FH Technikum Wien

MASTER'S THESIS

Thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Engineering

at the University of Applied Sciences Technikum Wien
Game Engineering and Simulation

Deferred Rendering of Planetary Terrains with Accurate Atmospheres

by
Stefan Sperlhofer, BSc
A-1200, Vienna, Höchstädtplatz 5

Supervisor 1: Dipl. –Ing. Stefan Reinalter
Supervisor 2: Dipl. –Ing. Dr. Gerd Hesina
Vienna, 31.08.2011

FACHHOCHSCHULE TECHNIKUM WIEN

Declaration

„I confirm that this thesis is entirely my own work. All sources and quotations have been fully acknowledged in the appropriate places with adequate footnotes and citations. Quotations have been properly acknowledged and marked with appropriate punctuation. The works consulted are listed in the bibliography. This paper has not been submitted to another examination panel in the same or a similar form, and has not been published. "

Place, Date	Signature

Kurzfassung

Für die realistische Darstellung der Erde unter verschiedensten Größenordnungen, ist eine korrekte Visualisierung der atmosphärischen Streuung von äußerster Wichtigkeit. Aufgrund der hohen Komplexität der entsprechenden physikalischen Gleichungen ist jedoch eine Berechnung in Echtzeit nicht möglich. Aufgrund dessen werden üblicherweise vereinfachte Modelle für die Berechnung herangezogen, welche üblicherweise geringere Genauigkeit oder eingeschränkte Flexibilität bieten.

In dieser Arbeit wird gezeigt wie Planeten mit physikalisch korrekten Atmosphären mittels Deferred Rendering visualisiert werden können. Dabei wird auf dem Modell von Bruneton und Neyret aufgebaut, welches die entsprechenden Gleichungen in einem separaten Vorberechnungsschritt löst. Diese vorberechneten Daten werden anschließend benutzt um die Atmosphärische Streuung mittels eines Post-Effekts darzustellen. Die daraus resultierenden Vorteile wären unter anderem eine vereinfachte Integration in bestehende Systeme, geringere Komplexität, vorhersehbare Berechnungskosten sowie die Vermeidung von Shader Permutationen. Die Darstellung des Planeten erfolgt auf Basis eines Würfels, welcher in einzelne Blöcke eingeteilt wird. Um die Darstellung des Planeten in unterschiedlichen Größenordnungen zu ermöglichen, wird ein Sichtbarkeits- sowie Detailierungsalgorithmus angewandt. Zudem wird Normal Mapping, Multi-Texturing sowie eine prozedurale Generierung der Oberfläche unterstützt.

Der erste Teil der Arbeit stellt die entsprechenden physikalischen Modelle vor und erklärt verschiedenste Phänomene der Atmosphäre, wie beispielsweise die Änderung der wahrgenommenen Farbe mit zunehmender Distanz, die Farbe des Himmels und den Lichtring der Sonne. Im zweiten Teil wird die Darstellung des Planeten, der Sonne und der Atmosphäre im Detail erklärt. Des Weiteren werden die Resultate des vorgestellten Models unter verschiedensten Bedingungen demonstriert. Der letzte Teil behandelt schließlich mögliche Einschränkungen sowie Verbesserungsvorschläge des präsentierten Verfahrens.

Schlagwörter: Atmosphärische Streuung, Planet, Deferred Rendering, 3D Computergrafik

Abstract

Correct atmospheric scattering effects are crucial when visualizing the earth on varying scales or time of the day. Due to the complexity of the corresponding light transfer equations, current hardware is not able to compute these effects in real-time. Hence, interactive frame rates are usually achieved by various simplifications over the physical model, which usually results in less accuracy or flexibility.

This thesis presents a deferred approach to rendering physical correct atmospheres and planetary terrains in real-time. The atmospheric model is based on the work of Bruneton and Neyret and pre-computes the scattering equations in a separate offline pass. This pre-computed data is then utilized, to apply atmospheric scattering as a single post-processing effect. Using a post-process technique has several advantages over a traditional approach. These are: simplified integration, reduced complexity, predictable rendering costs and prevention of shader permutations. The planetary terrain is based on a tiled block algorithm which utilizes a cube as its base geometry. To allow visualization on different scales, the proposed model offers level of detail and frustum culling capabilities. In addition, the planets are generated procedurally using noise functions and allow for multi texturing and normal mapping.

The first part of this thesis introduces the reader to the physical model of atmospheric scattering and explains various resulting phenomena such as the shifted colors of distant objects, the color of the sky and the visible halo surrounding the sun. The second part examines the proposed approach and provides detailed explanations on rendering the terrain, the sun and the atmosphere. Furthermore, the results of this model are demonstrated under various conditions. The last part of this thesis reveals the limitations of the presented approach and proposes various improvements for future work.

Keywords: Atmospheric scattering, planet, deferred rendering, 3d computer graphics

Acknowledgements

I would like to show special gratitude to my supervisor, Stefan Reinalter, for sharing his knowledge and experience. His effort and support throughout my thesis encouraged me to improve this work continuously.

Thanks to my dear brother, Johannes, for proofreading my thesis.

It is also a pleasure to thank Eric Bruneton for clarifying certain parts of his work.

I would also like to thank John McLaughlin and the following people from GameDev.net for their precious input on various topics: *alvaro*, *jyk*, *MJP*, *SiCrane* and *__sprite.*
You are awesome! Hopefully I can give something back to the community, by sharing this thesis.

Special thanks also belong to my lovely girlfriend, who always believed in me. Without your support over the past years, I would not stand where I am now.

Last but not least:
Thank you mum for not forcing me doing exhausting outdoor activities, when I was a kid. Playing video games finally paid off.

Johannes, thank you for suggesting spending all of my pocket money on the Nintendo Entertainment System, back then. Apparently, it was a safe investment!

Table of Contents

1 Introduction

Due to natural perception and our exposure to the physical world, we have a quite accurate conception of how things behave in our everyday life. We are therefore very sensitive to artificial approximations of the real world, even if the exact physical models are unfamiliar to a large extend. Thus reproducing physical phenomena is a vastly researched topic in computer graphics.

One of the main topics of physical approximation in computer graphics is the scattering of light in participating media*. Atmospheric scattering* describes the scattering of light due to the ingredients of the earths atmosphere (gases, water vapor, dust particles etc.).

The atmosphere of the earth is a layer of gases surrounding the earth. These gases are retained due to the earth's gravity. When light passes through this atmosphere, air molecules and so called aerosols (particles like dust or pollution) interact with it and scatter the light in different directions. This scattering is called atmospheric scattering. Simulation of atmospheric scattering is essential for reproducing realistic outdoor scenery or the earth viewed from space. One of the most obvious effects of atmospheric scattering is the blue color of the sky and the red and yellow colored sun during sunrise and sunset. A more subconscious effect is the blue tint of distant objects. This is the reason why for instance mountains are perceived with slightly washed out colors. These effects shift with changing composition within the atmosphere. So, for example, an increase of water vapor on a rainy day has the effect that everything looks a little bit grayish and even more washed out.

This thesis is built on previous research in the field of atmospheric scattering and extends these in various aspects. It makes use of pre-computed tables and therefore solves the complex scattering equations in a separate offline pass to preserve interactive frame rates during rendering. These pre-computed tables are then used to apply atmospheric scattering to an arbitrary scene in a single post-processing effect by using deferred rendering. Although the focus of this work lies upon atmospheric scattering, it is also shown how spherical terrains are generated by using a tiled-block approach.

In the following chapter the physical models of atmospheric scattering are discussed in further detail. Related work in the field of atmospheric scattering, terrain- and deferred rendering is presented in chapter 3. Chapter 4 introduces the reader to the proposed model and shows how atmospheric scattering can be applied to a spherical terrain as a post-processing effect. Limitations and future work are then discussed in chapter 5.

2 Principles of Atmospheric Scattering

This chapter serves as an introduction to the physical models of atmospheric scattering and introduces the reader to the most important light transfer equations.

2.1 Introduction to the Physical Models

The sun radiates light of all wavelengths in nearly equal intensities. When the sunlight penetrates the atmosphere it gets attenuated. This happens due to the various ingredients of the atmosphere which scatter and absorb the sunlight. Scattering of light differs with particle size and varies with wavelength.

Smaller air molecules scatter shorter wavelength light considerably stronger. Blue light has the shortest wavelength, so it is scattered much stronger by these than longer wavelength light. As blue light gets scattered and reflected all over the place, it reaches our eyes from every direction. This is the reason why the sky is blue on a clear day. When the sun is near the horizon at sunset or sunrise, the light travels a long distance through the atmosphere and therefore most of the short wavelength light, like blue and green light, gets scattered away, so it is perceived primarily with red colors.

Larger particles such as dust and pollution are called aerosols and basically scatter light of all wavelengths equally. In addition, aerosols also absorb parts of the light. Aerosols are the reason why the sky looks gray and washed out on a hazy day.

The proportion of light that is scattered away from its incident direction is a product of a scattering coefficient β^s and a phase function Ph. The angle θ describes the angle between the incoming light ray and the scattering direction. The phase function then returns the amount of light that is scattered under the given angle θ. Unlike air molecules which basically scatter light in every direction equally, aerosols scatter light primarily in the forward direction, which means they are scattered roughly in the same direction in which they originally started. The phase function describes this angular dependency and therefore differs for air molecules and aerosols.

Atmospheric scattering commonly used in computer graphics considers a clear sky model, which is only based on two constituents, air molecules and aerosols, in a thin spherical layer of decreasing density between the bottom R_g and the top R_t of the atmosphere [1].

2.2 Rayleigh Scattering

Scattering of air molecules and particles, which are smaller than 10% of the light's wavelength, is given by the Rayleigh theory [2], discovered and named after the Nobel prize winner Lord Rayleigh. The scattering coefficient for *Rayleigh scattering* β_R^s can be obtained as shown in Equation 2-1.

$$\beta_R^s(\lambda) = \frac{8\pi^3(n^2 - 1)^2}{3N\lambda^4} \qquad \text{Equation 2-1}$$

Constants of this equation are n which describes the refractive index of air and N which stands for the molecular density at the bottom of the atmosphere R_g. Rayleigh scattering is inverse proportional to the 4th power of the wavelength λ, which explains the strong attenuation of short wavelength light.

The extinction coefficient β^e determines how much light is scattered or absorbed. Air molecules only reflect light and do not absorb it. Therefore the corresponding extinction coefficient of air molecules equals the scattering coefficient: $\beta_R^e = \beta_R^s$.

As it was briefly stated in chapter 2.1, air molecules scatter light in every direction in nearly equal manner. Figure 2-1 shows the relative intensity of scattered light for the angles $[0, \pi]$ due to Rayleigh scattering.

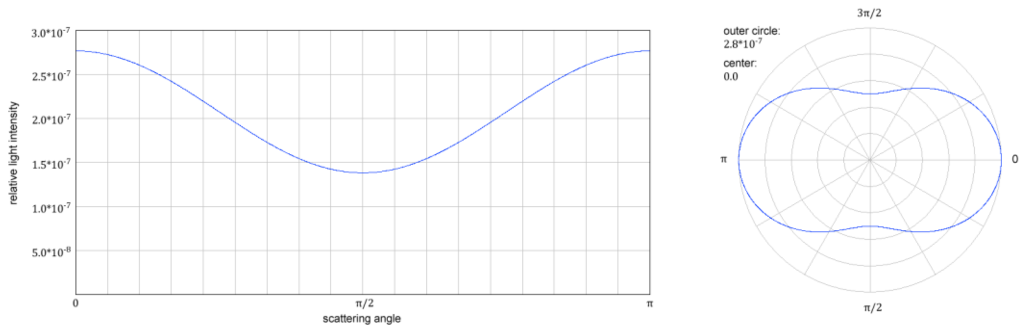

Figure 2-1: Rayleigh scattering ($\lambda = 0.45$μm)
(left) plot for range $[0, \pi]$ (right) polar plot for range $[0, 2\pi]$

As shown in Figure 2-1 the relative intensity of the scattered light falls off slightly at angles near $\frac{\pi}{2}$.

An approximation of the corresponding phase function (described in chapter 2.1) for Rayleigh scattering Ph_R is given by Equation 2-2.

$$Ph_R(\mu) = \frac{3}{16\pi}(1 + \mu^2) \qquad \text{where } \mu = \cos\theta \qquad \text{Equation 2-2}$$

2.3 Mie Scattering

Mie scattering describes the scattering of aerosols, which are particles larger or equal to 10% of the light's wavelength, and is named after Gustav Mie [3]. According to Nishita et al. [4] the scattering coefficient for Mie scattering β_M^s equals the scattering coefficient for Rayleigh scattering except the $\frac{1}{\lambda^4}$ dependence and is calculated as shown in Equation 2-3.

$$\beta_M^s = \frac{8\pi^3(n^2 - 1)^2}{3N} \qquad \text{Equation 2-3}$$

Aerosols also absorb parts of the incident light. The corresponding extinction coefficient of aerosols is the sum of an absorption coefficient β_M^a and the scattering coefficient: $\beta_M^e = \beta_M^a + \beta_M^s$.

The strong forward scattering of aerosols is shown in Figure 2-2. As can be seen most of the light is scattered in its original direction (scattering angles close to 0). Thus, the relative intensity scattered will fall off drastically if the scattering angle differs slightly from the incident direction.

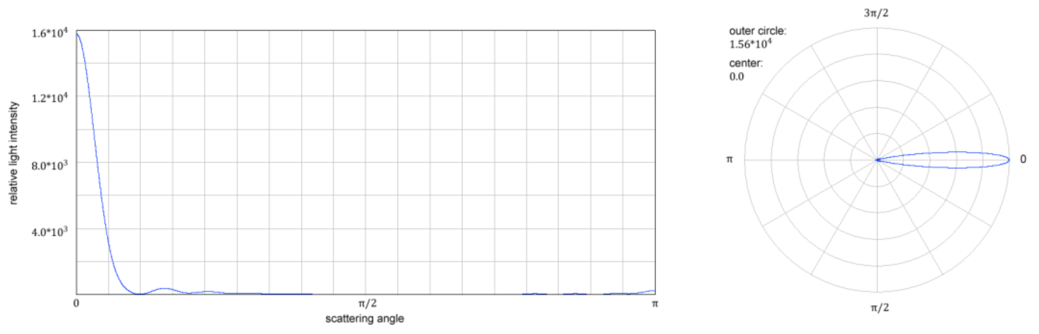

Figure 2-2: Mie scattering
(left) plot for range $[0, \pi]$ (right) polar plot for range $[0, 2\pi]$

This angular dependency of Mie scattering can be approximated by the improved Henyey-Greenstein phase function of Cornette and Shanks [5], which is given by Equation 2-4.

$$Ph_M(\mu) = \frac{3}{8\pi} \frac{(1 - g^2)(1 + \mu^2)}{(2 + g^2)(1 + g^2 - 2g\mu)^{3/2}} \qquad \text{Equation 2-4}$$

where g affects the symmetry of scattering. Setting g to 0 basically approximates Rayleigh scattering [6].

2.4 Optical Depth

The Optical Depth describes the optical thickness of a medium and is a measure of light transparency over a given path. It is dependent on the atmospheric density, which decreases toward the top boundary exponentially. The density ratio ρ at position x is given by Equation 2-5.

$$\rho_R(x) = e^{-\frac{h(x)}{H_R}} \qquad \rho_M(x) = e^{-\frac{h(x)}{H_M}}$$ Equation 2-5

where h describes the distance of x to R_g and H denotes the scale height. The scale height is used to vary the density ratio between R_g and R_t and is different for air molecules and aerosols.

The optical depth of a path S can be calculated by integrating the extinction coefficients and the density ratio over this particular path as shown in Equation 2-6.

$$t(\lambda, S) = \int^{S} \sum_{i \,\epsilon\, \{R,M\}} \beta_i^e(\lambda)\rho_i(s)ds$$ Equation 2-6

The optical depth can be used to obtain the transmittance of the medium and so the extinction factor along a path as described in Equation 2-7.

$$F_{ex}(\lambda, S) = e^{-t(\lambda,S)}$$ Equation 2-7

In this context, the extinction factor can be understood as the fraction or percentage of an incident light that remains after traversing the atmospheric medium over a given path.

Most of the following functions depend on the wavelength λ. To enhance readability denoting this dependency is omitted from now on.

