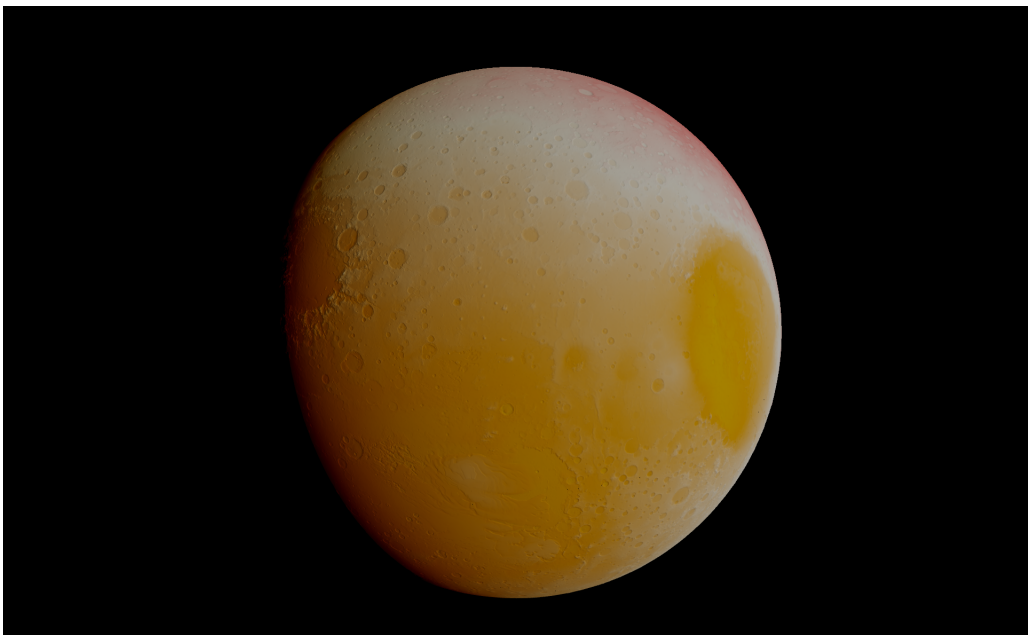

Figure 4.6: Attenuated color of the planet's surface

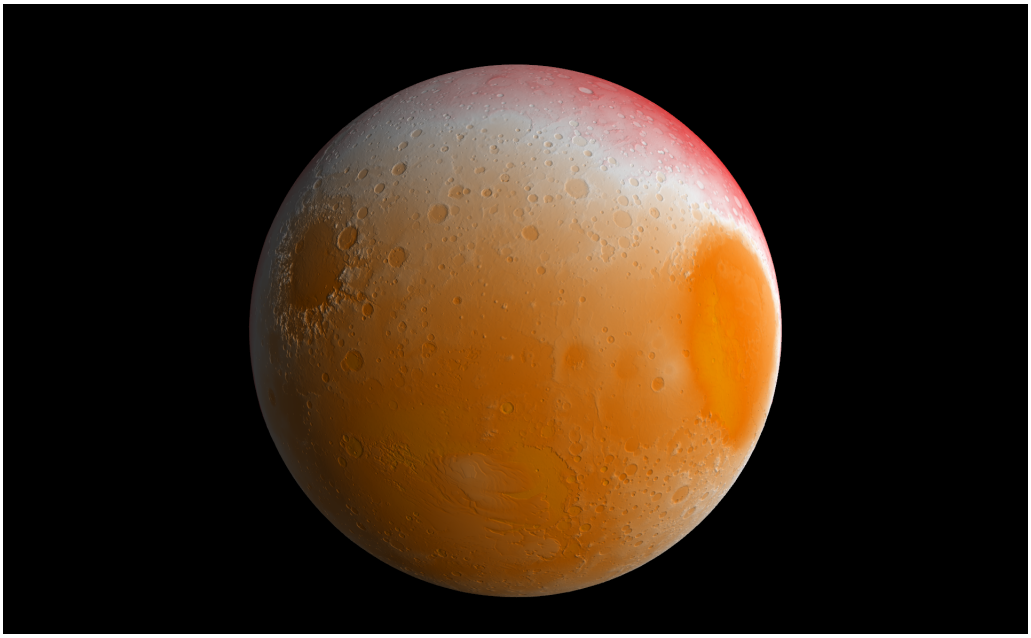

Figure 4.7: Color of the planet's surface

As with the transmittance calculation, $\beta_{R,M}(x)$ is partially precomputed and can be directly inserted into the calculation. The not-constant part of $\beta_{R,M}(x)$, namely ρ (eq. 3.3), is only depending on the altitude of the current point. Phase function $\gamma_{R,M}(\theta)$ is not included in the precalculation and is instead applied at runtime. This is because the

resolution of the created texture is too coarse, resulting in insufficient angular precision. Because the phase functions are fast to compute and θ constant for each ray, applying them at runtime is viable.
Therefore, $j(x, v)$ can be computed for each sample point along the view-ray assuming knowledge about its altitude. Integrating over said view-ray, as done in equation 3.13 can then be performed similar to the transmittance precomputation. First, the distance of the view-ray r is computed using a view-angle α and a starting altitude h_x. This ray is then sampled with an arbitrary amount n of sample points. Usually, for the presented implementation, 50 sample points are sufficient. $j(x, v)$ is then computed at each sample point. To further improve performance, the whole integral along the ray can be written as:

$$I_{\text{inscatter}} = I_{\text{total}} \cdot \beta_{R,M\backslash\rho(h)} \cdot \int_x^p t(\lambda, r)\rho(h)\text{dy} \tag{4.12}$$

At each point just $\rho(h)$ and $t(\lambda, r)$ have to be computed and summed up. All other variables are independent of the position and can be applied after the integration.
The light from one of the sample points reaching the observer has to be attenuated twice. Once for the distance from the atmosphere boundary to the sample point and once from the sample point to the observer. To obtain the correct transmittance from the observer to the sample point on the ray, first the transmittance from the observer to the atmosphere boundary is computed. Then the transmittance along the same view-ray, but starting from the position of the sample point to the boundary of the atmosphere, is calculated. Due to the exponential nature of the transmittance, the resulting transmittance is computed by dividing the two (equation 4.13).

$$t_{\text{res}} = \frac{t_{\text{observer}}}{t_{\text{sample point}}} \tag{4.13}$$

The transmittance from boundary to sample point is, due to single-scattering, a simple texture lookup with sample point altitude and a ray to the sun. Multiplying the transmittance from atmosphere to sample point and from sample point to observer then yields the correct attenuation for the intensity reaching the observer. Because all possible transmittance values are already computed and stored into a texture, one lookup for each value is sufficient. So to compute the correct transmittance for each sample point, three texture lookups have to be performed. The angle between sun and zenith,needed for the lookup, has to be recomputed for each sample point (eq. 4.14). The same applies to the altitude h, which is calculated using the same equation as applied in the transmittance calculation (eq.4.5).

$$\mu_{szi} = \frac{\mu_{vs} \cdot d + \mu_{sz} \cdot \text{h}}{\text{h}_i} \tag{4.14}$$