2.5 The Light Scattering Equations

The light scattering equations describe how much light arrives at a position due to scattering within the atmosphere. This light can be expressed as a series of linear operations. Therefore, the resulting light intensity arriving at position x over the path $x \rightarrow x_0$ is basically a sum of three components as shown in Equation 2-8: direct sunlight L_0, in-scattered light L_{in}[L] and reflected light $L_{ref}\,[L]$ [1].

$$L(x, x_0, x_s) = L_0(x, x_s) + L_{in}[L](x, x_0, x_s) + L_{ref}[L](x, x_0, x_s)$$ Equation 2-8

Where x_s describes the position where the sunlight enters the atmosphere. Each of these three components is described in further detail in the following chapters.

Note that in the following chapters light is also expressed using a more general term $L_*(x, v, s)$, which describes light that reaches a position x from direction v when the sun is in direction s [1] (L_* can be either L, L_0, L_{in} or L_{ref}). The direction vectors v and s can be described by two positions (x_{start}, x_{end}). In this case the vector describes the normalized vector resulting from $x_{end} - x_{start}$.

2.5.1 Direct Sunlight

Sunlight incident to the outer boundary of the atmosphere at point x_s is attenuated while traversing the atmospheric medium as shown in Figure 2-3. The remaining light reaching point x due to direct sunlight is obtained by attenuating the incident light intensity at x_s over the path $x \rightarrow x_s$ as described by Equation 2-9.

$$L_0(x, x_s) = F_{ex}(x \rightarrow x_s) L_{incident}$$ Equation 2-9

where $L_{incident}$ describes the incident sunlight at x_s. Note that L_0 is 0 when the direction (x, x_s) does not equal the direction to the sun or when the sun is occluded [1] (eg. by a mountain).

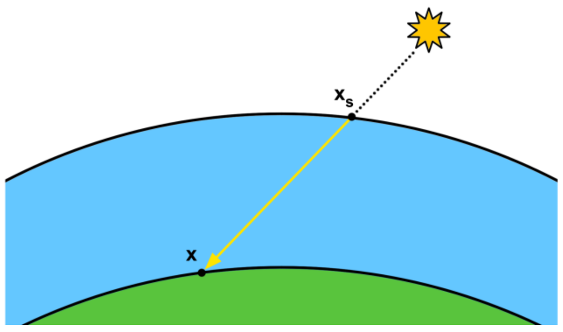

Figure 2-3: Sunlight traverses the atmosphere between x_s and x

2.5.2 In-scattered Light

When light is scattered it is removed from the original ray, but as long as it isn't absorbed, it will get reflected and *in-scattered* into the path of a ray headed in a different direction [7].

This results in a so called *self illumination* of the participating medium [8]. The in-scattered light is a result of the phase function Ph_R and Ph_M for air molecules and aerosols respectively. Figure 2-4 shows the scattering at y towards x.

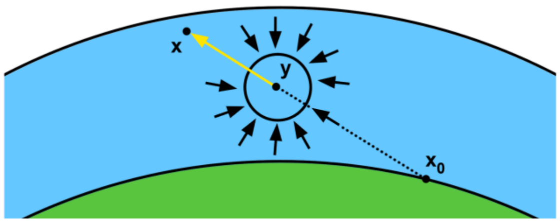

Figure 2-4: Scattering of light at y towards x is calculated by integrating over a sphere

The light scattered at a point y into direction v is given by Equation 2-10 [1].

$$J[L](y,v,s) = \int_0^{4\pi} \sum_{i \in \{R,M\}} \beta_i^s \rho(y) Ph_i(v.w) L(y,w,s) dw \qquad \text{Equation 2-10}$$

The total intensity of in-scattered light along a path ranging from $x \to x_0$ is obtained by integrating over all scattering points along this particular path as shown in Equation 2-11 [1].

$$L_{in}[L](x,x_0,x_s) = \int_x^{x_0} F_{ex}(x \to y)\, J[L](y, v(x,x_0), s(y,x_s)) dy \qquad \text{Equation 2-11}$$

Note that the light in-scattered at each point is attenuated before reaching position x.

2.5.3 Reflected Light

Usually, light incident to a surface is not absorbed. Instead it is reflected to a different direction. The light incident to a certain point on a surface is commonly referred to as the *irradiance.* The irradiance can be calculated by integrating over the hemisphere of surface point x_0 as described by Equation 2-12 [1].

$$I[L](x_0,s) = \int_0^{2\pi} L(x_0,w,s) w.n(x_0) dw \qquad \text{Equation 2-12}$$

where n describes the surface normal at point x_0.

The remaining intensity of light reflected at x_0 and arriving at x is obtained by attenuating the reflected light along the path $x \to x_0$ as described by Equation 2-13 [1].

$$L_{ref}[L](x, x_0, x_s) = F_{ex}(x \rightarrow x_0)\,\frac{\alpha(x_0)}{\pi} I[L](x_0, s(x_0, x_s))$$ Equation 2-13

where α describes a reflectance factor at x_0 (basically a value between 0 and 1). The term $\frac{1}{\pi}$ is a normalization factor, as the integration over the hemisphere yields π.

Figure 2-5 depicts the calculation of reflected light.

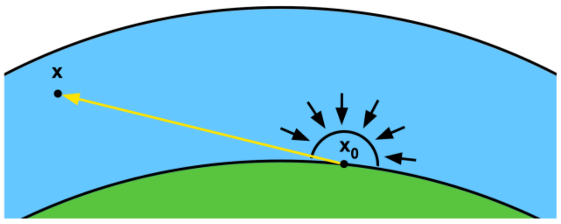

Figure 2-5: Irradiance at surface point x_0 is calculated by integrating over the hemisphere. Parts of this light are reflected towards x

2.6 Aerial Perspective

Depending on the time of the day and the composition of the atmosphere, objects far away are perceived with slightly shifted colors. This is what is generally referred to as the *aerial perspective*. According to Goldstein this effect is a fundamental requirement for humans to estimate distances, especially for unfamiliar objects [9].

Although aerial perspective is not a special case of atmospheric scattering, it is mentioned here for completeness. Special handling of aerial perspective in computer graphics papers is usually the result of various simplifications made to the scattering equations and therefore it is very common to devote an own chapter to it. Aerial perspective is just the result of an extinction and an addition part. For its calculation, the in-scattered and reflected light needs to be taken into account as described by Equation 2-14.

$$L_{aerial\ perspective}(x, v, s) = \left(L_{in}[L] + L_{ref}[L]\right)(x, v, s)$$ Equation 2-14

The reflected light on a distant object is scattered on its way to the observer. On a clear day the blue light is most affected. Therefore a large part of the blue color gets scattered away before reaching the observer. Without in-scattering this would just remove light, giving distant objects a strong shift towards yellow and brown tones as green and especially red colors are hardly affected by the scattering of air molecules. But scattering also adds colors. As mentioned before, blue light is scattered stronger and therefore has a higher probability to get scattered into an arbitrary viewing ray. This is what gives distant objects usually a blue hue. This shifting gets stronger with increased distances as more light is in-scattered. This ultimately leads to a whitening of objects very far away. The effect of aerial perspective is best seen on distant dark or shadowed objects. These objects are perceived with a strong hue towards blue, as little light is reflected and the impact of in-scattering is seen more clearly. In contrast to this, the effect of light scattering is less apparent on white objects, as the addition and extinction of blue light counter each other for the most part [10].

As stated above, this effect varies with composition of the atmosphere. At an atmospheric condition with increased aerosols, scattering is less dependent on wavelength and therefore green and red light gets scattered stronger. The result is that distant objects are perceived with a general loss of contrast and the colors are basically shifted towards gray [10].

2.7 Sunlight and the Color of the Sky

The colors of the sun and the sky are also the result of the scattering equations described in chapter 2.5.

$$L_{sun}(x, v, s) = (L_0 + L_{in}[L])(x, v, s)$$

Equation 2-15

The color of the sun perceived at position x is described by Equation 2-15. It is a combination of direct sunlight and in-scattered light.

As the sun radiates light of all wavelengths in nearly equal intensities, it is perceived as almost pure white. However, when the sunlight penetrates the atmosphere it gets attenuated due to scattering. Depending on the distance the light rays traverse within the atmosphere, the perceived color of the sun changes. The visible halo around the sun is the result of Mie-scattering, which scatters light more likely in its original direction and is therefore more obvious towards the direction of the sun. This effect is stronger in atmospheric conditions with increased aerosols (eg. hazy or rainy days).

The color of the sky is the result of in-scattered light as shown by Equation 2-16.

$$L_{sky}(x, v, s) = L_{in}[L](x, v, s)$$

Equation 2-16

Towards the horizon the sky is getting whiter (for the same reason why objects are getting whiter with distance – as described in the previous chapter). An increase of Aerosols shifts the color of the sky towards gray.

When the sun approaches the horizon at sunset or sunrise, the sunlight traverses a long path within the atmosphere and thus most of the blue light and some parts of the green light are scattered away before reaching an observer. Therefore the colors of the sky and the sun itself changes to yellow and red tones. In addition the slight fall off near $\frac{\pi}{2}$ of the Rayleigh phase function (as mentioned in chapter 2.2) becomes more apparent, as the darkest part of the sky can be found near the zenith, while the region near and opposing the sun are the brightest.

2.8 Multiple Scattering

Considering just a single scattering event per light ray is generally referred to as *single-scattering*. In reality, light rays are scattered multiple times and thus can change their direction more than once. This effect is usually called *multiple scattering*.

Multiple scattering refers to a model, where multiple scattering events per light ray are taken into account. Figure 2-6 shows three different light rays. Ray a) is only scattered once and then stays on its direction towards x. In a single-scattering model ray b) and c) would not be considered, as they are scattered multiple times. Ray b) depicts multiple in-scattered light and ray c) multiple reflected light.

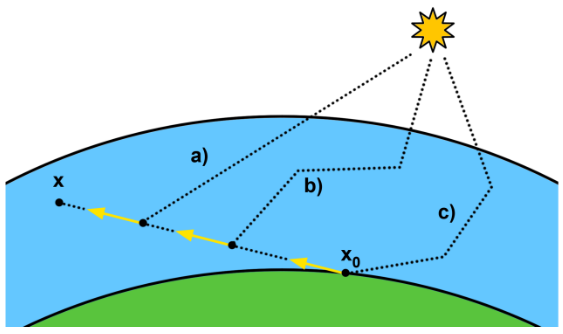

Figure 2-6: a) single in-scattered light b) multiple in-scattered light c) multiple reflected light

Recall that the total light reaching a point within the atmosphere can be expressed as a series of linear operations (as described in chapter 2.5). This means when taking multiple scattering into account, the total light reaching a point x is calculated as shown in Equation 2-17 [1].

$$\begin{aligned} L(x,v,s) &= L_0 + \left(L_{in} + L_{ref}\right)[L_0] + \left(L_{in} + L_{ref}\right)\left[\left(L_{in} + L_{ref}\right)[L_0]\right] + \cdots \\ &= L_0 + L_1 + L_2 + \ \ldots = L_0 + L_i \end{aligned}$$ Equation 2-17

L_i describes the light that is scattered or reflected exactly i times [1].

In simpler atmospheric models multiple scattering is completely ignored, as leaving it out is less noticeable in midday scenarios, where the light beams are traversing short distances through the atmosphere. Yet, the effects of multiple scattering become crucial when the sun is near the horizon or in hazy conditions, as the light rays are much more affected by the atmospheric media.

3 Previous Work

In this chapter relevant work devoted to atmospheric scattering as well as terrain and deferred rendering is reviewed.

3.1 Atmospheric Scattering

Over the past few years, there has been a considerably large amount of work devoted to reproducing atmospheric scattering.

Hoffman and Preetham propose an atmospheric scattering model which is capable of producing real time frame rates without any pre-computation [7]. This is possible due to a simplification of the optical depth, as the atmospheric density is assumed to be constant. Basically, an extinction coefficient and the in-scattered light are calculated for every vertex in the scene. These two values are then combined in the pixel shader. In this model the observer is assumed to stay near the ground. Due to the assumption of constant atmospheric density, the model is not capable of realistically handling cases with substantial differences in terrain height and major changes of the observer's altitude.

O'Neil also proposes a real time approach [6]. His model basically solves the scattering equations by low sampling in the vertex shader. The phase function is then applied in the pixel shader to avoid interpolation artifacts. The model produces correct scattering for observers inside and outside of the atmosphere, but is only considering single scattering. To achieve real time frame rates without any pre-computation, he is using a polynomial scale function to calculate the optical depth. However, this scale function is only valid for a fixed ratio between the radius of the earth, thickness of the atmospheric layer and scale height.

Schafhitzel et al. propose an approach of rendering planets with atmospheres by using a pre-computed table [11]. The proposed table stores the optical depth and is accessed by three parameters: the sun and view angle and the observer's altitude.

Bruneton et al. extends this approach with support for multiple scattering [1]. In addition, they introduce a new parameter by storing the angle between the view and sun direction. This allows for better parameterization, the reproduction of the earth's shadow in the atmosphere and the possibility to simulate lightshafts. This results in a four dimensional table, which stores the in-scattered light. The optical depth and the surface irradiance are stored in two separate tables.

3.2 Planetary Terrain Rendering

Visualizing a planetary terrain on many scales (e.g. on the planetary surface or in space - thousands of kilometers away) requires dynamic Level of Detail (LOD) algorithms to preserve details when near the ground. Most publications only deal with planar LOD terrain rendering, which is sufficient for most applications. To render spherical terrains (like whole planets) these algorithms have to be adapted accordingly.

A common approach is to form a cube of six planar terrains where each vertex is then normalized to create a unit sphere. The vertices are then multiplied by the planets radius and a corresponding height to form a planetary structure. O'Neil adapts this approach by using the traditional ROAM algorithm [12] to render spherical terrains [13]. Hill however shows that the ROAM algorithm is not reasonable for modern hardware and proposes a tile-based approach with terrain chunks of fixed resolution to create a planetary terrain [14]. His model uses a quad-tree approach to replace one tile with four if a certain threshold is exceeded. Cignoni et al. adapt their original BDAM algorithm [15] to spherical terrains [16]. The BDAM algorithm is capable of managing massive textured terrain data which is stored in a binary tree in a separate pre-processing step.

A notable exception to these approaches is proposed by Clasen and Hege [17] whose model does not rely on a cube as base geometry. Their implementation renders spherical terrains based on the GPU-Based geometry clipmap algorithm [18], which basically makes use of concentric rings rather than rectangles.

The planetary rendering proposed in this thesis is based on the work of Vistnes [19]. The model reuses a small vertexbuffer to render large terrains and has therefore minimal memory requirements. Although the model is intended for planar terrains only, it is shown that it can be easily adapted for rendering spherical terrains.

3.3 Deferred Rendering

Deferred rendering is an approach to rendering where shading calculations for pixel fragments are postponed until visibility is entirely determined. The idea of a deferred renderer was first proposed by Deering et al. in 1988 [20]. Intermediate geometric information is usually stored in a *geometry buffer* (GBuffer). This principle is based on the work of Saito et al. [21].
Over the past few years, deferred Rendering has been getting increasingly popular. A comprehensive overview of deferred rendering and a description of its advantages and disadvantages are given by Hargreaves [22] .

4 Deferred Rendering of Planetary Terrains with Pre-computed Atmospheres

This chapter proposes a planetary rendering model, which allows seamless transitions from space to the ground. The creation of the planetary surface is based on the work of Vistnes [19], which offers important features that allow for large scale terrain rendering like LOD and frustum culling capabilities. The terrain is generated procedurally and offers normal mapping and multi texturing to enhance realism when near the ground.
The rendered planet itself is surrounded by an atmosphere, similar to the earth's atmosphere, and accurately scatters the incident sunlight. The model is based on the work proposed by Bruneton and Neyret [1] and takes multiple scattering into account. To preserve interactive frame rates, the scattering equations are solved in a separate pre-computation step. The results are then stored in tables that are accessed during rendering. It is shown how atmospheric scattering can be applied in a single post-processing step that works on arbitrary scenes. The model assumes a common GBuffer that stores depth, color, normals and reflectance values.

At first the motivation for this approach is stated. After this, the requirements for this model on the underlying hardware and software are discussed. Furthermore the GBuffer of the proposed model is revealed along with basic considerations for storing and reconstructing geometrical information. Chapter 4.4 presents the planetary terrain model and introduces the reader to the implementation of key features like procedural generation of spherical terrains, LOD and frustum culling functionality, normal mapping and multi texturing. Chapter 4.5 shows how the sun is rendered. In chapter 4.6 the atmospheric scattering model is closely examined. It is shown how the look-up tables can be pre-computed on the GPU and how they can be used to apply accurate atmospheric scattering in real-time as a post-processing effect. Chapter 4.7 finally presents the results of the proposed model.

Chapter 4.4 - 4.6 provides various code samples of the actual implementation. These are written in the *High Level Shading Language* (HLSL) and in C++.

4.1 Motivation

Applying atmospheric scattering as a post-processing effect has several major advantages over traditional forward rendering. These are:

- Simple and straightforward integration into existing projects
- Complex shader permutations are prevented
- Predictable rendering costs

As atmospheric scattering is considered a very complex rendering effect, integration into an existing shader library can be cumbersome and time-consuming. In contrast, by applying it as a post-processing effect, integration is simple in comparison and completely detached from existing code. Another advantage is that the costs of applying this effect are completely independent from the complexity of the scene and the number of objects drawn. This makes the computation costs more predictable and does not add a constant rendering overhead to every object contained in the scene. Additionally, no specific GBuffer values are needed what makes this approach even more appealing and comfortable.

4.2 Requirements

Due to vertex texture fetches, dynamic branching and the high instruction count, shader model 3.0 is required at a minimum. In fact, shader model 4.0 is recommended as it introduces a geometry shader stage which enables writing to 3D textures directly by the GPU. Older shader models have to write every slice to a separate 2D texture and perform the merge on the CPU. However, these 3D textures are created in a pre-computation step and merging on the CPU would not affect the final performance during rendering.

The model also assumes support for *multiple render targets* (MRT). MRT refers to the capability to render to multiple textures at once, while performing a single draw call. Support for MRT heavily influences final performance due to the high computational expenses of certain vertex shader, which otherwise would need to be executed multiple times.

Memory consumption also needs to be considered as 3D textures quickly grow in size when increasing the resolution. For instance, the look-up tables used in the example implementation require slightly over 8 Mbytes. However memory consumption of the look-up tables is highly dependent on the accuracy needed. Several ways how the table sizes can be reduced are discussed in chapter 5.2.

The proposed implementation also makes use of the hardware depth buffer to store depth information. This buffer is set as an input texture for successive render stages. If this is not supported a separate floating point render target will be needed to store depth information.

4.3 Geometry Buffer Layout

As mentioned, the GBuffer used in the proposed model has a very common layout and does not require special components.

Red 8 Bit	**Green 8 Bit**	**Blue 8 Bit**	**Alpha 8 Bit**	**Description**
←	Depth 24 Bit	→	Stencil	Hardware Depth Stencil Buffer
Color red	Color green	Color blue	Color alpha	Color Buffer
Normal X	Normal Y	Normal Z	Reflectance	Normal & Reflectance Buffer

Figure 4-1: Layout of the GBuffer

As shown in Figure 4-1 the GBuffer is assumed to consist of two 32 bit render targets, which store the color and normal values and a hardware depth stencil buffer, which stores deph information.

The reflectance value is contained in the alpha channel of the normal buffer and is a common part of most GBuffers. Basically this value represents α of Equation 2-13. However the proposed model stores the full term $\frac{\alpha(x_0)}{\pi}$ in the buffer.

4.3.1 Linear Depth

Perspective projection of a 3D scene onto a 2D image involves a linear part and a non-linear part. The linear part is the multiplication of a vertex by the projection matrix, which stores the original z component of the vertex in the w component. After this, the resulting components of the vertex (x,y and z) are divided by the w component (the original value of z before multiplication by the projection matrix). This represents the non-linear part and happens automatically between the vertex shader stage and the pixel shader stage. This operation is often referred to as the *homogeneous divide* or the *perspective divide* and is a non-linear function which enables the hardware's depth-buffering algorithm by mapping the resulting z component to the range $[0,1]$ (in the case of Direct3D).

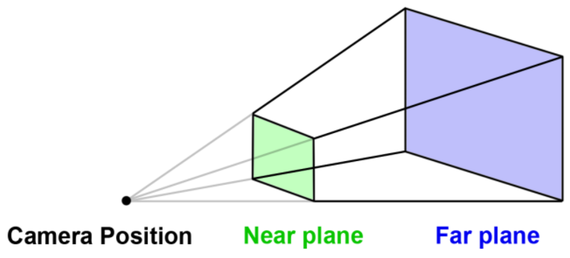

Figure 4-2: View Frustum and its near and far plane

The linear and non-linear parts represent a function $g(z)$ that maps a depth value between the far plane f and the near plane n (Figure 4-2) to the range $[0,1]$ and is given as follows:

$$g(z) = \frac{f}{f-n} - \frac{nf}{(f-n)z}$$ Equation 4-1

Although the function described in Equation 4-1 is strictly increasing and order preserving, the resulting graph is non-linear. Figure 4-3 shows the resulting non-linear depth values for varying near planes.

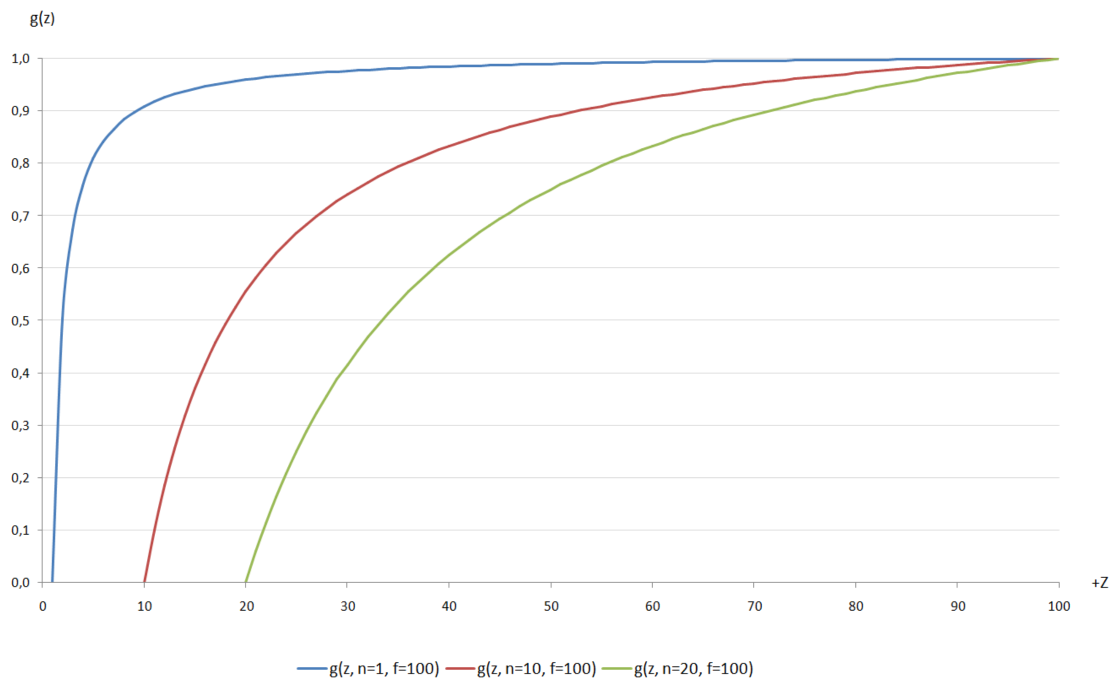

Figure 4-3: Resulting non-linear depth for varying near planes

As shown, the majority of the resulting depth range is consumed by depth values close to the near plane. This can lead to precision problems when the ratio between the near and far plane increases.

It is very common in deferred rendering, to reconstruct the position of a pixel by making use of its depth stored in the depth buffer. The accuracy of reconstruction is however dependent on the precision of these depth values. In order to allow visualization of a planet on many scales (at the ground or thousands of kilometers away), the ratio between the near and far plane has to be set accordingly.

4.3.1.1 Storing Linear Depth

In order to solve the inevitable precision problems described in the previous chapter, depth is stored linearly, which means that the accuracy of depth values is not dependent on its distance to the near plane. Linear depth distribution can be obtained by multiplying the z

component of the vertex by the w component and the reciprocal of the far plane f before the perspective divide. This finally results in a function $g'(z)$ as follows:

$$g'(z) = \frac{z-n}{f-n} \qquad \text{Equation 4-2}$$

Solving Equation 4-2 leads to a linear distribution of the resulting depth values and hence to constant precision over the whole range.

4.3.1.2 Reconstructing the Original Position

The original position of a fragment in the depth buffer can be reconstructed easily when the corresponding positions on the near and far plane are known. As the depth values are distributed equally between the near and far plane, the position can be recreated by solving Equation 4-3.

$$\begin{aligned} OriginalPosition = CameraPosition + CameraToNear + \\ DepthVal * NearToFar \end{aligned} \qquad \text{Equation 4-3}$$

$$CameraToNear = NearPlanePosition - CameraPosition$$
$$NearToFar = FarPlanePosition - NearPlanePosition$$

The position is obtained by offsetting the camera position by the distance to the near plane and a linear interpolation between the near and far plane according to the depth value stored in the depth buffer. Figure 4-4 shows an example of reconstructing the position of a point having a depth value of 0.45.

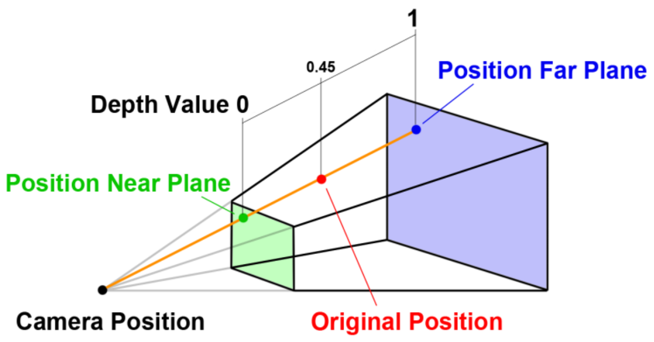

Figure 4-4: Reconstructing the position of a point having depth value 0.45 by interpolating between the near and far plane

4.3.2 Encoding of Surface Normals

A common technique used in deferred renderers is to store only two components of the normal in the GBuffer. As the normal has unit length the third component can be reconstructed. In this case memory consumption is traded for computational costs. This however yields to slight errors [23].

The simplest approach to encoding assumes that only normals, which face in the direction of the camera, are seen. By storing the x and y components of the normal in view space the z component can be reconstructed by making use of the fact that the following equation is true for normalized vectors $x^2 + y^2 + z^2 = 1$. Thus, the z component is retrieved by solving this equation for z . The appropriate sign of the z component is then assumed due to the fact that visible normals can only point towards the camera. However, normals can also point away from the camera due to perspective projection [24]. In this case the decoding fails and can produce errors, which are subtle and therefore hard to detect. A good and fast alternative to this is presented by Mittring [25], which even offers better precision.

However, in favor of simplicity, the proposed implementation stores all three components of the normal in the buffer. Usually the components of the normal vector are stored in 16 bit channels. Storing these in 8 bit channels produces quantization errors, which are most apparent when specular lighting is involved. In the presented approach 8 bits are sufficient, as specular lighting is ignored and the resulting errors are hardly notable.

4.4 Planetary Terrain Rendering

The following chapters give a comprehensive overview of the key concepts used to render the planetary terrain. As already mentioned, the model is based on a terrain rendering algorithm originally proposed by Vistnes [19].

At first the basic algorithm proposed by Vistnes [19] is presented. Chapter 4.4.2 then shows how this model can be adapted to allow rendering of procedural planetary terrains, while preserving its LOD functionality. Furthermore, the algorithms behind normal mapping and multi texturing are revealed and it is shown how the final values are rendered into the GBuffer. Chapter 4.4.4 finally describes how frustum culling can be implemented efficiently and discusses the resulting performance gain.

The full source code of the planetary terrain shader is provided in Appendix A.

4.4.1 Basic Algorithm

This chapter serves as a brief introduction to the basics of the planar terrain rendering algorithm originally proposed by Vistnes [19].

The terrain algorithm is based on a quad-tree approach to implement its LOD functionality. Imagine a simple block of triangles that lies in the x, z plane of a coordinate system. This block can be recursively divided into a number of smaller quadratic blocks. Each division results in four children, which are four times smaller than the parent. Each block is represented by a fixed quantity of vertices. The number of vertices along an edge of the quadratic block describes the $Blocksize$ of the quad-tree. Parts of the plane that are divided more often have therefore a higher triangle density. The number of divisions represents the LOD of this part of the terrain.

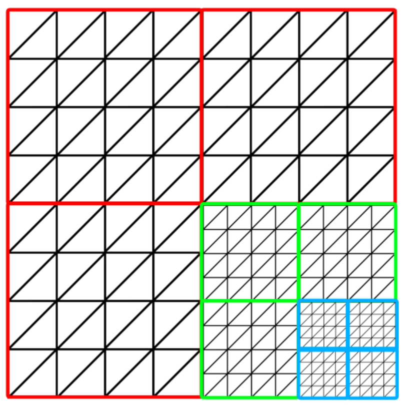

Figure 4-5: Divided blocks with different LOD

Figure 4-5 shows a plane with different LOD. The red outlined blocks are the result of the first division. The bottom, right block then got divided into four child blocks (green outline). Finally the bottom, right block of these children got divided again into four additional children (blue outline). As each block contains an equal number of triangles, the bottom right corner has the highest density of triangles.

The primary advantage of this model is the very low memory requirements when implemented with shaders. In this case it's possible to render an unlimitedly sized terrain by using a very small vertex buffer, which represents a single block.

4.4.1.1 Calculating the position

The vertices of each block originally form a planar quadratic block that lies flat on the x, z plane. The values of the vertices along the x and z axis range from 0 to 1. The position of a vertex within this coordinate system is from now on referred to as Pos_{Block}.

The terrain is composed of many different blocks with different scales, as described above. Therefore the vertices of the block need to be positioned on the right location within this particular terrain. To accomplish this, the blocks are transformed into a new coordinate system, which is from now on refered to as the *uvw coordinate system*. At first the u and v coordinates are calculated. These u and v coordinates can be determined by calculating three values: a $size$, a $uMin$ and a $vMin$ value. $size$, as the name implies, describes the size of a block and thus the length of an edge along the u and v axis. The $uMin$ and $vMin$ values describe the offset of the block from the origin along the u and v axis respectively. The values of the u and v axis range from 0 to 1. The positioning of a block on the u and v axis is shown in Figure 4-6. The highlighted block for example has a $size$ of 0.25, a $uMin$ value of 0.5 and a $vMin$ value of 0.25.

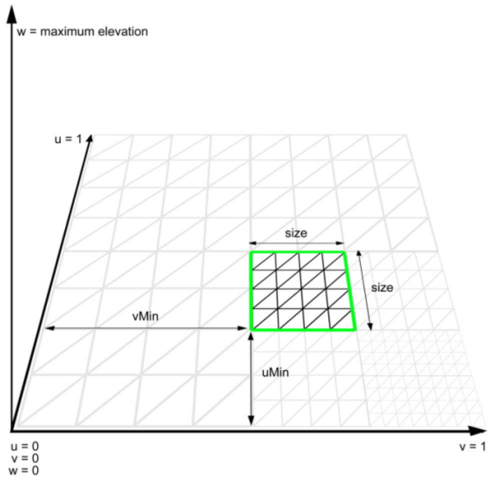

Figure 4-6: Positioning of a block within the uvw coordinate system using bias values and a size

The w axis describes the elevation of the terrain. The original algorithm uses a height field texture to elevate vertices. Note that the uv coordinates of the vertices range from 0 to 1 and so they can be used as texture coordinates to obtain an elevation value from a height map.

The uvw coordinates of a vertex can therefore be calculated as described in Equation 4-4.

$$\begin{aligned} pos_{UVW}.u &= pos_{Block}.x * size + uMin \\ pos_{UVW}.v &= pos_{Block}.z * size + vMin \\ pos_{UVW}.w &= elevation \end{aligned}$$

Equation 4-4

pos_{UVW} describes the position of the vertex in the uvw coordinate system and $elevation$ the elevation of the vertex along the w axis, as described above.