The altitude h needed for ρ is calculated using the previously computed h and the corresponding scale height $H_{R,M}$. Thus both ρ and t_{res} can be computed using a loop over

all sample points and summing up both Rayleigh and Mie components.
To account for details like the shadow of the planet itself, a check is performed at each sample position. After checking if the current angle between sun and zenith is greater than the angle to the horizon, samples which don't pass this test are discarded. Approximating the integral is again done by using trapezoidal integration between the sample points (eq. 4.6).
Just like the transmittance, also the inscatter has to be computed for all possible combinations of the input parameters. But in contrast to the transmittance, the inscatter equations depend on the view-zenith, sun-zenith and view-sun angles, as well as the height of the sample point. Both view-zenith (μ_{vz}) and view-sun(μ_{vs}) are easy to parameterize in a linear fashion:

$$\mu_{vz/vs} = -1.0 + 2.0 \cdot u \tag{4.15}$$

So both parameters are scaled from 0° to 180° to account for all possible angles. The sun-zenith angle has, because of the relative sun position, no need for full 180° closure and is scaled for a smaller range:

$$\mu_{sz} = -0.2 + u \cdot 1.2; \tag{4.16}$$

The parameter $\mu_{vz/sz/vz}$ represent the cosine of the actual angle α and the altitude h is parameterized using eq.

$$h = \sqrt{(R^2_{\text{Ground}} + u \cdot (R^2_{\text{Atmosphere}} - R^2_{\text{Ground}}))} \tag{4.17}$$

Similar to the transmittance, Bruneton gives an alternate parameterization to optimize precision in areas which need high resolution. The view-sun angle has no special non-linear version, but μ_{sz} changes to equation 4.18.

$$\mu_{sz} = \frac{\tan(2.0 \cdot u - 1.0 + 0.26) \cdot 1.1}{\tan 1.26 \cdot 1.1} \tag{4.18}$$

Using a linear mapping on the view-zenith angle produces visible artefacts, because the corresponding length of the view-ray is discontinuous. This is due to the nature of the horizon, since the ray gets shortened the instant it intersects with the planet instead of the atmosphere [BN08]. To fix this error, Bruneton suggests a discontinuous mapping of the parameter itself, having the discontinuity at 90°. Is the angle > 90°, the following equation yields the correct μ_{vz}:

$$\mu_{vz} = \frac{(R^2_{Ground} - h^2 - u \cdot \sqrt{h^2 - R_{Ground}}}{(2.0 \cdot h \cdot u \cdot \sqrt{h^2 - R_{Ground}})} \tag{4.19}$$

otherwise the equation is changed to:

$$\mu_{vz} = \frac{(R^2_{Atmosphere} - h^2 - (\sqrt{r^2 - R^2_{Ground}} + \sqrt{R^2_{Atmosphere} - R^2_{Ground}})^2}{(2.0 \cdot h \cdot (\sqrt{r^2 - R^2_{Ground}} + \sqrt{R_{Atmosphere} - R^2_{Ground}}))} \tag{4.20}$$

In all equations, u one of the texture axes scaled from 0 to 1, already hinting the same texture scaling as applied in the transmittance precomputation. r represents the current height sample, also scaled linearly from 0 to 1, 0 meaning the current sample is on the planet's radius and 1 meaning the sample is located on the atmosphere boundary. In contrast to the transmittance, the inscatter texture would need four dimensions to account for all possible combinations of the angles and altitude. To map this 4D texture onto a 3D texture, the view-sun angle and the sun-zenith angle are combined into one texture axis. So instead of creating one 3D texture with one parameter on each axis, we put multiple of these 3D textures next to each other [BN08]. This is naturally limited by the quickly growing size of the texture, due to the additional dimension. Fortunately, a resolution of 8 is sufficient for the view-sun angle and thus yields the final inscatter texture. As seen in figure 4.8, the eight panels with different view-sun angles are evenly distributed over the global x-axis. Each of the single panels has the sun-zenith angle on their respective x-axis and the view-zenith angle on the y-axis. The altitude is sampled in 32 samples, mapped onto the range from 0 to 1 using equation 4.17.

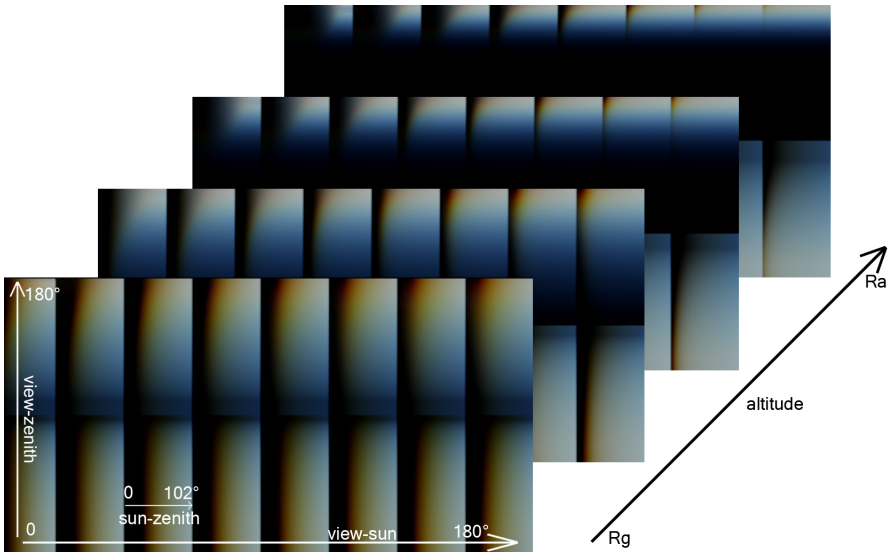

Figure 4.8: Inscatter texture sampled at 0.25 / 0.5 / 0.75 and 1.0 altitude

Figure 4.8 shows Rayleigh scattering only, but the result of the Mie scattering is build in a similar fashion and stored in a seperate texture. Should the space on the GPU be too limited to create two of these four-dimensional textures, the Mie-Scattering can be reduced to its red-channel and later be reconstructed, as suggested by [BN08].

4.4 Applying the atmospheric Effect

Up until this point, the whole precomputation and all assumptions are, except for the parameters, completely independent from an underlying planet renderer. To maintain the independence, first we show how to implement and render the atmosphere in a simple, standalone scenario, before including it into an existing rendering engine.

4.4.1 Rendering a simple Atmosphere

Having both transmittance and inscatter lookup textures computed, we are ready to put these to use and display a realistic atmosphere. Recall that the four parameters of the inscatter texture are the altitude, view-zenith, sun-zenith and view-sun angle. So in order to perform correct lookups in the inscatter texture, these parameters have to be extracted from the scene.
Assuming the scene is a simple 3D-space with the planet's origin laying in the origin of the coordinate system. This allows us to calculate the height of the observer relative to the planet's spherical surface as the length of its position vector. Without accounting for collision, we assume the observer to always be positioned above the planet's surface and inside the atmosphere, so $R_{\text{Ground}} < r < R_{\text{Atmosphere}}$. This way we already got the first parameter for the lookup in the inscatter texture, namely the observer's altitude. The sun-angle is the result of a dot product of the light-direction l vector with the current position vector of the observer:

$$\mu_{sz} = l \bullet x \tag{4.21}$$

While both altitude and sun-zenith angle are constant for the whole frame, the view-ray from the observer into the space, differs with each pixel. Hence the view-angles also differ from pixel to pixel, thus they have to be computed for each pixel in the fragment shader. Obtaining the correct view-angle for any pixel can be done by casting a ray from the observer's position into the scene, in the direction of the far-plane.