After obtaining a position within the uvw coordinate system, the vertices are transformed to world space. This is usually done by a scaling matrix that scales the vertices to the desired dimension of the terrain in world space.

Some of the algorithms in the following chapters need to calculate certain positions on a single block. These positions are often described relative to the uvw coordinate system with the aid of certain values. These values are shown in Figure 4-7 and given as described in Equation 4-5.

$$uMid = \frac{size}{2} + uMin \qquad vMid = \frac{size}{2} + vMin$$

$$uMax = size + uMin \qquad vMax = size + vMin$$

Equation 4-5

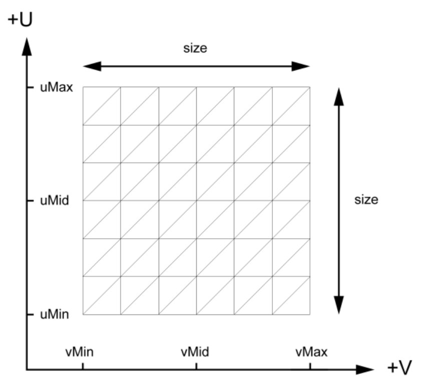

Figure 4-7: Important values of a block on the u and v axis used by successive algorithms

4.4.1.2 Implementing Level of Detail Functionality

When a block gets divided into four children, the new values for $size$, $uMin$ and $vMin$ are calculated for each children. The values are given as described in Equation 4-6.

$$\begin{aligned} size_{Child1,Child2,Child3,Child4} &= \frac{size}{2} \\ uMin_{Child1} &= uMin, \qquad vMin_{Child1} = vMin \\ uMin_{Child2} &= uMid, \qquad vMin_{Child2} = vMin \\ uMin_{Child3} &= uMin, \qquad vMin_{Child3} = vMid \\ uMin_{Child4} &= uMid, \qquad vMin_{Child4} = vMid \end{aligned}$$ Equation 4-6

The actual decision, if a block is divided into four children, is based on the simple test shown in Equation 4-7 [19].

$$\frac{l}{d} < C$$ Equation 4-7

Where the value l denotes the distance from the center of the block to the camera, d the world space extend of a single triangle and C an adjustable constant that controls the quality and therefore the number of divisions of the rendered terrain. If the test is true, the current block will be divided into four children. If it fails, the current block will be rendered. This evaluation is an adaption of an idea introduced by Röttger et al. [26]. Note that a higher value for C results in more divisions and therefore in a higher triangle density near the camera. The maximum number of divisions is usually limited by a threshold value.

4.4.1.3 Avoiding Cracks

A common drawback of most terrains algorithms which offer LOD functionality is that cracks appear at the transitions of blocks with different LOD. A simple solution to this problem is proposed by Ulrich [27]. In his model, the vertex buffer of the block is extended by so called *skirt vertices* around the border of the block. Then the original vertices (inside the skirt) are initialized with a y position of 1 and the skirt vertices with a y position of -1. Note that the y component of each vertex was unused till now, as the elevation is applied in the uvw coordinate system. The assignment of $pos_{UVW}.w$ in Equation 4-4 is then changed to a multiplication as shown in Equation 4-8.

$$pos_{UVW}.w = pos_{Block}.y * elevation$$ Equation 4-8

This creates a vertical skirt (vertices with negative elevations) at the borders of each block as shown in Figure 4-8.

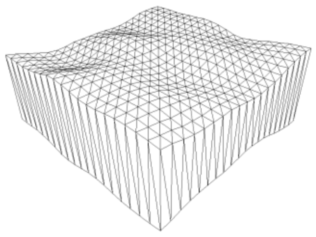
Figure 4-8: The borders of a block are extended with a vertical skirt

As a result the cracks are filled by a vertical skirt. Although, this approach can cause some lighting problems and texture stretching near the borders of a block, the crack fillings are usually too small to be distracting [19].

4.4.2 Generating a Planetary Terrain

As briefly introduced in chapter 3.2 most planar terrain rendering algorithms can be adopted to render planetary terrains. The following sections describe how the terrain rendering model described in the previous chapter can be adapted to allow a creation of spherical terrains.

4.4.2.1 Generating a Spherical Terrain out of a Cube

Generating spherical terrains can be achieved by forming a cube out of six planar terrain surfaces, where the center of the cube is in the origin of the coordinate system. After this, the vertices of each surface are normalized, which results in a perfect unit sphere. Finally, each vertex is extruded by an elevation factor and the radius of the planet itself. This approach is an adaption of an approach proposed by O'Neil [13] and is shown in Figure 4-9.

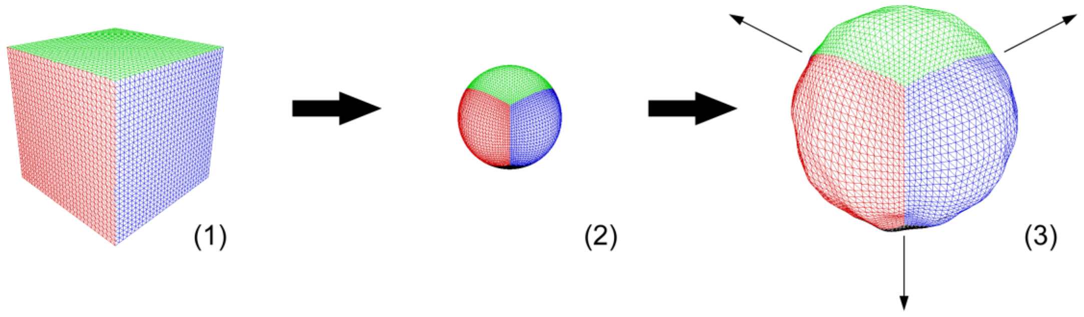

Figure 4-9: The process of modeling spherical terrains: (1) forming a cube out of six planar terrains (2) normalizing each vertex and (3) finally extrusion by planetary radius and an elevation factor

In the following chapters positions are described in three different spaces.

- In step (1) the positions on each face of the cube are described relative to one of the six terrains. These positions are described by the uvw coordinate system, which was introduced in chapter 4.4.1.1. An important change however is the fact, that the elevation value is not applied in the uvw coordinate system anymore. Instead this is done in step (3).
- The normalized positions of step (2) are described as positions on the *unit sphere*.
- The resulting positions after step (3) are the positions in world space.

Note that the position on the unit sphere can be retrieved by normalizing the position in world space.

In the proposed implementation, the three steps shown in Figure 4-9 are carried out in a vertex shader as shown in Listing 4-1.

```
// input - posBlock: position within the vertex buffer of the block
// output: position after transformation to uvw space
float3 getUVW(in float3 posBlock)
{
  float3 posUVW;
  posUVW.x = posBlock.x * g_sizeBlock + g_uMin;
  posUVW.y = 0.0f;
  posUVW.z = posBlock.z * g_sizeBlock + g_vMin;

  return posUVW;
}

...

// transform position to uvw space
float3 posUVW = getUVW(input.pos);

// transform to cube and normalize to obtain position on unit sphere
float3 posCube = mul(float4(posUVW, 1.0f), g_cube).xyz;
float3 posUnitSphere = normalize(posCube);

// extrude by planetary radius and an elevation factor to obtain
// position in world space
float elevation = getElevation(posUnitSphere) * input.pos.y;
float3 posWorld = posUnitSphere * (g_Rg + elevation);
```

Listing 4-1: Calculating planetary position in vertex shader

At first the uvw coordinates of the current vertex is obtained. This basically positions the block within the planar terrain. By multiplying these blocks with a certain matrix they are rotated and translated to form a cube. The resulting positions on the cube are then normalized to obtain the position on the unit sphere. This position is then scaled again by the radius of the planet R_g and an elevation factor. The called function to obtain an elevation factor for a given position is explained in further detail in chapter 4.4.2.3. Note that the skirt vertices are also extruded by the planetary radius but get a negative elevation to fill the cracks.

4.4.2.2 Adapting the Level of Detail Functionality

LOD functionality needs to be adapted due to the fact that the blocks of the terrain are now curved. For this the calculations of d, the world space extend of a single triangle, need to be adjusted to approximate the arc length of a block in world space.

This approximation is done by transforming the positions $P0$, $P1$ and $P2$ as depicted in Figure 4-10 from the uvw coordinate system into world space.

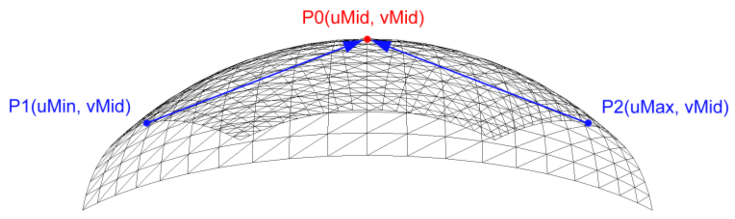

Figure 4-10: Approximating the arc length of a block

This results in the corresponding points with respect to world space: $P0_{World}$, $P1_{World}$ and $P2_{World}$. The final world space extend d of a single triangle is then given as shown in Equation 4-9.

$$arcLength = length(P1_{World} - P0_{World}) + length(P2_{World} - P0_{World}) \qquad \text{Equation 4-9}$$

$$d = \frac{arcLength}{Blocksize - 1}$$

Where $length$ calculates the length of the given vector.

Note that, due to performance reasons, the actual elevation of the vertex is not taken into account when calculating the arc length. The planet is thus treated as a perfect sphere. However, this is negligible as the arc length is approximated anyway.

4.4.2.3 Generating Elevations using Perlin Noise

As the planet is visualized on many different scales, a function is needed that allows generating rough mountain ranges that are visible from far away as well as very fine bumps when the camera is near the ground.

It is common to elevate terrains using a height map texture. The drawback of this approach is that a height map is limited in its resolution. In order to preserve details when near the ground (and prevent an oversampling of the texture) a height map with an unreasonably high resolution would be needed. To overcome this problem the planetary surface is elevated procedurally by a function that allows retrieving both: a rough surface pattern as well as fine details.

The noise function proposed by Perlin [28] is one of the most important noise functions to create procedural content. The function offers controlled randomness, which means, that the same input always produces the same output. What makes it particularly useful for rendering large scale terrains is the fact, that it outputs smoothed noise, rather than discrete noise for values that are continuous (for example texture coordinates). Increasing and decreasing the inputted value range looks like zooming in and out of the resulting noise. It doesn't matter how far you zoom in – the outputted noise is always perfectly smooth.

This allows generating a rough, low frequency noise by supplying values in a lower range and high frequency details by supplying values in higher ranges. The resulting values can also be summed up to generate a so called *fractal sum* as shown in Figure 4-11.

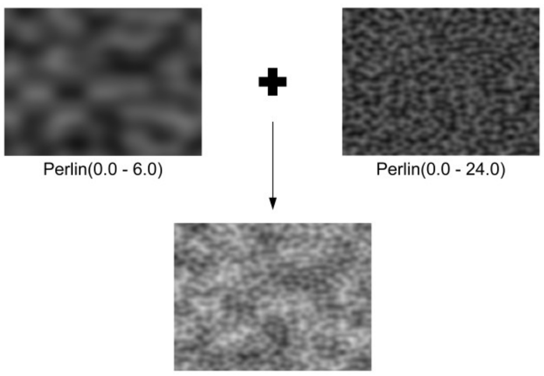

Figure 4-11: Fractal sum of two 2D Perlin noise functions with different ranges for x and y

The dimension of a Perlin noise function determines for how many dimensions smoothed values can be generated. For generating the planetary terrain, a 3D Perlin noise function is used to generate a smooth and random surface elevation. Although the Perlin noise function is considered a fast algorithm for the resulting quality, it is still quite expensive.

In the proposed model an elevation in the range $[0,4]$ is generated for each vertex by combining the results of two Perlin noise functions. Listing 4-2 shows a function which returns a random but perfectly smooth elevation value depending on the position on the unit sphere.

```
static const float g_lowFrequencyHeight = 3.25f;
static const float g_highFrequencyHeight = 0.75f;
static const float g_lowFrequencyScale = 400.0f;
static const float g_highFrequencyScale = 2500.0f;

...

// input - posUnitSphere: position on the unit sphere
// output: elevation value generated by fractal sum
//         of high and low frequency Perlin noise
float getElevation(in float3 posUnitSphere)
{
  float noiseHeight = 0;

  // low frequency noise
  float n = Perlin(posUnitSphere * g_lowFrequencyScale);
  n = shift(n);
  noiseHeight += n * g_lowFrequencyHeight;

  // high frequency noise
  n = Perlin(posUnitSphere * g_highFrequencyScale);
  n = shift(n);
  noiseHeight += n * g_highFrequencyHeight;

  return noiseHeight;
}
```

Listing 4-2: Generating an elevation value

The Perlin noise function returns a smooth random value between -1 and 1. This value is then shifted to the range $[0,1]$ and scaled by a constant that determines the maximum elevation for this frequency. This is done two times to generate low and high frequency noise. Note that the second time, the input value is scaled by a considerably greater value to obtain high frequency noise. Then the fractal sum is calculated by combining high and low frequency noise.

Figure 4-12 shows the results of the noise function and demonstrates the effectiveness of combining high and low frequency noise. The importance of high frequency noise becomes more apparent when approaching the planetary surface.

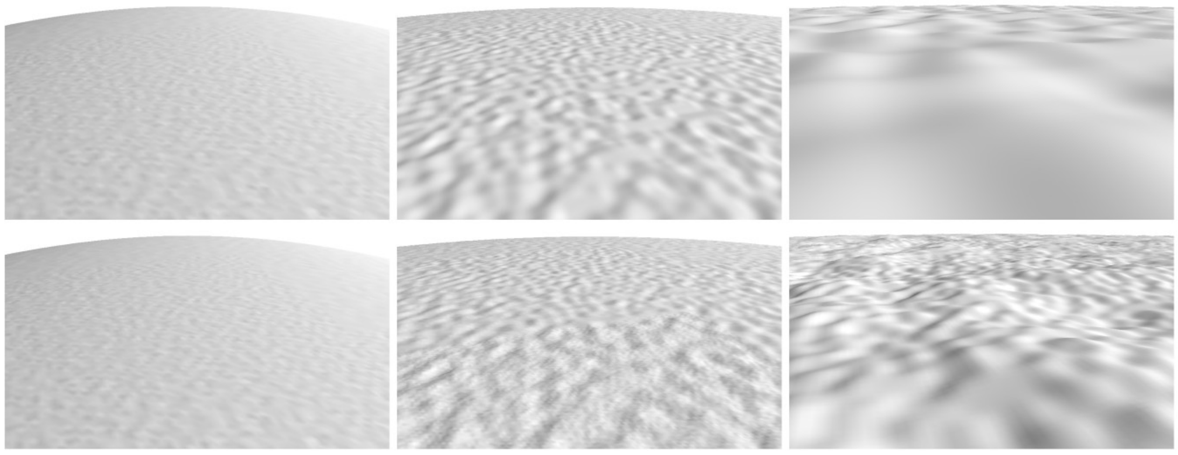
Figure 4-12: Importance of high frequency noise
(top row) using only low frequency noise
(bottom row) combining low and high frequency noise

4.4.2.4 Texturing

A problem with spherical terrains is the fact that a sphere cannot be textured by a rectangular image without any distortions. This problem is a direct result of the so called *hairy ball theorem* [29]. Fortunately, the original texture coordinates of the planar terrain (as described in chapter 4.4.1.1) offer minimal distortions (like stretching at the center of each face and seams at edges between each terrain) and can be reused.

Note that the coordinates in terrain space range from 0 to 1 and would stretch a single texture over a whole terrain, so these coordinates are scaled accordingly to repeat the texture several times. This however yields to visible repeating texture patterns on distant terrains. In order to reduce these patterns the scaling of these texture coordinates are a function of the surface's distance to the camera as proposed by Rosa [30].

```
// controls which distance is considered as Near or Far
static const float g_textureDistNear = 100;
static const float g_textureDistFar = 1000;
// scale factors for textures
static const float g_grassTexScaleNear = 600.0f;
static const float g_grassTexScaleFar = 8.0f;
static const float g_stoneTexScaleNear = 600.0f;
static const float g_stoneTexScaleFar = 32.0f;
// value between 0 and 1 that controls the height
// where the stone texture starts to blend in
static const float g_stoneColorStart = 0.4f;
// factor that controls the softness of
// transition from grass to stone texture
```

```
static const float g_stoneColorTransition = 3.5f;

// input - texC: texture coordinates in the range [0,1]
// input - positionWorld: position in world space
// output - return value: color of surface
float3 getSurfaceColor(in float2 texC, in float3 positionWorld)
{
  float distCamToSurface   = length(g_cameraPos - positionWorld);
  float distForTextures = saturate((distCamToSurface - g_textureDistNear)
                                  / (g_textureDistFar - g_textureDistNear));

  // lerping between near and far texture color
  float4 grassNear = g_grass.Sample(LinearSamplerWrap, texC *
                     g_grassTexScaleNear);
  float4 grassFar  = g_grass.Sample(LinearSamplerWrap, texC *
                     g_grassTexScaleFar);
  float4 grass = lerp(grassNear, grassFar, distanceForTextures);

  float4 stoneNear = g_stone.Sample(LinearSamplerWrap, texC *
                     g_stoneTexScaleNear);
  float4 stoneFar  = g_stone.Sample(LinearSamplerWrap, texC *
                     g_stoneTexScaleFar);
  float4 stone = lerp(stoneNear, stoneFar, distanceForTextures);

  // lerping between grass and stone texture
  float n = Perlin(normalize(positionWorld) * g_lowFrequencyScale);
  n = shift(n);
  n = saturate(n - g_stoneColorStart);
  n = saturate(g_stoneColorTransition * n);

  return lerp(grass.rgb, stone.rgb, n);
}
```

Listing 4-3: Determining the color of the surface by interpolation between different texture sets as a function of the surface's distance to the camera

Listing 4-3 shows how the final surface color is determined. In addition to the interpolation based on the distance from the camera to the surface, the color is also interpolated between two different texture sets depending on the height of the surface. This allows texturing higher elevated surfaces with a different set than lower ones. This is particularly useful to increase the impression that high surfaces are indeed mountains.

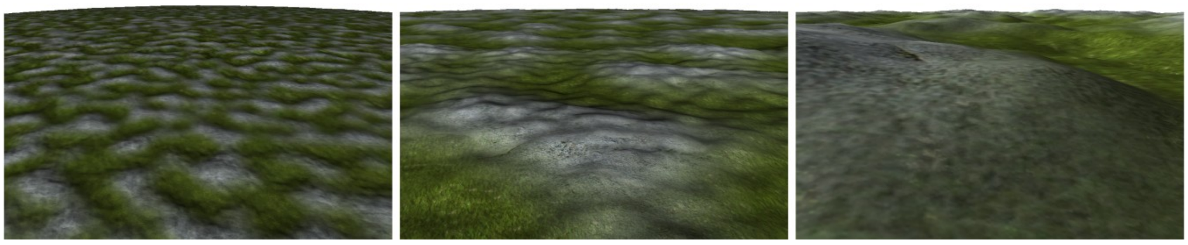

Figure 4-13: Multi textured terrain on varying scales

The final texturing of the planet is shown in Figure 4-13. Higher surfaces are textured using a stone texture while lower ones use a grass texture.

4.4.2.5 Calculating Surface Normals

In real-time rendering, the illumination of a surface by a light source is usually calculated by using the dot product as described in Equation 4-10.

$$lightScale(\theta) = \max\,(l.n, 0)$$ Equation 4-10

Where $lightScale$ describes a function that scales the intensity of light per unit area according to the cosine angle between the surface normal n and the direction vector to the light source l. This principle is depicted in Figure 4-14.

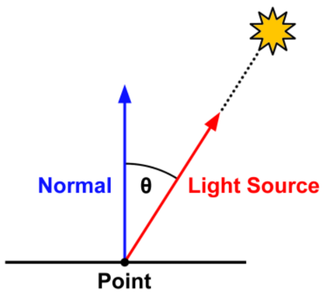

Figure 4-14: Illumination of a unit area is scaled by the cosine angle between the normal vector and the direction vector to the light source

The normal can be understood as the direction a point on a surface faces. Usually, these normals are stored for each vertex and are pre-generated in most cases. This means that they are part of the 3D model that is loaded for rendering. In the case of procedurally generated objects (like the planetary terrain), the normal for each vertex has to be calculated manually. The normal can be calculated, when the slope of a particular point on the surface is known. For this, two perpendicular tangents are calculated, where each tangent matches the slope of the surface in one particular direction. These tangents are usually referred to as the *tangent* and *bitangent* vector.

Tangent and bitangent are calculated by considering the positions of the four neighboring vertices in the north, east, south and west direction as described by Equation 4-11.

$$tangent = normalize\ (\,NeighborE - NeighborW\,)$$
$$bitangent = normalize\ (\,NeighborN - NeighborS\,)$$ Equation 4-11

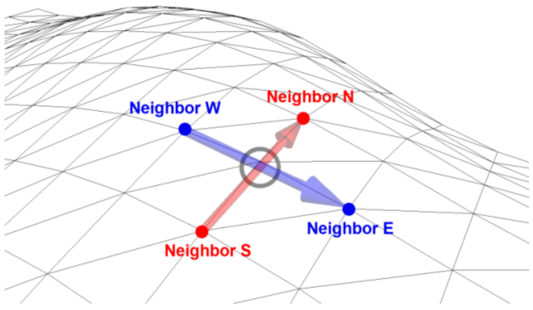

Figure 4-15: Tangents of a vertex calculated by examining the positions of the neighbors in the north, east, south and west direction

Figure 4-15 depicts this operation for a single vertex. As the terrain is generated procedurally, the position of neighboring vertices can be retrieved at any time when the distance between the vertices in the local coordinate system of a single block is known. In fact this distance can be calculated by Equation 4-12.

$$vertexDistance = \frac{1}{Blocksize - 1} \qquad \text{Equation 4-12}$$

Having the positions in the local coordinate system of a block, they can be transformed to world space. Listing 4-4 shows how this is done in the corresponding vertex shader.

```
// input - posBlock: position within the vertex buffer of the block
// output - posS: position of south neighbor in world space
// output - posN: position of north neighbor in world space
// output - posW: position of west neighbor in world space
// output - posE: position of east neighbor in world space
void GetNeighborPos(in float3 posBlock, out float3 posS, out float3 posN, out
float3 posW, out float3 posE)
{
  posS = getUVW(posBlock - float3(0.0f, 0.0f, g_vertexDistance));
  posN = getUVW(posBlock + float3(0.0f, 0.0f, g_vertexDistance));
  posW = getUVW(posBlock - float3(g_vertexDistance, 0.0f, 0.0f));
  posE = getUVW(posBlock + float3(g_vertexDistance, 0.0f, 0.0f));

  posS = normalize(mul( float4(posS, 1.0f), g_cube ).xyz);
  posN = normalize(mul( float4(posN, 1.0f), g_cube ).xyz);
  posW = normalize(mul( float4(posW, 1.0f), g_cube ).xyz);
  posE = normalize(mul( float4(posE, 1.0f), g_cube ).xyz);

  posS *= g_Rg + getElevation(posS);
  posN *= g_Rg + getElevation(posN);
  posW *= g_Rg + getElevation(posW);
  posE *= g_Rg + getElevation(posE);
}
```

Listing 4-4: Calculating the neighbors' positions

$g\_vertexDistance$ describes the spacing of the vertices within the local space of a block as described in Equation 4-12.

The actual slope is the difference between two opposing neighbors as described in Equation 4-11 and shown in Listing 4-5.

```
// obtain tangents
float3 neighborS, neighborN, neighborW, neighborE;
GetNeighborPos(input.pos, neighborS, neighborN, neighborW, neighborE);

output.tangent   = normalize(neighborE - neighborW);
output.bitangent = normalize(neighborN - neighborS);
```

Listing 4-5: Calculating the tangent and bitangent for each vertex

Once the neighboring positions are obtained, the normal can be calculated as described in Equation 4-13.

$$normal = tangent \times bitangent$$ Equation 4-13

This means that the vertex normal is orthogonal to both, the tangent and the bitangent vectors of this particular point. Having the vertex normal, it is possible to calculate the illumination by a light source as described in Equation 4-10.

4.4.2.6 Applying Normal Mapping

Instead of using the calculated vertex normal directly in Equation 4-10 it is used to apply *normal mapping* [31]. Normal mapping allows the simulation of fine details and bumps on the surface of an object without the need to increase the number of vertices. This is accomplished by using so called *normal maps*.

Normal maps are textures, which store perturbed normals and therefore allow obtaining an individual normal for each pixel. These normals represent 3D vectors in a local coordinate system, usually referred to as the *tangent space*, where the red, green and blue channel of each texel describes the value on the appropriate axes. The tangent space is usually individual for each vertex and depends on the actual slope of the surface on this particular position. The basis vectors of this tangent space are given by the normal, tangent and bitangent vectors which are calculated as described in chapter 4.4.2.5. Figure 4-16 shows the tangent space for three different vertices on a curved surface.

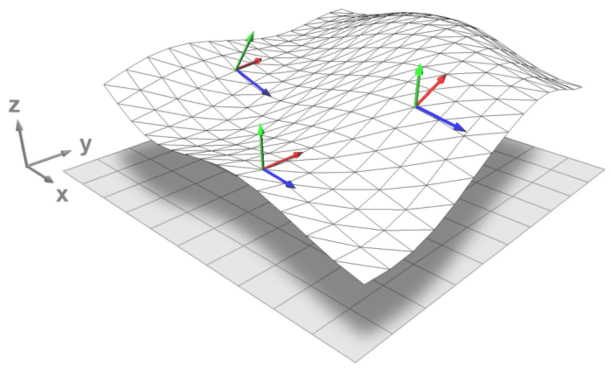

Figure 4-16: Normal, tangent and bitangent build up a tangent space on each vertex

Before the normals of a normal map can be used for illumination calculations, as described in Equation 4-10, they need to be transformed to world space accordingly. For this a matrix is generated that transforms these vectors from tangent space to world space. This matrix is commonly referred to as the *TBN* matrix. Its contents are shown in Equation 4-14.

Equation 4-14

$$TBN = \begin{bmatrix} Tangent.x & Bitangent.x & Normal.x \\ Tangent.y & Bitangent.y & Normal.y \\ Tangent.z & Bitangent.z & Normal.z \end{bmatrix}$$

For this the tangents calculated in the vertex shader (as described in chapter 4.4.2.5) are passed to the pixel shader where the TBN matrix is built.

To ensure orthogonality of the tangent vectors, the *Gram-Schmidt orthogonalization procedure* [32] (named after Jorgen P. Gram and Erhard Schmidt, 1983) is used. This method allows transforming a set of vectors into an orthogonal basis by using a simple projection. Listing 4-6 shows the Gram-Schmidt orthogonalization of the tangent vectors, the calculation of the normal vector and the construction of the TBN matrix.

```
// set up tangent-to-world matrix
tangent   = normalize(tangent);
bitangent = normalize(bitangent);
bitangent = normalize(bitangent - dot(bitangent, tangent) * tangent);
float3 normal  = normalize(cross(bitangent, tangent));
float3x3 TBN   = float3x3(tangent, bitangent, normal);
```

Listing 4-6: Setting up the TBN matrix using the Gram-Schmidt orthogonalization procedure and the cross product of the tangents

Listing 4-7 shows the process of retrieving and transforming normals from the normal map to world space.

```
float3 normalTangent = g_normal.Sample( LinearSamplerWrap, texC *
g_normalTexScale).xyz;
normal = normalize(mul(2.0f * normalTangent - 1.0f, TBN));
```

Listing 4-7: Transforming normals retrieved from the normal map to world space

Note that the coordinates of the normal map are also scaled (as described in chapter 4.4.2.4) to prevent stretching. The stored vectors of a normal map basically define a vector on a unit sphere. Usually these vectors are encoded and stored in the range of $[0, 1]$. This means that the retrieved values of the texture need to be mapped to the range $[-1,1]$ accordingly, prior their transformation to world space.

Normal mapping is most notable when the camera is very close to the surface. As shown in Figure 4-17 it allows adding fine details to the terrain without the need to increase the required number of vertices.

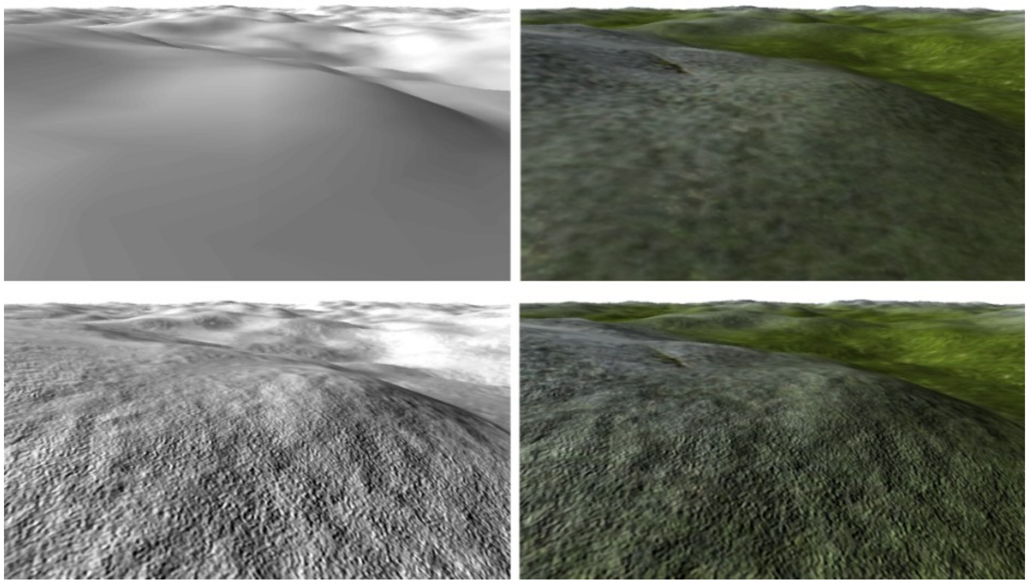

Figure 4-17: Surface details added through normal mapping (single colored and textured terrain)
(top row) lighting using single normal per vertex
(bottom row) perturbed normals retrieved from normal map

4.4.3 Rendering to the Geometry Buffer

Finally, the terrain is rendered into the GBuffer. As described in chapter 4.3, the GBuffer stores color, normal, depth values and a reflectance value.

Recall that the proposed implementation uses the hardware depth buffer to store linear depth values. Hence, linearization has to be considered in the vertex shader. For this the z component of the projected vertex is multiplied by the w component and the reciprocal of the far plane as described in chapter 4.3.1.1. Listing 4-8 shows how this is done in the vertex shader before the perspective divide that happens automatically between the pixel and the vertex shader stage.

```
    output.posH   = mul(float4(posWorld, 1.0f), g_viewProj);
    output.posH.z = output.posH.z * output.posH.w * g_invFarPlane;
```

Listing 4-8: Forcing linear depth with the hardware depth buffer

The depth buffer will be filled automatically by the hardware. The remaining values are set by the pixel shader as shown in Listing 4-9.

```
float3 color = getSurfaceColor(input.texC, input.posW);
float3 normal = getNormalVector(input.tangent, input.bitangent, input.texC);

output.color.rgb  = color;
output.color.a    = 1.0f;
output.normal.xyz = 0.5f * normal + 0.5f;
output.normal.w   = AVERAGE_GROUND_REFLECTANCE / M_PI;
```

Listing 4-9: Storing the color, normal and reflectance value in the GBuffer

The color and normal values are obtained as described in chapter 4.4.2.4 and 4.4.2.6 respectively. Note that the normal value needs to be shifted from the range $[-1, 1]$ to $[0, 1]$ as the corresponding texture can only store unsigned integer values. The reflectance value is set to the constant ground reflectance divided by π as described in Equation 2-13.

Figure 4-18 shows the contents of the GBuffer after rendering.