4.4.2 Obtaining the Far-Plane

The far-plane is the backplane of the view-frustrum, thus everything behind this plane is omitted during rendering. After model-view and projection transformations are applied on the scene, the far-plane is screen aligned and has the z coordinate 1. Thus to get the world-coordinates of the edges of the far-plane, we set an imaginary viewport with the dimensions [0,1]x[0,1] and then create four imaginary vertices with the coordinates:

$$v_1 = [0, 0, 1]; \quad v_2 = [1, 0, 1]; \quad v_3 = [0, 1, 1]; \quad v_4 = [1, 1, 1]; \tag{4.22}$$

Assuming these are the edges of the far-plane after the transformation, we apply the inverse of the model-view-projection transformation and thus obtain the world-coordinates of the far-plane edges. Computing the vector from the observer to each corner yields the view-direction of the camera in world space

4.4.3 Preparing the Fragment Shader

To be able to actually render the atmosphere with the information at hand, we recall the idea of *deferred rendering* (section 4.1). A simple screen-space aligned rectangle, quad, is used as a canvas and assign the previously calculated world-coordinates of the far-plane edges as a parameter to each edge-vertex. When passing this coordinate through the vertex- into the fragment shader, the values are interpolated on the GPU and thus we can compute a per-pixel correct view-direction from the observer to the specific pixel in world coordinates. For later usage, also the position of each vertex is passed into the fragment shader, where it is interpolated again, to receive the 2D pixel position in the range of [0,1]x[0,1].
With these calculations already processed and two of the four needed parameters calculated, the two missing parameters can be easily calculated in the fragment shader. As already mentioned, the direction is the normalized difference between the current pixel's world coordinate and the observer position. The two missing angles μ_{vz} (view-zenith) and μ_{vs} (view-sun) can now be easily computed using a simple dot product:

$$\mu_{vz} = x \bullet v; \quad \mu_{vs} = x \bullet l; \tag{4.23}$$

With x representing the observer's position, v being the normalized view-direction and l the direction to the sun. As μ_{vz} and μ_{vs} represent the cosine of the actual angle, these calculations yield the correct values.
These two angles complete the collection of parameters needed for a successful lookup in the inscatter texture. Accessing the correct texel of the inscatter texture is done by applying the reverse parameterization from the precomputation step. Since the view-zenith angle was parameterized in a discontinuous way, this has to be taken into account when reverting the calculation. Due to the low resolution of the view-sun angle μ_{vs}, a better result is obtained when linearly interpolating between the calculated lookup coordinate x and $x + 1$.
Applying all computations and performing the texture lookups then result in two rgb-color values per pixel, one for Rayleigh- and one for Mie-Scattering. Recall that the phase functions $\gamma_{R,M}(\theta)$ were left out of the precomputations because of the limitations in precision. Both parameters can now be calculated using μ_{vs} and then multiplied with the inscatter values.
All computations so far base on the assumption that the observer is situated inside the atmosphere, such that $R_{\text{Ground}} < r < R_{\text{Atmosphere}}$. This however cannot be assumed to be always the case, especially because a view from space should be possible and should yield a realistic image.

4.4.4 View from outer Space

When looking at the planet from outer space, the distance the view-ray travels through the open space is not affected by any attenuation. So if the observer is situated outside the atmosphere and the view-ray intersects with the atmosphere, the result on the attenuation is the same when the observer is moved along the view-ray on top of the atmosphere boundary. Again the needed ray-sphere intersection is performed using the algorithm presented in section 4.3.1. Also, if the intersection test fails, no texture lookups and realtime computations have to be performed thus reducing the amount of computation needed. Applying this method then results in a realistic outlook of a planet from outer space (figure 4.9).

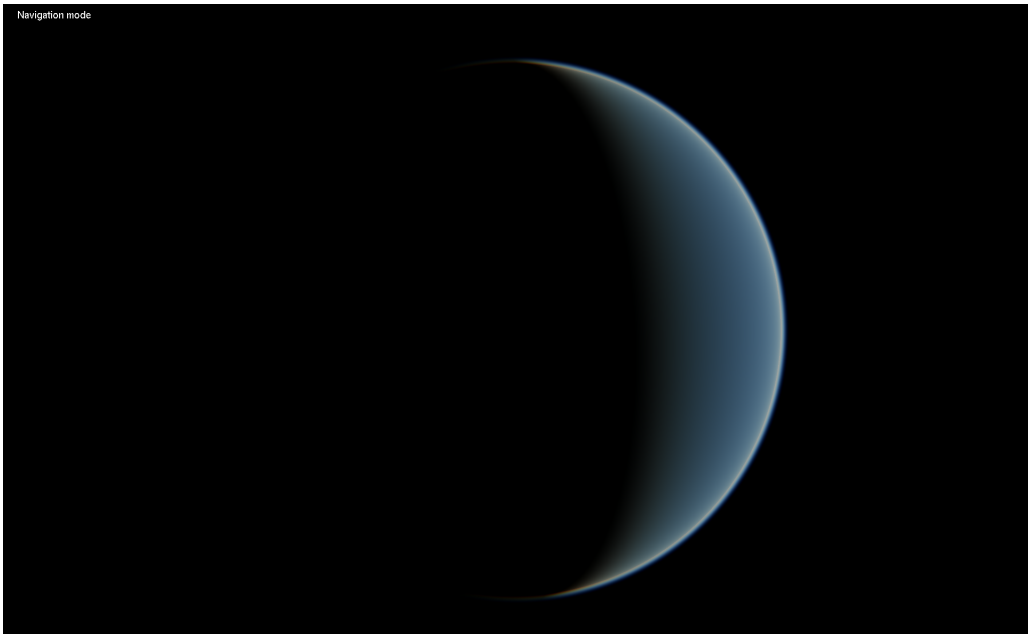

Figure 4.9: Halo of the planet as seen from space

4.5 Rendering the Planet

As the rendering is completely independent of any planet renderer, the planet just appears as a black sphere for now. Using what is already described in section 4.2, two additional textures are available for usage in the previously described fragment shader. Is the radius of the planet in the atmosphere calculation the same as in the planetary renderer, the image of the previously rendered scene can directly be used to color the planet. In this case, as the planet is already shaded and textured, the only missing effect is the attenuation from the surface of the planet to the viewer. This however introduces

another problem, because the distance from the viewer to the intersection point with the view-ray and the planet's surface is unknown, especially when assuming the planet is not a perfect sphere. Reconstructing the position of said pixel on the planet's surface in world coordinates is done in the next section.

4.5.1 Reconstructing the Planet's Surface

With the depth-buffer available, the geometry of the planet can be restored and then used for the atmosphere without influencing the planet rendering. The depth buffer contains the depth information from each fragment relative to the camera position in *normalized device coordinates* (NDC). This results in a depth component which is bounded by $0 < z < 1$, where $z = 0$ is true when the fragment is lying on the *near-plane* and thus $z = 1$ when the fragment is lying on the *far-plane*. To perform a perspective correct rasterization, the GPU computes $\frac{1}{z}$ at each vertex and then interpolates it along the triangles. So to increase performance, not z but $\frac{1}{z}$ is stored into the depth buffer. Unfortunately the reconstruction of the planet's surface requires the depth buffer to be linear, which makes a transformation into a linear z-value necessary.

$$z_{linear} = \frac{2n}{f + n - z(f - n)} \tag{4.24}$$

The linearization equation 4.24 takes the z-values of both near- (n) and far-plane (f) and linearizes the depth z [3].
After the depth is linearized, it can be used to reconstruct the geometry of the scene. Multiplying the not normalized vector from observer to the far-plane with the pixel's linearized depth component yields the vector from observer to the pixel in world space. Adding the position of the observer to that vector then results in the position of the pixel in world-space.
With this information, the transmittance from the pixel to the observer can be accessed by performing two texture lookups and dividing the results. This step is similar to the transmittance calculation done in precomputing the inscatter textures. Looking up the correct color of the planet's surface is done by a simple texture lookup in the color texture at the current pixel position. Attenuating the result with the transmittance factor then yields the correct color reaching the observer's eye.