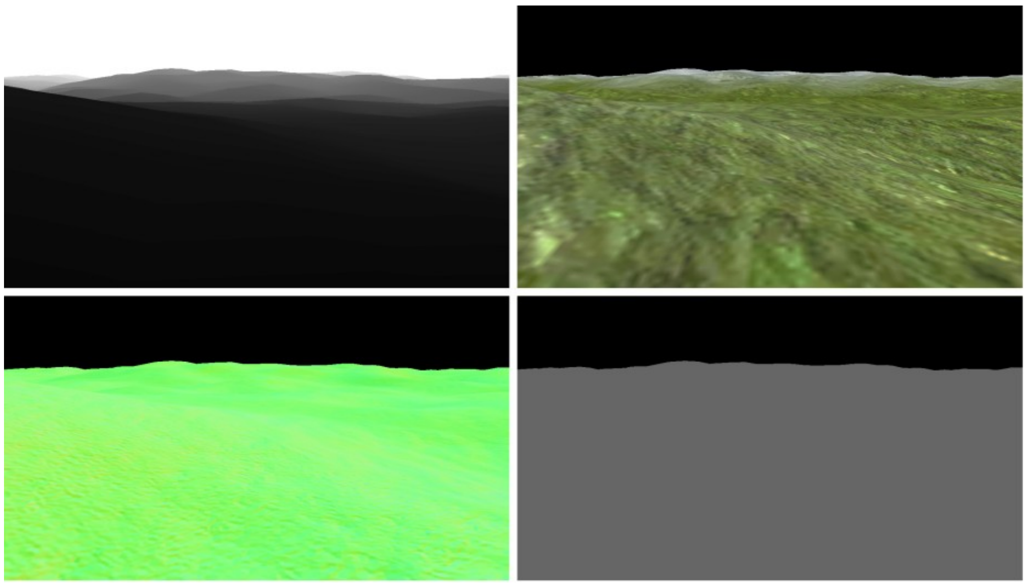

Figure 4-18: Filled GBuffer (top row): depth buffer and color buffer
(bottom row) normal and reflectance values
(depth and reflectance colors are scaled for better visualization)

4.4.4 Enabling Frustum Culling

When the camera is near the surface, only a fraction of the total planetary geometry is within the cameras view frustum and hence only a small part of it contributes to the final image. A frustum culling algorithm is used, to reduce the geometry sent to the GPU. This prevents processing of invisible terrain blocks.

4.4.4.1 Calculating the Axis Aligned Bounding Boxes

Frustum culling is basically an intersection test that determines if certain objects lie within or outside the view frustum. Testing the actual geometry of these objects for intersection is often computational expensive and thus not possible in real-time. Instead it’s common to encapsulate the actual geometry of these objects in so called *bounding volumes*. A bounding volume describes a tight fitting geometrical object that allows for inexpensive intersection and collision tests [33]. For this, each block of the terrain gets encapsulated by a so *called axis aligned bounding boxs* (AABB), which acts as the bounding volume for intersection tests with the view frustum. In general, an AABB (in 3D) is a six sided rectangular bounding volume where the normals of each face are at all times parallel with the axes of a given coordinate system, in this case the world space. An AABB can be described by only two points, where one point has minimum and the other one has maximum coordinate values along each axis [33]. Figure 4-19 shows a generated AABB for a single terrain block.

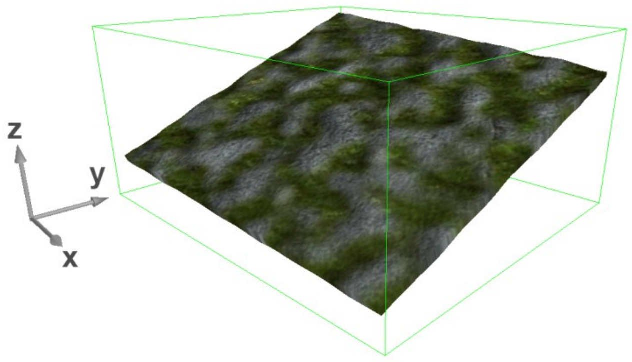

Figure 4-19: AABB of a single terrain block

To calculate the AABB, nine vertices of each block need to be considered, as these can have either minimum or maximum values in world space, depending on the block's orientation. These nine vertices (shown in Figure 4-20) need to be transformed to world space. The maximum and minimum values with respect to world space are then found by a simple comparison between the values on each axis.

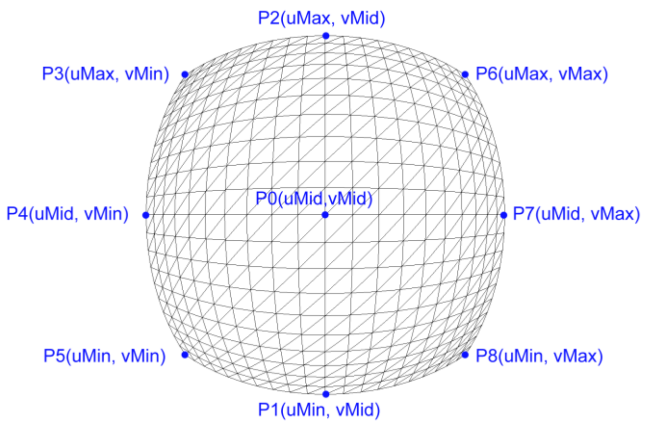

Figure 4-20: To calculate an AABB nine positions for each block are considered

Note that $P0$, $P1$ and $P2$ were already calculated during approximation of the arc length (as described in chapter 4.4.2.2). For the transformation to world space the actual elevation by the noise function can be safely ignored because the minimum and maximum points will be extended by the maximum elevation possible.

4.4.4.2 Testing the AABBs against the View Frustum

A view frustum is a truncated pyramid that is defined by six planes as shown in Figure 4-21. The normal of each plane faces the interior of this particular pyramid.

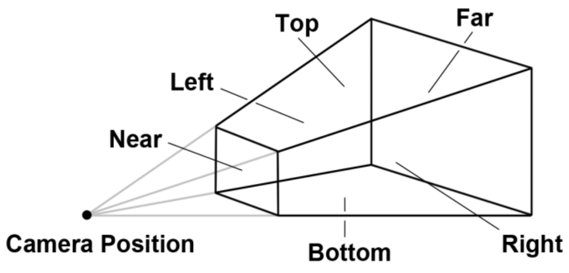

Figure 4-21: View frustum and the corresponding planes

In the proposed model, frustum culling is done by testing the orientation of objects against these planes. The orientation of a plane and a position in 3D can be easily determined by the plane equation shown in Equation 4-15.

$$ax + by + cz + d = 0 \qquad \text{Equation 4-15}$$

Where a, b and c define the plane normal N, x, y, z define the coordinates of a point P on the plane and d equals $-N.P$. Using this equation, the orientation of a point to the plane can be easily tested by replacing the x, y and z terms with the corresponding values of this particular point. If the point lies in front of the plane, which means the normal of the plane faces towards the point, the equation will result into a value greater than 0 and the point has a positive distance to the plane. A resulting value, which is smaller than 0 indicates a point that lies behind the plane and thus has a negative distance.

This means that for culling purposes, a single point has to be tested against all six planes. If all equations result in values greater or equal to 0 the point will lie inside the frustum. If one equation results in a negative distance, the point will be outside. Thus, a simple approach to do frustum culling with AABBs would be to perform this test with all eight points that define the bounding box. If all points of a single AABB result in a negative distance for a single plane, the object inside the AABB will not be seen and can be culled safely. However, this can be optimized in a way that allows doing frustum culling by testing only a single point of the AABB. If it is desirable to differentiate between objects that lie partially, and objects that lie entirely within the frustum, two points will need to be tested [34].

These two points are called the *p-vertex* and the *n-vertex* [35]. The p-vertex defines the point of the AABB that has the greatest positive distance from the plane and the n-vertex defines the point that has the greatest negative distance from the plane. Figure 4-22 shows some examples of p- and n-vertices with respect to a particular plane.

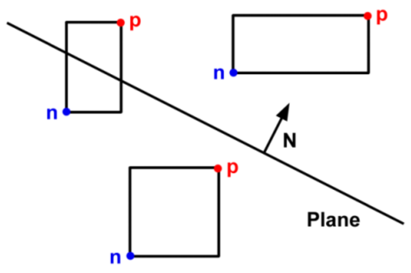

Figure 4-22: p-vertices, n-vertices and their orientation to a plane

By testing these two points the orientation of the AABB with respect to the plane and its normal can be determined. If the p-vertex has a negative distance, the AABB will lie completely in the negative half-space of the plane. If it is greater than 0 the AABB will partially intersect the frustum (if the n-vertex has negative distance) or will lie completely within the positive half-space (if n-vertex has a positive distance).

For frustum culling it is often enough to know if a certain objects lies completely outside the frustum so it can be discarded. In this case it is not even necessary to test the n-vertex. This means frustum culling can be done by testing only a single point of the AABB.

Listing 4-10 shows the calculation of the p-vertex and the test that determines if the AABB lies outside the view frustum. Note that solving Equation 4-15 can be done by a simple dot product of four dimensional vectors.

```
// tests frustum for intersection with an axis aligned bounding box
// returns true if AABB does not intersect frustum
bool Frustum::CullAABB(const D3DXVECTOR3& minAABB, const D3DXVECTOR3&
maxAABB) const
{
        // test if box is completly outside frustum
        for (unsigned int i=0; i<NumPlanes; ++i)
        {
                D3DXVECTOR4 pVertex = D3DXVECTOR4(minAABB, 1.0f);
                if (m_Planes[i].a >= 0)
                        pVertex.x = maxAABB.x;
                if (m_Planes[i].b >=0)
                        pVertex.y = maxAABB.y;
                if (m_Planes[i].c >= 0)
                        pVertex.z = maxAABB.z;
                if (D3DXPlaneDot(&m_Planes[i], &pVertex) < 0)
                        return true;
        }
        // box is completly inside or intersects frustum
        return false;
}
```

Listing 4-10: Testing frustum against AABB for intersection

When a block's AABB gets culled, the recursive divide can be safely stopped, as the child blocks would not be visible either.

Figure 4-23 shows the results after culling is performed. As can be seen, only a fraction of the total planetary surface is rendered and geometry which does not intersect the view frustum is culled (grey surfaces).

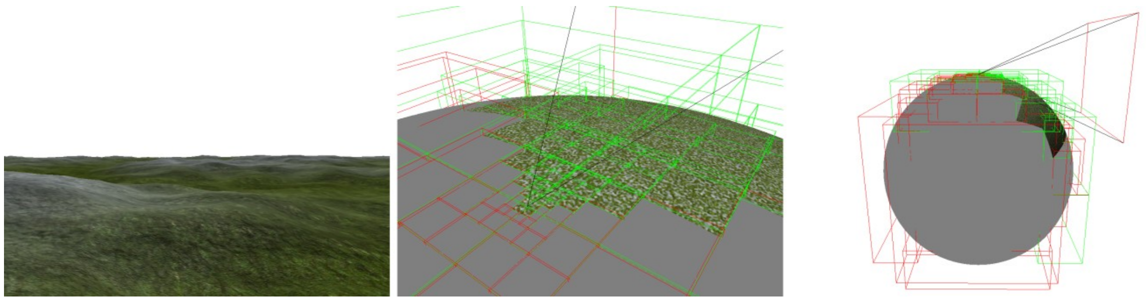
Figure 4-23: Frustum culling (left) camera's point of view (middle and right) rendered geometry after culling (red AABB: no intersection, green AABB: intersecting view frustum)

4.4.4.3 Resulting Performance Gain

The performance gain due to frustum culling is remarkable. It depends primarily on the value of the constant C that controls the LOD of the resulting terrain, the current altitude and the view direction of the camera. To demonstrate the importance of frustum culling the required milliseconds to render a single frame are compared during three different scenes (Figure 4-24), with varying values for the LOD constant C.

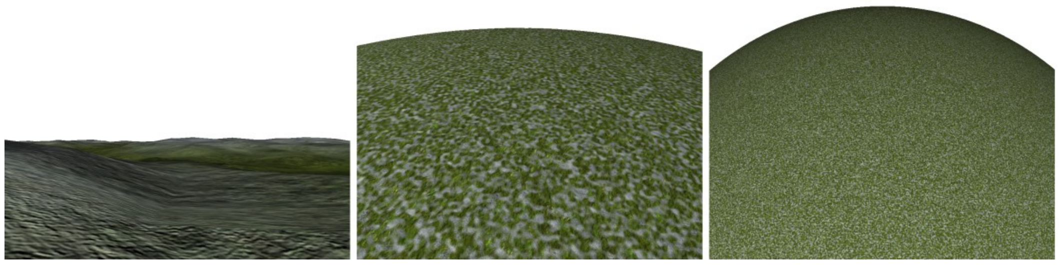
Figure 4-24: Scenes used for performance comparison of enabling frustum culling (left) scene A (middle) scene B (right) scene C

Figure 4-25 shows that frustum culling is important for scenes near the ground, when only a small fraction of the total planetary surface contributes to the final image. For scene A rendering took up to seven times longer with frustum culling disabled when using a high LOD constant.

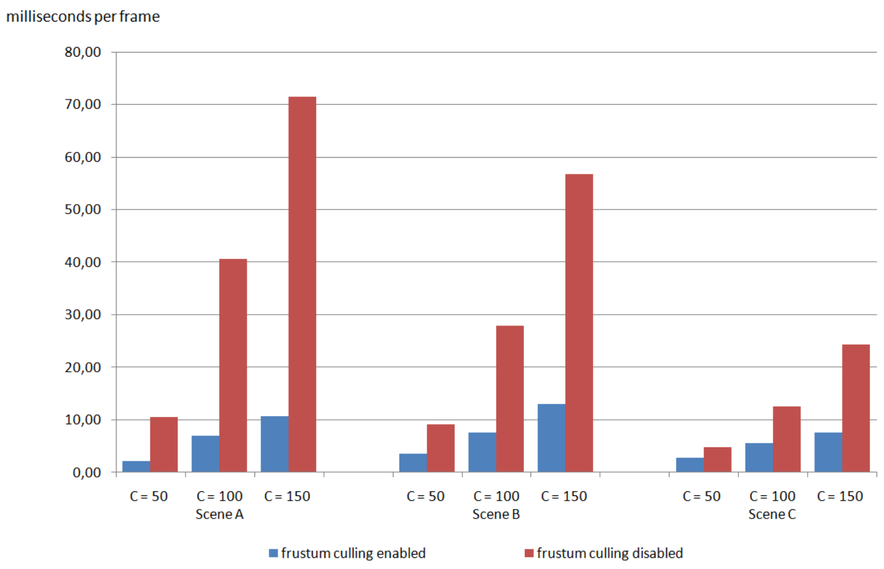

Figure 4-25: Comparing milliseconds per frame with and without frustum culling enabled $(blocksize: 33, max\ recursion: 9)$

Note that the relative performance gain will decrease, if much of the geometry lies within the frustum (for example in scene C). In this case the computational overhead increases as more AABBs have to be calculated and tested for intersection, while only few of them can be actually culled. But even then, the resulting gain in rendering speed is more than worth it.

4.5 Rendering of the Sun

This chapter presents a simple method to render the sun as a single textured quadrilateral. The proposed method should be understood as a technique to enhance plausibility of the resulting scene and is by no means a physically correct simulation of the real sun.

To achieve a convincing representation of the sun, the quadrilateral faces the camera at all times. Allowing a free navigation of the camera and a sun rotating around the planet thus requires generating a proper transformation matrix at each frame. This chapter shows how this matrix is generated by a translation and a rotation matrix. After this, it is shown how direct sunlight can be expressed by the equations of reflected light. This simplifies rendering atmospheric scattering as a post-processing effect.

The full source code of the sun shader is provided in Appendix B.

4.5.1 Generating the Translation Matrix

To simulate a great distance, the sun is always positioned relative to the camera position close to the far plane. This means, the distance from the camera to the sun is constant at all times. The sun position in world space is calculated as shown in Listing 4-11.

```
// calculates position of sun relative to camera position
float3 sunPosition = g_cameraPos + (g_sunDirection * g_sunDistance);
```

Listing 4-11: Determining the position of the sun relative to the camera

The sun position is then used to generate a translation matrix, which translates the vertices of the quadrilateral accordingly as shown in Listing 4-12.

```
void generateTranslationMatrix(in float3 sunPosition, out float4x4
translationMatrix)
{
	translationMatrix[0] = float4(1.0f, 0.0f, 0.0f, 0.0f);
	translationMatrix[1] = float4(0.0f, 1.0f, 0.0f, 0.0f);
	translationMatrix[2] = float4(0.0f, 0.0f, 1.0f, 0.0f);
	translationMatrix[3] = float4(sunPosition, 1.0f);
}
```

Listing 4-12: Generating the translation matrix of the sun

4.5.2 Generating the Rotation Matrix

Forcing the quadrilateral to face the camera at any time requires setting up a proper rotation matrix. The three basis vectors of this particular rotation matrix are found as follows:

- The first basis vector describes the direction from the sun to the camera and equals the negative sun direction vector.