4.5.2 Accounting for rough Terrain

Besides calculating the transmittance from the planet's surface to the observer's eye, the world-space position of the pixel can be used to account for terrain decals. Up until

[3] http://www.geeks3d.com/20091216/geexlab-how-to-visualize-the-depth-buffer-in-glsl/

now the planet was always assumed to be perfectly spherical, without accounting for mountains or valleys. Including these into the atmosphere calculation would disable the simplification that every point is parameterizable by just the height and view-direction. This would imply to either precompute the whole atmosphere or perform all calculations in realtime. Precomputing the whole atmosphere results in a greatly increased need of memory. Performing all computations in realtime has an unlike bigger influence on the framerate than the proposed method using lookup textures. But not accounting for rough terrain results in a transparency effect on the terrain, as shown in figure 4.10

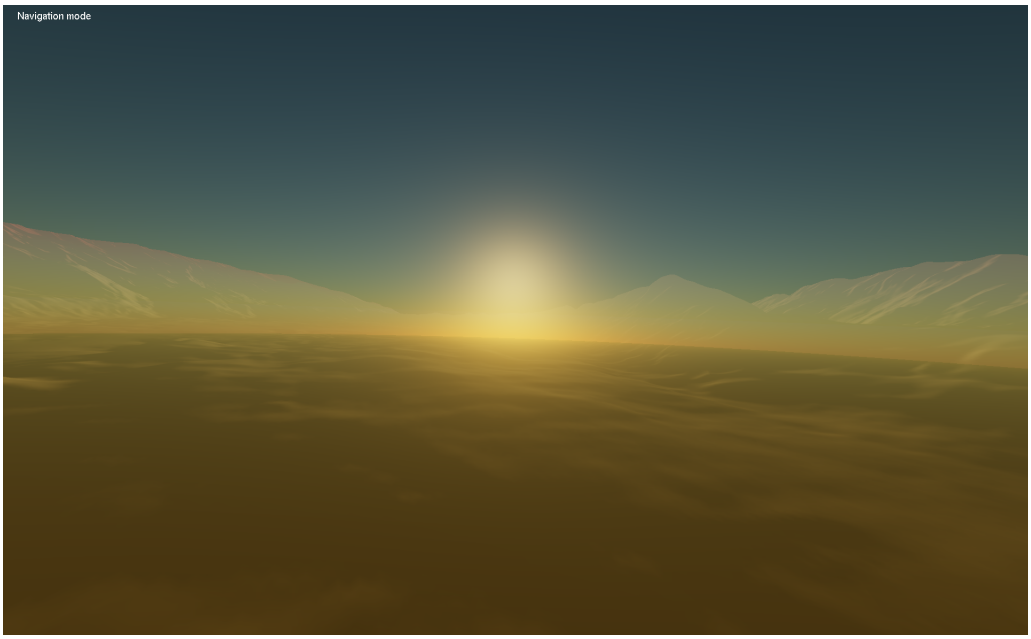

Figure 4.10: Transparency effect without accounting for rough terrain

Bruneton [BN08] presents an approach to include terrain features like hills and valleys into the atmosphere visualization, using just another texture lookup. Assuming a ray is cast from the observer through the atmosphere, hitting a mountain in the distance. In this current implementation, the mountain would be visible but shaded as if it wasn't there, because the inscatter lookup yields the color for the ray hitting the outer boundary instead of the mountain. But due to the nature of the atmosphere, the attenuation is additive. So when looking at the mountain, the atmosphere behind the mountain can be eliminated from the view by calculating the attenuation starting from the mountain reaching until the boundary of the atmosphere (see figure 4.11).

Because the world position of the pixels displaying the planet are already known, only one second lookup in the inscatter table is needed to adjust the atmosphere color. The four needed parameters for this lookup are the altitude of the pixel, the view-sun angle already calculated at the observer's position and new computed μ_{vz} and μ_{sz} using the pixels position.
After performing the second texture lookup, the resulting color has to be multiplied with

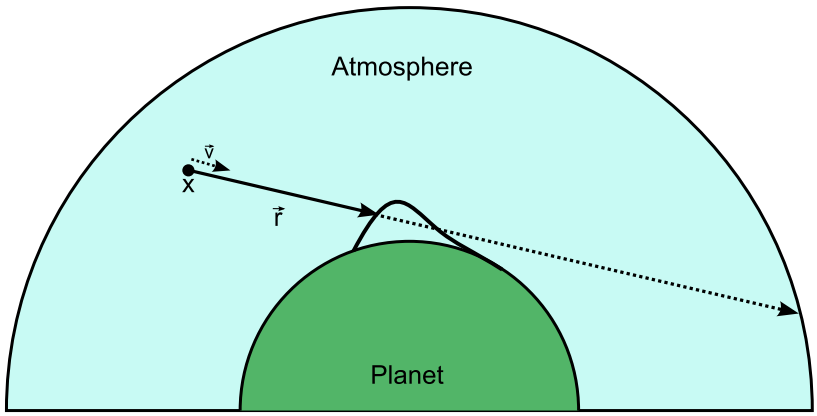

Figure 4.11: View-ray when colliding with rough terrain

the attenuation over distance to the observer. Subtracting the result from the original computed inscatter value then yields the correct inscatter values. With the correct inscatter color computed and the color from the planet generated and attenuated in the previous step, both can now be combined to the result. After adding the results of Rayleigh- and Mie-Scattering, the outcome is then multiplied by the sun-intensity. This value is arbitrary and has to be chosen in correspondence with the High-Dynamic-Range (HDR) rendering, applied after the whole image is generated. Adding this result to the planet's color then returns the correct color for the current pixel and thus produces the correct end result (see figure 4.12)

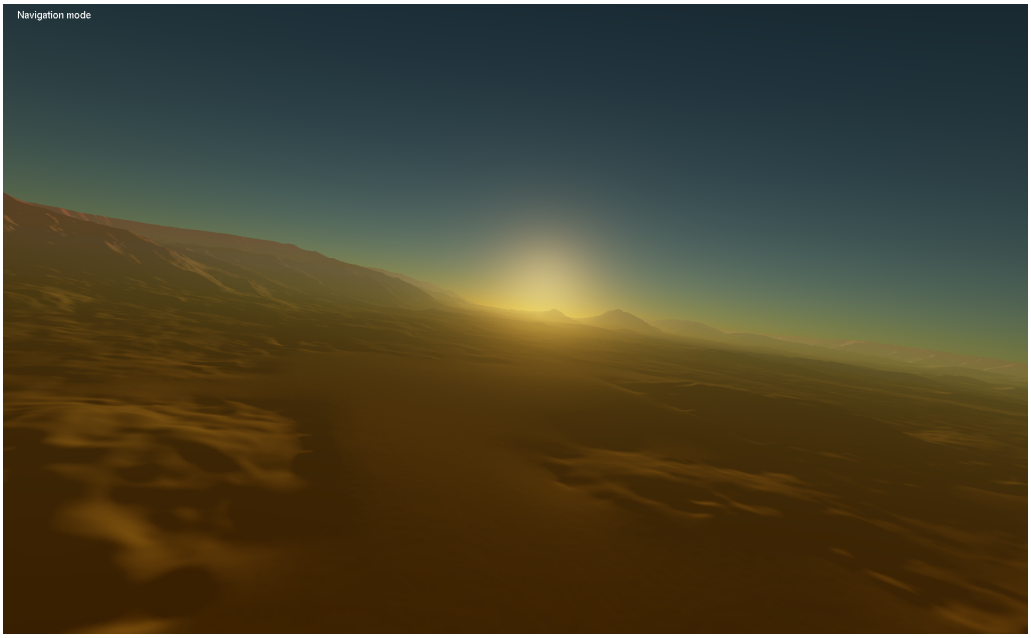

Figure 4.12: Corrected view of the terrain

4.6 Post-Processing

To improve the visual appearance of the atmosphere even further with simulating the view from the human eye or through a camera, two additional effects are applied after the scene itself is rendered. First a bloom operator is applied to simulate the perception of the human eye when looking at objects with high brightness, second the resulting image is modified using High-Dynamic-Range rendering and a fitting tonemapping operator.

4.6.1 Bloom

When taking a picture of a very bright object with a darker surrounding, the light from the bright object tends to bleed into the darker areas. This glow can be reproduced and emulated with taking a snapshot of the current scene and modifying its brightness. Because the effect is strongest when looking into bright light, the brightness of the snapshot is increased, e.g. with simply increasing the colors with the power of 4. To get the fuzzy and smooth light bleeding effect, the snapshot with the boosted brightness is then passed through two blur-shaders, one for horizontal and one for vertical blur.
Using a incremental Gaussian blur presented by Turkowski [TUR07] allows for efficient and flexible blurring of the rendered image. While blurring in general interpolates the color of one pixel by a weighted interpolation of the surrounding pixels, this process is very expensive due to the high number of texture lookups. When performed in one pass, the number of texture lookups in order to blur a point with all its n neighbours, the number of necessary lookups in the texture is n^2. Due to the nature of the Gauss-Kernel, its factors can be seperated into two passes, one in x- and one in y- direction. This reduces the computation power to $2n$.
With the incremental Gaussian blur, the kernel is computed for every lookup. When observing the quotient of successive computed values, it becomes apparent, that besides the first quotient being exponential, the following quotients are constant. This results in one exponential computation followed by constant caluclations for the remaining number of samples n.
The resulting image after increasing the brightness and applying both blur-passes applied is shown in figure 4.13

4.6.2 High-Dynamic-Range Rendering

A rendering of the atmosphere usually has a very broad range of brighness, due to the intensity of sunlight and shadow on the planet's surface. Because all rendering is done using a 16bit float texture, which provides 4bit float for each color channel, the texture is close to how the picture would look like in reality. But with the exception of few very

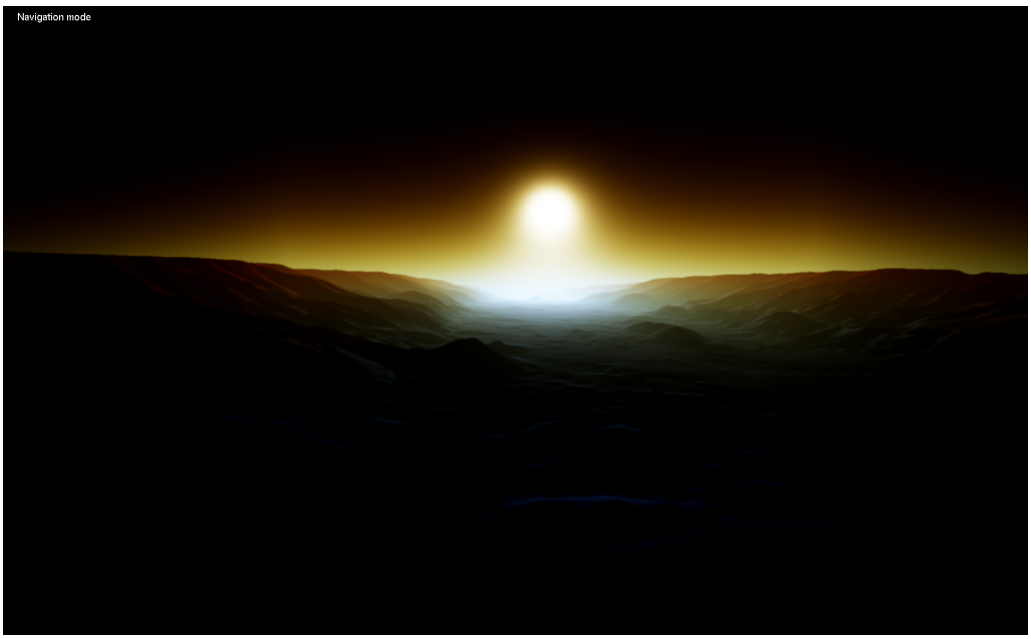

Figure 4.13: Blurred image with increased brightness

special monitors, modern hardware can't display such a wide range of brightness for each color. Most displays can just display up to 8bit per color channel, thus clamping the color values of the actual rendered texture. When a color channel exceeds the limit it is clamped and thus the two other color channels are too bright in comparison. This results in color-banding artefacts.
With the 16bit float texture available, the rendered image is considered to have a high dynamic range and while this is desired, it has to be modified to fit into the 8bit of most monitors or displays. Hence a tonemapping operator is applied on the image before rendering it onto the screen.

4.6.3 Tonemapping

Mapping the HDR-image into the used spectrum of 8bit per color can be done using a variety of operators. Again Bruneton [BN08] is using a modified version of a tonemapping operator by Reinhard [RSSF02], split into two equations.
The first one is applied for the intensity $I < 1.1413$:

$$I' = I \cdot \epsilon \cdot 0.38317^{\frac{1.0}{2.2}} \tag{4.25}$$

Otherwise the following operator is applied:

$$I = 1.0 - e^{(-I \cdot \epsilon)} \tag{4.26}$$

The exposure ϵ is a factor to reduce the overall brightness of the scene to match the image the human eye or a camera would perceive. Bruneton sets the exposure to a constant factor which already yield good results, although using a dynamic exposure adds additional realism to the rendering.
Dynamic exposure simulates the adjusting of the eye e.g. when switching from a dark to a bright environment. An example would be when an observer is situated in a dark room and then opens the door to the outside. It would take a moment to adjust his eyes to the new brightness until they can perceive additional detail despite the sudden increase of brightness. To simulate that behaviour, the overall brightness of the scene has to be known, also called the *luminance*. A quick way to get the overall brightness of the scene is exploiting the calculations of Mip-Maps. Due to the fact that mipmaps can be produced very quickly on the GPU, it is very efficient to generate the complete set of mipmaps of the screen texture. The lowest mipmap channel, consisting of one single pixel, then contains the average brightness of the whole scene for each color-channel. These values are then adjusted to fit the characteristics of the human eye, e.g. that green is perceived in much more detail than red and blue (equation 4.27).

$$l = 0.2126 \cdot R + 0.7152 \cdot G + 0.0722 \cdot B \tag{4.27}$$

Equation 4.28 then converts the luminance into the exposure. This is done by accounting for the gamma curve of typical displays, which transform the gamma using $x^{2.2}$. Already accounting for this effect then yields an exposure usable for the tonemapping correction.

$$\epsilon = \frac{1}{l} \cdot \frac{1}{(2^{2.2} - 1)} \tag{4.28}$$

This exposure is completely dynamic and changes with each frame, simulating adjustment of the human eye.
The $x^{2.2}$ factor of display gamma is also noticeable in the tonemapping by Reinhard. An alternate tonemapping operator proposed by Jim Hejl and Richard Burgess-Dawson [4]:

$$I_e = I \cdot \epsilon; \quad I = \frac{(I_e \cdot (6.2 \cdot I_e + 0.5)}{I_e \cdot (6.2 \cdot I_e + 1.7) + 0.06)} \tag{4.29}$$

The exposure and the tonemapping have a strong impact on the outcome of the atmosphere. Depending on the exposure, the result can look dull or overbright, depending

[4] http://filmicgames.com/archives/75

on what is actually visible on the screen. A comparison between the tonemapping used in this thesis and the tonemapping used by Bruneton is visible in figure 4.14