- The second basis vector is found by normalizing the cross product of the first basis vector with a cardinal basis vector. The appropriate cardinal basis vector is found by examining the x,y and z component of the first basis vector and picking the one where the corresponding component has the least magnitude.

- The third basis vector is simply the normalized cross product of the first and the second basis vector.

Listing 4-13 shows the building of the rotation matrix.

```
void generateRotationMatrix(out float4x4 rotationMatrix)
{
	float3 lookAtDirection = normalize(-g_sunDirection);
	float3 baseVector0, baseVector1, baseVector2;

	// first base vector equals lookAtDirection
	baseVector0 = lookAtDirection;

	// second base vector found by crossing first base vecotr with
	// cardinal basis vector corresponding to element with least magnitude
	float3 crossVector = float3(1, 0, 0);
	float absX = abs(lookAtDirection.x);
	float absY = abs(lookAtDirection.y);
	float absZ = abs(lookAtDirection.z);

	if((absY <= absX) && (absY <= absZ))
		crossVector = float3(0.0f, 1.0f, 0.0f);
	else if((absZ <= absX) && (absZ <= absY))
		crossVector = float3(0.0f, 0.0f, 1.0f);

	baseVector1 = normalize(cross(baseVector0, crossVector));

	// third base vector equals crossing first and second base vector
	baseVector2 = normalize(cross(baseVector0, baseVector1));

	rotationMatrix[0] = float4(baseVector2, 0.0f);
	rotationMatrix[1] = float4(baseVector1, 0.0f);
	rotationMatrix[2] = float4(baseVector0, 0.0f);
	rotationMatrix[3] = float4(0.0f, 0.0f, 0.0f, 1.0f);
}
```

Listing 4-13: Generating the rotation matrix of the sun

Consider that using this technique can result in orientation discontinuities and a rolling of the quadrilateral depending on the direction vector. This however is not noticeable as long as the applied texture is radially symmetric.

4.5.3 Expressing Direct Sunlight as Reflected Light

As described by Equation 2-8, scattering of light is a summation of direct sunlight, in-scattered light and reflected light. The terrain rendered in the previous chapter will contribute to the final image as part of the reflected light. In order to keep the rendering process as simple and as efficient as possible, direct sunlight is expressed as reflected light and will thus contribute to the final image implicitly.

Transforming direct sunlight into reflected light is done by assuming an irradiance value equal to the emittance of the sun. For this the normal vector of the quadrilateral is set equally to the sun direction vector. Additionally, the term $\frac{\alpha(x_0)}{\pi}$ of Equation 2-13 has to be set to 1. This renders the quadrilateral to reflect the full intensity of the sun without any loss and thus allows simulation of direct sunlight.

Transforming direct sunlight into reflected light allows rendering into the GBuffer and simplifies the rendering process, as the entire contents of the Buffer can be treated equally.

Listing 4-14 shows how the pixel shader is used to transform direct sunlight into reflected light.

```
// sample texture with alpha channel
output.color = g_texture.Sample( LinearSamplerClamp, input.texC );

// set normal vector to sun direction vector
float3 normal = g_sunDirection;
output.normal.xyz = 0.5f * normal + 0.5f;

// set sun reflectance value to 1.0f
output.normal.w = 1.0f;
```

Listing 4-14: Transforming direct sunlight into reflected light

4.6 Pre-Computed Atmospheric Scattering

This chapter shows how atmospheric scattering can be applied to the planetary terrain as a post-processing effect. The described approach is based on the work of Bruneton and Neyret [1] and allows rendering of accurate atmospheric scattering effects in real-time.

Interactive frame rates are preserved by performing a separate pre-processing step that solves the computational intensive scattering equations. The corresponding results are then stored in tables, which are accessed during run-time

The pre-computation process generates three different tables storing the transmittance, the irradiance and the in-scattered light for a perfect spherical ground. For this, single and multiple scattering are taken into account and computed exactly. The accuracy of the resulting tables is only limited by their resolution and their storage format.

In the following chapters a detailed overview of the whole pre-computation process is given. It is then shown how the resulting tables can be used to apply atmospheric scattering as a single post-processing effect.

4.6.1 Pre-Computation

The pre-computation step generates three different look-up tables which are then used during run-time. The tables allow retrieving the extinction factor and the in-scattered light for a given path as well as obtaining the total irradiance at a certain position within the atmosphere.

The pre-computation process is performed entirely on the GPU. For this a set of shaders and intermediate textures are required. A full overview is given in Figure 4-26. As shown, pre-computation can be split up in two main tasks: Computing single scattering and computing multiple scattering. The latter is an iterative process, where the number of iterations determines the accuracy of the resulting tables.

During pre-computation the planet is treated as a perfect spherical ground. This allows storing the results of the scattering equations into tables of reasonable size by following an idea originally proposed by O’Neil [36]. Assuming the planet as a perfect spherical ground allows describing a position within the atmosphere only by its altitude (the distance to the center of the planet) and a direction by its angle to the zenith vector. However, this parameterization implies that the composition of the atmosphere does not vary with changes in latitude or longitude.

Not considering the actual terrain during pre-computation results in minimal errors during multiple scattering. However, these errors are insignificant as contribution of multiple scattering is subtle.

In the next chapters, an overview of the look-up tables is given and the pre-computation process is described in further detail. The presented code samples are based on the published source code [37] of Brunetons and Neyrets approach [1].

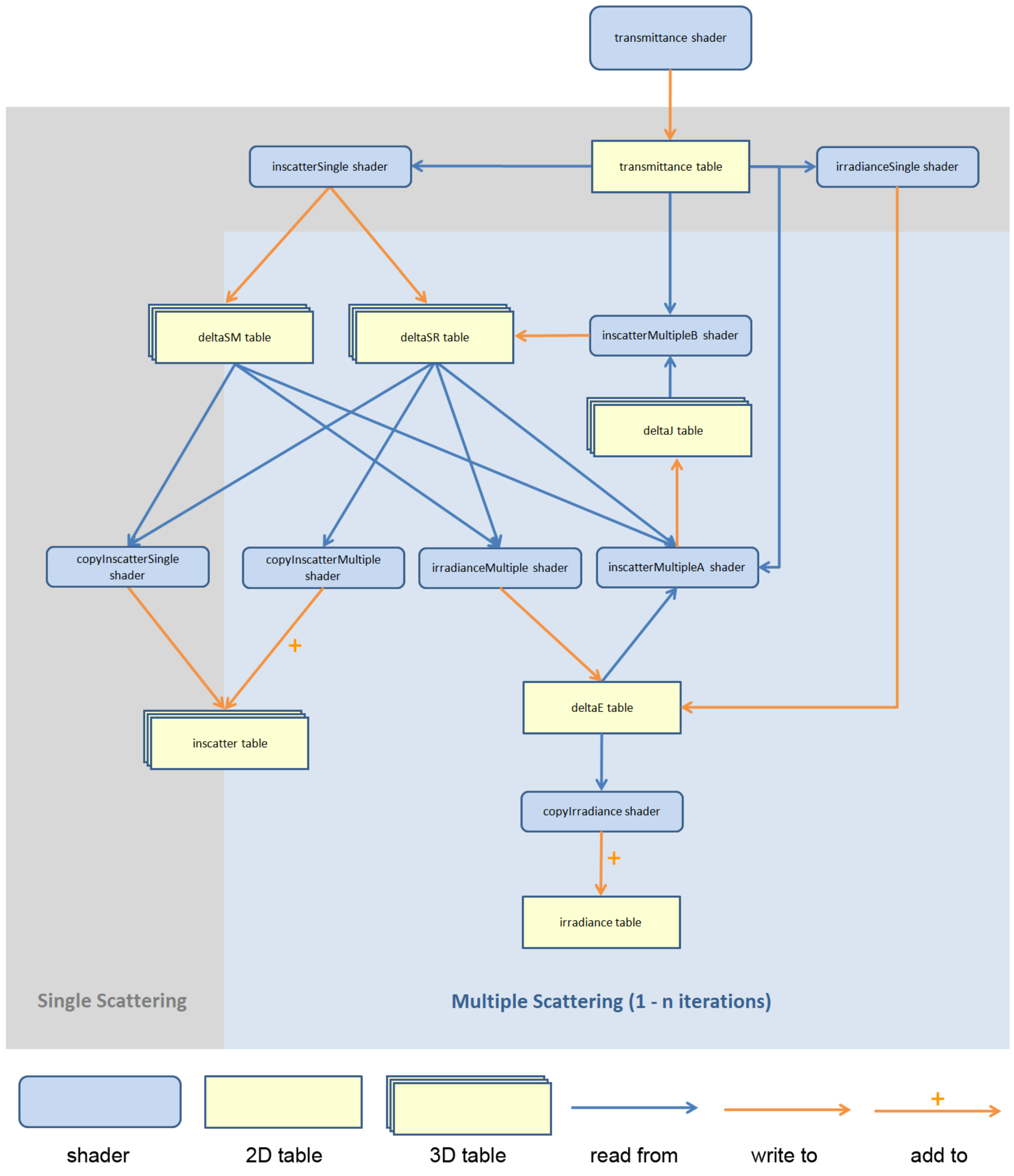

Figure 4-26: Overview of the pre-computation process

4.6.1.1 Lookup Tables

As described, the pre-computation step generates three tables: the *transmittance* table, the *irradiance* table and the *inscatter* table. These are basically 16 bit floating point textures that act as look-up tables to retrieve the extinction factor, the in-scattered light and the irradiance value.

Transmittance table

The *transmittance* table is used to obtain the extinction factor for a given path. This path is parameterized by an altitude where the ray starts and the cosine angle between the zenith and the corresponding direction vector. The structure of the table is shown Figure 4-27.

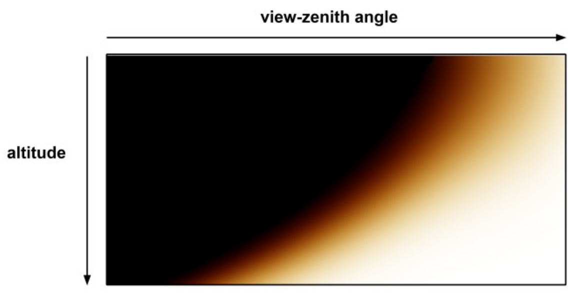

Figure 4-27: Structure of the $transmittance$ table

The table stores the extinction factor for infinite paths, which end by either hitting the top of the atmosphere at an altitude of R_t or the planetary ground at an altitude of R_g.

However, by fetching two values it is possible to calculate the correct extinction factors along arbitrary paths within the atmosphere. Recall that the extinction factor can be understood as the fraction or percentage of an incident light that remains after traversing the atmospheric medium over a given path. Hence, the attenuation value for each path shown in Figure 4-28 can be obtained by rewriting Equation 4-16.

$$F_{ex}(t_{a\to c}) = F_{ex}(t_{a\to b}) * F_{ex}(t_{b\to c})$$ Equation 4-16

For example, the transmittance value for path $a \to b$ would be $\frac{F_{ex}(t_{a\to c})}{F_{ex}(t_{b\to c})}$. Both values can be retrieved from the table.

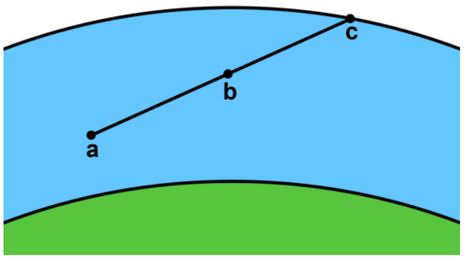

Figure 4-28: Extinction factor for path $a \rightarrow c$ equals the product of the extinction factor for path $a \rightarrow b$ with the extinction factor for path $b \rightarrow c$

In order to enhance accuracy, the extinction factors are just pre-calculated for angles that lie between 0 and $\frac{\pi}{2} + \varepsilon$ (the angle to the horizon being $\sim\frac{\pi}{2}$). In this case ε describes a small value to take the curvature of the planetary surface into account. This means angles greater than the angle to the horizon cannot be retrieved from the table directly and thus need special handling.

Figure 4-29 shows how angles greater than the angle to the horizon need to be handled. In the depicted example the extinction factor for the path $\mathrm{a} \rightarrow \mathrm{b}$ needs to be calculated. The angles α and β represent the viewing angles relative to the zenith vector of a and b respectively. As shown, the angle α is clearly greater than the angle to the horizon. Due to spherical symmetry it is valid to use the angles of α' and β' instead. These angles are smaller than $\frac{\pi}{2}$ and can be used without any problems. In this case the extinction factor for the reverse path $b \rightarrow a$ is calculated (which is equal to the extinction factor for the path $a \rightarrow b$).

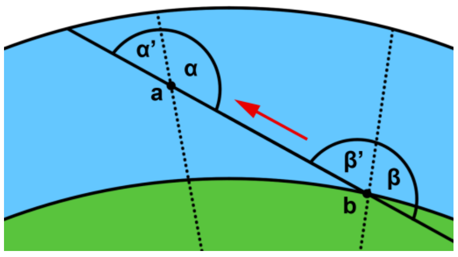

Figure 4-29: The transmittance for paths whose angles are above the angle to the horizon ($a \rightarrow b$) are calculated assuming the reversed path ($b \rightarrow a$)

Note that instead of using the angle between the view and the zenith vector directly, the cosine of this angle is used, when accessing the $transmittance$ table.

In-scatter table

The *inscatter* table stores the total in-scattered light along an infinite path through the atmosphere. A value in this table is parameterized by four values: An altitude and the cosine of the view-zenith, the sun-zenith and the view-sun angle. As handling 4D tables is not supported by ordinary GPUs, a 3D texture is utilized to simulate a 4D table. For this, the dimension of the sun-zenith angle is subdivided into additional segments to allow storage of the view-sun angle. The structure of the $inscatter$ table is shown in Figure 4-30.

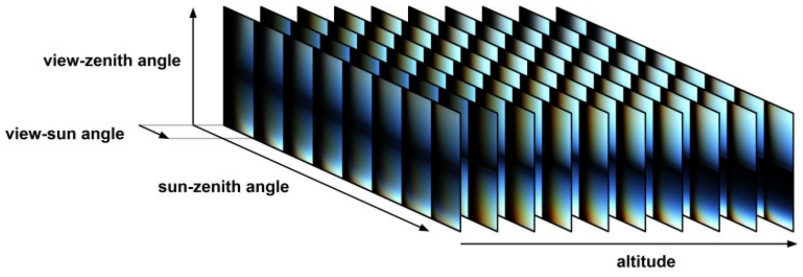

Figure 4-30: Structure of the $inscatter$ table

3D textures will grow in size rather quickly if their resolution is increased. This limits the angular resolution of the stored angles. However, according to Bruneton and Neyret [1] this is only problematic at single Mie scattering because the corresponding phase function has a high angular dependency (as described in chapter 2.3). Thus, Bruneton and Neyret propose to separate the single Mie scattering term from all the others. The corresponding phase function is then applied at run-time.

Storing single Mie scattering separately would require storing three additional values (one for each of the red, blue and green channel). This means, the memory requirements of the $inscatter$ table would be doubled as Rayleigh scattering also occupies three channels.

Fortunately the memory consumption can be reduced considerably by a proportion rule [1]. This proportion rule allows approximating the whole single scattered Mie term as shown by Equation 4-17.

$$C_M \cong \frac{C_M(red)}{C*(red)} \frac{\beta_R^S(red)}{\beta_M^S(red)} \frac{\beta_M^S}{\beta_R^S} \qquad \text{Equation 4-17}$$

$$C* = L_{in}[L_0](Rayleigh) + \frac{L_{in}[L_x]}{Ph_R}$$

C_M describes the full approximated Mie term and $C*$ equals the sum of single Rayleigh scattering and the full multiple scattering term divided by the Rayleigh phase function.

Therefore, the $inscatter$ table stores in the red, blue and green channel the term $C*$ and in the alpha channel $C_M(red)$. This allows approximating the value C_M during run-time. Note that after retrieving these values, the Rayleigh and Mie phase functions need to be reintroduced.