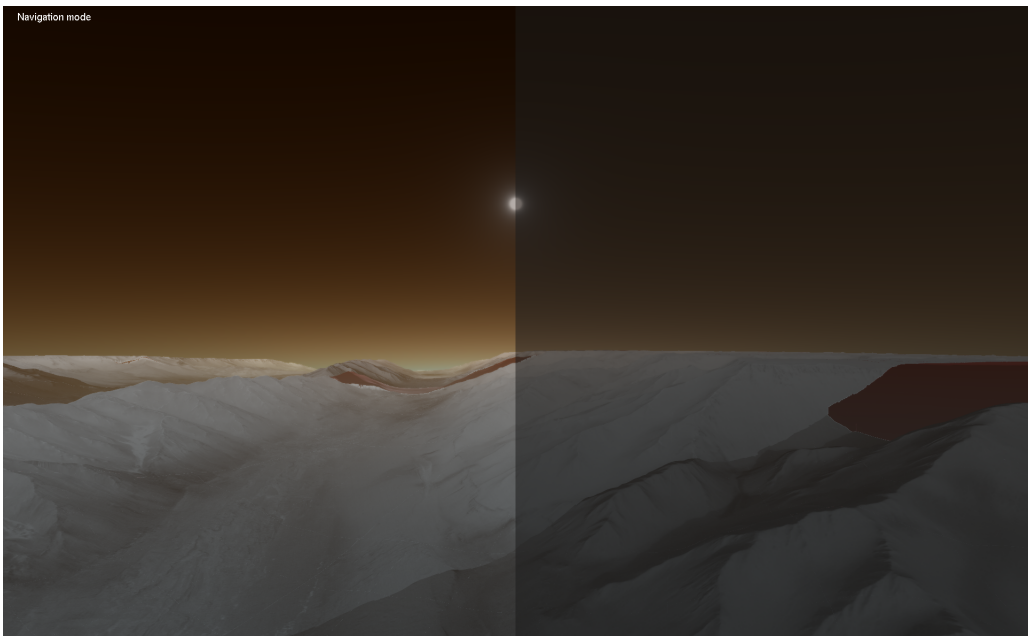

Figure 4.14: Tonemapping comparison; Left: Hejl Dawson, Right: Reinhard

Additionally a comparison between disabled HDR and enabled HDR, both with the previously introduced blooming and high sun intensity is visible in figure 4.15

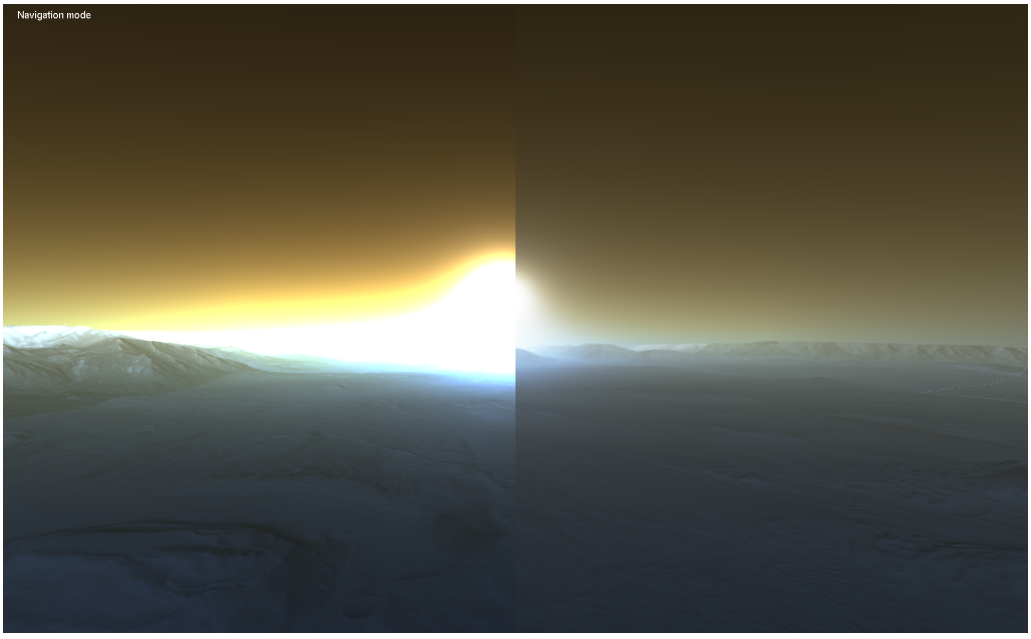

Figure 4.15: HDR comparison; Left: HDR disabled, Right: HDR enabled

4.7 Rendering the Sun

Even though the Mie-Scattering already simulates the sun's position due to an increased light-intensity (figure 4.16), the halo still lacks of intensity. Because the sun has a very

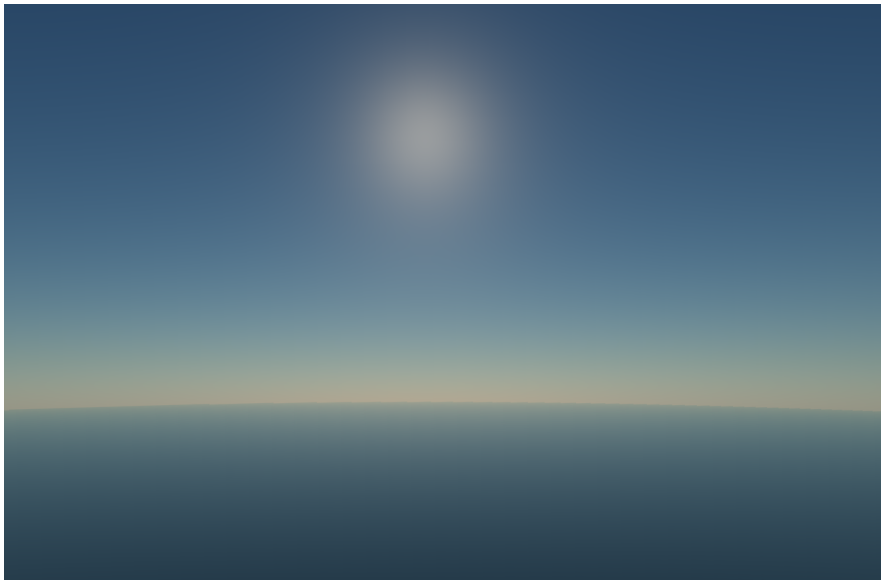

Figure 4.16: Halo of the sun due to Mie-Scattering

high light intensity, there normally is a circle visible when the sky is clear. Related implementations use a straight forward approach of rendering a simple point somewhere into the scene that resembles the sun [BN08].
Once again, because one of the goals is complete independence from the scene, rendering a sun into 3D-space is not an option. Also, because the far-plane of the scene should be as close as possible while still not cutting the planet, rendering an object very far away is not desirable. To circumvent this limitation, the sun's position can be analytically computed and then passed into the shader to draw a halo. First, the true position of the sun relative to the planet is computed by marching the distance from planet to sun along the current light-direction. Then, the position of the sun is transformed into screen-space by applying modelview- and projection matrices. Although the sun-direction is an approximation because it's assumed to be constant for every point in the atmosphere, this yields a fitting center position of the sun on screen. Based on the position on screen, we can display the sun in any arbitrary way. In this implementation, it is displayed using a fixed pixel distance to the center point and very high light intensity inside that radius. To generate a fading border of the sun, this light intensity is lowered by the power of two when the pixel's distance is greater than a fixed value. With this rather simple method, we already get a good result (figure 4.17) for the earth atmosphere, while the sun's size is set to resemble the view from Mars.
A similar result is observed when using the same formulation of the sun on the approximated Martian atmosphere (figure 4.18).

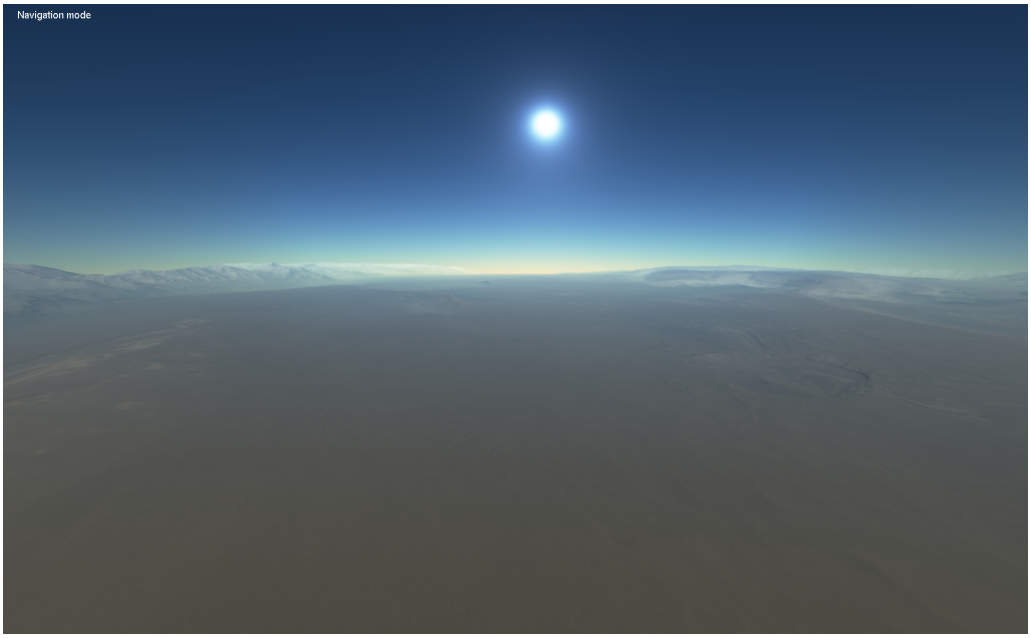

Figure 4.17: Earth's atmosphere with sun halo

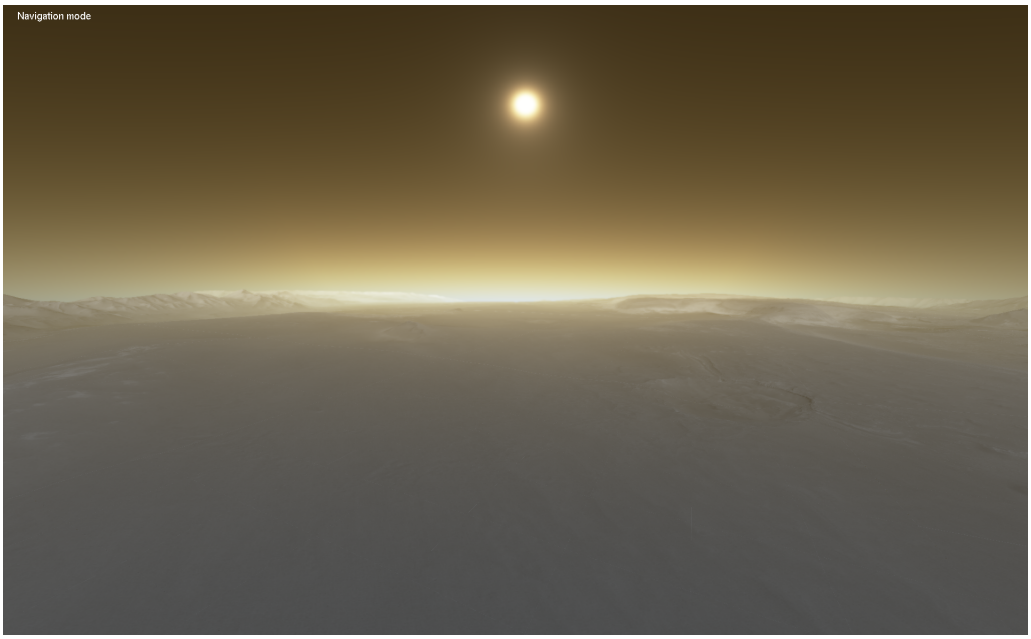

Figure 4.18: Mars atmosphere with sun halo

CHAPTER 5

RESULTS

To validate the results of the atmospheric scattering, the rendered images can be easily compared to real life photos or videos. When rendering the earth or a planet with a similar atmosphere, photos of the atmosphere with differing daytime can be easily researched. Also using the clear-sky model approach to visualize the atmosphere comes very close to the actual earth atmosphere, when no clouds are visible in the sky. In contrast to this, pictures from the Martian atmosphere are rare due to the low number of on-planet missions with high quality cameras. To recreate a realistic looking atmosphere, the photos from the Martian surface have to be color-corrected in order to yield the real atmosphere color. Also due to sandstorms and the dusty atmosphere, two pictures from the surface can look completely different although showing the same atmosphere. The dusty atmosphere is also a reason that parameters like the scale-height are often not applyable to the current state of the atmosphere.
Thus reliable information to base an atmosphere visualization on is limited. With the new Mars-Rover *Curiosity* which has high quality cameras to take pictures of the surroundings, these images are available as a reference for the Martian atmosphere [1]. Figure 5.1 shows a color-corrected photo of the *Mount-sharp*, situated in *Gale-Crater*. Fitting the parameters to resemble the atmosphere in that picture yields in the atmo-

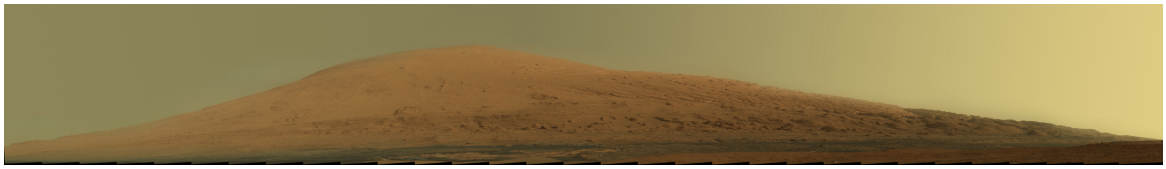

Figure 5.1: Photograph of mount sharp in gale crater

[1] http://www.nasa.gov/msl/

sphere on figure 5.2 This atmosphere is rendered and preprocessed using the parameters

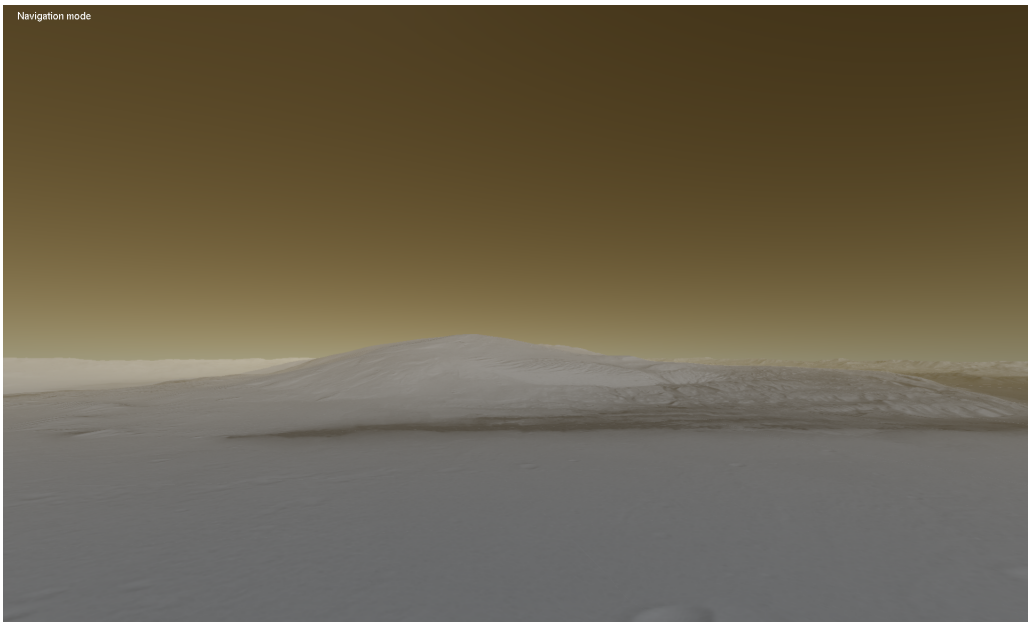

Figure 5.2: Rendered image of mount sharp with an approximated atmosphere

$\beta_R = (0.019918, 0.01357, 0.00575)$, $\beta_M = 0.003$, $H_R = 11.0$,$H_M = 3.2$ and a fitted radius to resemble Mars $R_{ground} = 3386km$ and $R_{atmosphere} = 3518km$.

5.1 Performance Evaluation

The performance of the implementation of the thesis is highly depending on the used hardware and the planetary renderer the implementation is based on. The implementation accompanying this thesis is build using the ViSTA framework [TvR00], which allows usage of VR-Technology. Also, the planetary renderer is expensive in rendering, thus the measured performance is always influenced by the surrounding software. All performance measurements are performed on a Workstation equipped with an *Intel Xeon Westmere X5670 (2,93 Ghz)*, a *NVIDIA Quadro FX5800* and 24GB DDR3 RAM.
With an disabled atmosphere, the planetary renderer is running with 39 frames-per-second (FPS) on a low level-of-detail. Adding the atmosphere reduces the framerate to 27 FPS. Measuring the runtime of the code on the CPU results in 0.03 milliseconds per rendered frame.
Using a high level-of-detail, the rendering performance drops to 13 FPS, whith the atmosphere disabled. Enabling the atmosphere results in a framerate loss to 7 FPS. Measuring the time it takes to complete one rendering call of the atmosphere resuls in 0.07 ms. Even though just the planetary rendering has been changed, the atmosphere takes longer to render. This is due to the GPU being already stressed with increased

detail on the planet and thus can't perform the atmosphere rendering as fast.
This shows the high dependency of the implementation on the rendering of other objects like the planet. Even in combination with the planet renderer, the atmosphere can still be rendered on a laptop or consumer computer. The available interactivity mostly depends on the level-of-detail setting used to render the planet and the desired resolution. Thus the goal of interactivity is met in general, even though the quality of the planetary rendering is highly depending on the hardware used for rendering.

CHAPTER 6

CONCLUSION

The work presented in this thesis allows for an interactive visualization of a wide varieties of atmospheres due to its parametrization. Even though simplifications are made to allow for interactive rendering, the results are satisfying for the usage in a Virtual-Reality environment and assist in the feeling of presence in VR. Because the implementation is only relying on an already existing render of any planet, it can be applied to a wide variety of planetary renderers and also can be configured to be a standalone software. In contrast to other methods on rendering the atmosphere, this work also allows rendering of the Martian atmosphere, which is also fit to available and color-corrected photographs by Mars missions. Additionally, a bloom method is integrated to mimic the human eye when looking into high brightness and a different tonemapping is applied to improve the atmosphere as seen from the surface of Mars. Also different from other implementations is the way the sun's position is passed onto the GPU and then rendered in a shader, producing a convincing looking sun.
While the parameterization allows for physical correctness, it is not intuitive and due to the lack of information on atmosphere data, the parameters can mostly be fit with comparing the atmosphere to existing photgraphs. The implementation itself is based on a physical background but is still using a simplified and approximated calculation to allow for interactive framerates. Still, the result is convincing for the eye and especially convincing when comparing to the earth's atmosphere.

6.1 Outlook

As with other implementations of an atmospheric scattering effect, a cloudy sky is a possible extension to increase the perceived realism. Also other techniques like the rendering of light shafts and potentially water would further improve the realism of the planet and atmosphere. Specifically to the atmosphere of Mars, storms of sand and dust also improve both the visual and physical aspect of the rendering, but are conflicting with the current simplifications made. Depending on the time of day and the density of the atmosphere, stars can be visible at night-time and thus should be rendered into the scene. Due to the way the rendering of the atmosphere is implemented in this thesis, this can be performed before the atmosphere itself is rendered and is then automatically attenuated correctly.

BIBLIOGRAPHY

[AMMH02] Tomas Akenine-Moller, Tomas Moller, and Eric Haines. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition, 2002.

[BN08] Eric Bruneton and Fabrice Neyret. Precomputed atmospheric scattering. In *Computer Graphics Forum*, volume 27, pages 1079–1086. Wiley Online Library, 2008.

[CS92] William M. Cornette and Joseph G. Shanks. Physically reasonable analytic expression for the single-scattering phase function. *Appl. Opt.*, 31(16):3152–3160, Jun 1992.

[DR06] Wolfgang Dahmen and Arnold Reusken. *Numerik fuer Ingenieure und Naturwissenschaftler*, volume 2. Springer, 2006.

[GHB+05] Krzysztof M Gorski, Eric Hivon, AJ Banday, Benjamin D Wandelt, Frode K Hansen, Mstvos Reinecke, and Matthia Bartelmann. Healpix: a framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759, 2005.

[Kel77] DH Kelly. Visual contrast sensitivity. *Journal of modern optics*, 24(2):107–129, 1977.

[NSTN93] Tomoyuki Nishita, Takao Sirai, Katsumi Tadamura, and Eihachiro Nakamae. Display of the earth taking into account atmospheric scattering. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 175–182. ACM, 1993.

[ON05] Sean ONeil. Accurate atmospheric scattering. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005.

[Ray99] Lord Rayleigh. Xxxiv. on the transmission of light through an atmosphere containing small particles in suspension, and on the origin of the blue of the sky. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 47(287):375–384, 1899.

[RSSF02] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. Graph.*, 21(3):267–276, July 2002.

[SFE07] Tobias Schafhitzel, Martin Falk, and Thomas Ertl. Real-time rendering of planets with atmospheres. 2007.

[Spe11] Stefan Sperlhofer. Deferred rendering of planetary terrains with accurate atmospheres, msc thesis, 2011.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. *SIGGRAPH Comput. Graph.*, 24(4):197–206, September 1990.

[TUR07] K TURKOWSKI. Incremental computation of the gaussian. In *GPU Gems 3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2007.

[TvR00] Andreas Gerndt Joerg Henrichs Christian Bischof Thomas van Reimersdahl, Torsten Kuhlen. Vista: A multimodal, platform-independent vr-toolkit based on wtk, vtk, and mpi. *Fourth International Immersive Projection Technology Workshop (IPT 2000)*, 2000.

[WGH+11] Rolf Westerteiger, Andreas Gerndt, Bernd Hamann, Christoph Garth, Ariane Middel, and Hans Hagen. Spherical terrain rendering using the hierarchical healpix grid. In *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering-Proceedings of IRTG 1131 Workshop 2011*, volume 27, pages 13–23. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.