Similar to the $transmittance$ table, the stored in-scattered light is only valid for an infinite path. This means special care has to be taken here as well. Consider again the paths depicted in Figure 4-28. The in-scattered light along the path $a \rightarrow c$ can be described as shown in Equation 4-18.

$$L_{in}(a \rightarrow c) = L_{in}(a \rightarrow b) + F_{ex}(t_{a \rightarrow b})L_{in}(b \rightarrow c)$$ Equation 4-18

This means the in-scattered light for path $a \rightarrow b$ can be obtained by solving $L_{in}(a \rightarrow c) - F_{ex}(t_{a \rightarrow b})L_{in}(b \rightarrow c)$ where both terms can be retrieved from the table.

Irradiance table

The $irradiance$ table stores the incident light on a surface due to multiple scattering. A value in this table is accessed by providing an altitude and the cosine of the sun-zenith angle as shown in Figure 4-31.

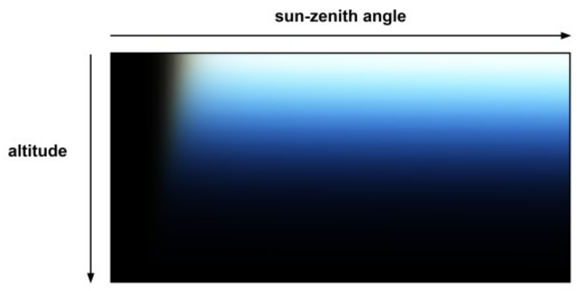

Figure 4-31: Structure of the $irradiance$ table

Recall that irradiance is calculated by an integral over the hemisphere (as described in Equation 2-12). However, when calculating single scattering, the equation can be greatly simplified. This is because during single scattering only one light source is available (which is the sun). This is not the case during multiple scattering as light can be incident from every direction due to the scattering of light.

Considering only one light source allows reducing Equation 2-12 to a single dot product of the surface normal and the sun direction. Because of the low computational costs and the fact that the table offers only limited angular precision, incident light due to single scattering is calculated in real-time.

4.6.1.2 Calculating Transmittance

Building up the $transmittance$ table has to be done first. For this the *transmittance* shader is executed. To calculate the extinction factor that is stored in the table, Equation 2-6 needs to be solved. This means the density ratio needs to be integrated over the given path as shown in Listing 4-15.

```
float densityOverPath(in float scaleHeight, in float alt, in float mu)
{
  // if ray below horizon return max density
  float cosHorizon = -sqrt(1.0f - ((g_Rg*g_Rg)/(alt*alt)));
  if(mu < cosHorizon)
    return 1e9;

  float totalDensity = 0.0f;
  float dx = itersectAtmosphere(alt,mu) /
             float(TRANSMITTANCE_INTEGRAL_SAMPLES);

  float y_j = exp(-(alt-g_Rg)/scaleHeight);

  for (int i = 1; i<=TRANSMITTANCE_INTEGRAL_SAMPLES; ++i)
  {
    float x_i = float(i)*dx;
    float alt_i = sqrt(alt*alt + x_i*x_i + 2.0f*x_i*alt*mu);
    float y_i = exp(-(alt_i-g_Rg)/scaleHeight);
    totalDensity += (y_j+y_i)/2.0f*dx;
    y_j = y_i;
  }

  return totalDensity;
}
```

Listing 4-15: Calculating the optical density over a path given by altitude and view-zenith angle

$scaleheight$ stands for H_R and H_M respectively, alt for the altitude that describes the current entry in the table and mu denotes the corresponding cosine of the view-zenith angle.

In Listing 4-15 the density ratio ρ of each integration sample is summed up and returned as the total density over the path. For this the path traversed within the atmosphere needs to be known.

The length of this path is found by a ray/sphere intersection test within the function $intersectAtmosphere$. This path starts at the given altitude and ends when either hitting the surface of the planet at an altitude of R_g or the top boundary of the atmosphere at an altitude of R_t.

After obtaining the corresponding length, the path is sampled and the density ratio ρ (given by Equation 2-5) for each sample point is calculated. Calculating ρ requires the altitude alt_i of each sample point. alt_i is obtained by using the law of cosine. As shown in Figure 4-32 (left) alt, alt_i and dx build a triangle. Together with the cosine angle $\cos(\alpha) = mu$ it is possible to obtain alt_i and hence the value used for h in Equation 2-5.

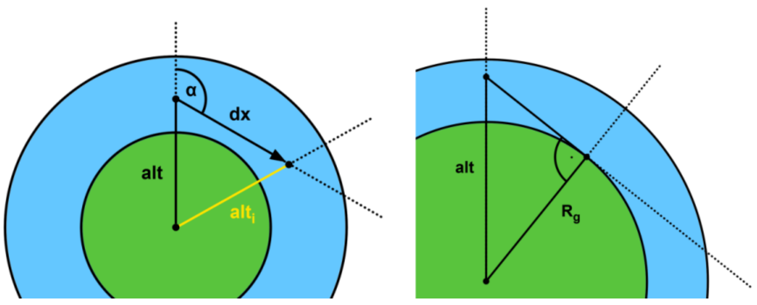

Figure 4-32: (left) Calculating the altitude alt_i (right) R_g, alt and the horizon vector build a right triangle

The total density is then calculated by integrating the value of ρ of each sample point over the whole path by using the trapezoidal rule.

Note if the view-zenith angle approaches $\frac{\pi}{2}$ the density will converge to infinity. The initial check prevents calculations with arbitrary high values and simply returns a maximum density ratio for angles below the horizon. As shown in Figure 4-32 (right) R_g, the current altitude and the horizon vector build a right triangle. Thus, the angle from the zenith to the horizon can be calculated by applying the Pythagorean theorem.

As the scale height varies for aerosols and air molecules, the density ratio for the given path has to be calculated twice. The resulting densities are then multiplied by the extinction coefficients β_R^e and β_M^e for Rayleigh and Mie scattering respectively to obtain the optical depth as described by Equation 2-6. After these calculations the extinction factor is determined and stored in the $transmittance$ table as shown in Listing 4-16.

```
// calculates extinction factor of given altitude and view direction
float3 t = betaR * densityOverPath(HR, alt, mu) +
           betaM * densityOverPath(HM, alt, mu);

return float4(exp(-t), 0.0f);
```

Listing 4-16: Calculating the extinction factor

4.6.1.3 Calculating Single Scattering

The following chapters show how single scattering is calculated in the pre-computation process.

Calculating Irradiance

The irradiance caused by single Rayleigh and Mie scattering at each point within the atmosphere is calculated by the *irradianceSingle* shader.

As described in chapter 4.6.1.1, the irradiance value due to single scattering can be reduced to a single dot product as shown in Listing 4-17.

```
float3 attenuation = transmittance(alt, mus);
return float4(attenuation * saturate(mus), 0.0f);
```

Listing 4-17: Calculating irradiance due to single scattering

mus describes the cosine of the sun-zenith angle. As the planet is treated as a perfect spherical ground, the zenith vector equals the surface normal. Therefore mus already represents the required dot product and only needs proper clamping.

As the light traverses the atmosphere before hitting the surface point, it has to be attenuated accordingly. This means the dot product needs to be multiplied by the appropriate extinction factor, which is retrieved from the $transmittance$ table. The resulting value is stored in an intermediate table $deltaE$, which is parameterized like the $irradiance$ table.

Calculating In-scattered Light

Single Rayleigh and Mie scattering is calculated by the $inscatterSingle$ shader.

Recall that in-scattering describes the total light that is scattered into a viewing path. This means that the light was initially headed in a different direction but a fraction of it is redirected into the viewing path as a result of Rayleigh or Mie scattering. Before reaching the scattering point, the light has already traversed the atmosphere to some extent and needs proper attenuation. It is then attenuated a second time from the scattering point to the origin of the view ray. This is described by Equation 2-11. However, as with calculating irradiance, the given formula can be heavily simplified by the fact that only single scattering is considered. In this case the integral of J does not need to be solved as the sun is the only light source that needs to be considered (there is no need to integrate over the total solid angle 4π).

To calculate the in-scattered light, the viewing path has to be sampled. At each sample point the incident light has to be determined. Listing 4-18 shows how this can be accomplished.

```
void integrand(in float alt, in float mu, in float mus, in float nu, in float
dist, out float3 ray, out float3 mie)
{
  ray = float3(0.0f, 0.0f, 0.0f);
  mie = float3(0.0f, 0.0f, 0.0f);

  float alt_i  = sqrt(alt*alt + dist*dist + 2.0*alt*mu*dist);
  float mus_i = (nu*dist + mus*alt)/alt_i;

  alt_i = max(g_Rg, alt_i);

  // if angle between zenith and sun smaller than angle to horizon
  // return ray and mie values
  if (mus_i >= -sqrt(1.0f - ((g_Rg*g_Rg)/(alt_i*alt_i))))
  {
    float3 trans = transmittance(alt, mu, dist)*transmittance(alt_i, mus_i);
    ray = exp(-(alt_i - g_Rg) / HR) * trans;
    mie = exp(-(alt_i - g_Rg) / HM) * trans;
  }
}
```

Listing 4-18: Obtaining light arriving at each sample point during single in-scattering

nu describes the cosine angle between the view and the sun direction and $dist$ the distance of the sample point from the origin of the ray.

First the height of each sample point along with the appropriate sun zenith angle is calculated. In case that the angle to the sun is smaller than the angle to the horizon (which means that the earth isn't casting a shadow onto this sample point) the fraction of light that reaches this particular position is calculated.

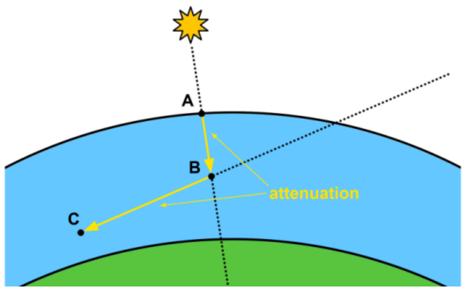

Figure 4-33: Single in-scattered light is attenuated two times

Figure 4-33 depicts single in-scattered light at position C. In the example, the sun ray enters the atmosphere at position A. The ray then traverses the atmosphere between position A and B before it gets scattered towards point C. The light is attenuated two times: At first on the path from $A \rightarrow B$ then on the path $B \rightarrow C$. The attenuation value of both paths can be calculated as described in chapter 4.6.1.1. The total extinction factor is then the product of these. To obtain the in-scattered light at this particular sample position, the total extinction factor is multiplied with the appropriate density as described by Equation 2-10. Recall that the phase function is applied at runtime so the term Ph_i of Equation 2-10 is left out.

The resulting value represents the in-scattered light at a single position on the viewing path. This means that the path has to be sampled several times to obtain the total in-scattered light. For this Equation 2-11 needs to be solved, as shown in Listing 4-19.

```
void inscatter(float alt, float mu, float mus, float nu, out float3 ray, out
float3 mie)
{
  ray = float3(0.0f, 0.0f, 0.0f);
  mie = float3(0.0f, 0.0f, 0.0f);
  float dx = intersectAtmosphere(alt, mu)/float(INSCATTER_INTEGRAL_SAMPLES);
  float x_i = 0.0;
  float3 ray_i;
  float3 mie_i;
  integrand(alt, mu, mus, nu, 0.0f, ray_i, mie_i);
  for (int i = 1; i <= INSCATTER_INTEGRAL_SAMPLES; ++i)
  {
    float x_j = float(i) * dx;
    float3 ray_j;
    float3 mie_j;
    integrand(alt, mu, mus, nu, x_j, ray_j, mie_j);
    ray += (ray_i + ray_j) / 2.0f * dx;
    mie += (mie_i + mie_j) / 2.0f * dx;
    x_i = x_j;
    ray_i = ray_j;
    mie_i = mie_j;
  }

  ray *= betaR;
  mie *= betaMSca;
}

PS_OUTPUT PS_InScatterSingle(PS_InLayer input)
{
  ...
  inscatter(g_r, mu, mus, nu, ray, mie);

  output.ray = float4(ray, 1.0f);
  output.mie = float4(mie, 1.0f);

  return output;
}
```

Listing 4-19: Calculating in-scattered light due to single scattering

After solving the integral, the scattering coefficients are brought into the calculation and the results are stored in the intermediate tables $deltaSR$ and $deltaSM$ for Rayleigh and Mie scattering respectively. These tables are structured similar to the $inscatter$ table.

Transferring Single In-scattered Light to Final Scattering Texture

The intermediate results of in-scattering are transferred to the final $inscatter$ table by the *copyInscatterSingle* shader. As described in chapter 4.6.1.1 the red channel of single Mie scattering is stored separately in the alpha channel as shown in Listing 4-20.

```
float4 PS_CopyInscatterSingle(PS_InLayer input) : SV_TARGET0
{
  ...
  float4 ray = g_texDeltaSR.SampleLevel(LinearSamplerClamp,
               float3(input.texC, layer), 0);

  float4 mie = g_texDeltaSM.SampleLevel(LinearSamplerClamp,
               float3(input.texC, layer), 0);

  // store only red component of single Mie scattering
  return float4(ray.rgb, mie.r);
}
```

Listing 4-20: Copying single in-scattered light into final $inscatter$ texture

This concludes the pre-calculations of single scattered light.

4.6.1.4 Calculating Multiple Scattering

Recall that calculating multiple scattering is an iterative process. As the contribution of multiple scattered light is subtler than it is the case with single scattering, usually three iterations are enough to allow achieving very accurate scattering effects in real-time.

Pre-computing multiple scattering is more complex than single scattering. Irradiance is calculated in a single step by examining the total incident light over the hemisphere. In contrast to this, calculating in-scattered light is a two step process. In the first step the total light scattered into each position is determined. In the second step this information is used to get the total in-scattered light over a whole path. The results of irradiance and in-scattering are then stored in intermediate tables, which are reused as input parameters for the next iteration. In addition, after each iteration the intermediate results are added to the final $irradiance$ and $inscatter$ tables.

Calculating Total In-scatterred Light at each Position

Calculating multiple in-scattering requires solving the integral of Equation 2-10 as light may be incident from different directions (as a result of previous scattering iterations).

The *inscatterMultipleA* shader solves this integral over the total solid angle 4π with two nested loops. For this, the integral over solid angles is transferred into an integral over spherical coordinates. This means that a set of direction vectors w on the surface of a unit sphere are generated. These vectors are described by two angles: θ (ranging from 0 to π) and φ (ranging from 0 to 2π) as shown in Figure 4-34.

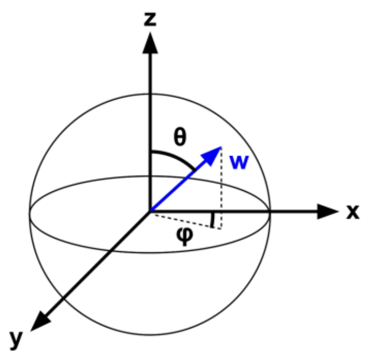

Figure 4-34: A direction vector w defined in spherical coordinates by two angles θ and φ

Listing 4-21 shows how this integration over spherical coordinates is done.

```
static const float dphi = M_PI/float(INSCATTER_SPHERICAL_INTEGRAL_SAMPLES);
static const float dtheta = M_PI/float(INSCATTER_SPHERICAL_INTEGRAL_SAMPLES);

void inscatter(float alt, float mu, float mus, float nu, out float3 raymie)
{
  alt = clamp(alt, g_Rg, g_Rt);
  mu = clamp(mu, -1.0, 1.0);
  mus = clamp(mus, -1.0, 1.0);

  float cosHorizon = -sqrt(1.0 - ((g_Rg*g_Rg)/(alt*alt)));

  float3 v,s;
  getViewAndSunVector(mu, mus, nu, v, s);

  raymie = float3(0.0f, 0.0f, 0.0f);

  // integral over 4.PI around current position with two nested loops
  // over w directions (theta,phi)
  for (int itheta = 0; itheta < INSCATTER_SPHERICAL_INTEGRAL_SAMPLES;
       ++itheta)
  {
    float theta = (float(itheta) + 0.5) * dtheta;
    float ctheta = cos(theta);

    float greflectance = 0.0f;
    float dground = 0.0;
    float3 gtransp = float3(0.0f, 0.0f, 0.0f);

    // check if ground visible
    if (ctheta < cosHorizon)
    {
      greflectance = AVERAGE_GROUND_REFLECTANCE / M_PI;
      // ground is visible - calculate distance and transparency to
      // surface point
      float hx = alt * sqrt(1.0f - ctheta * ctheta);
      dground = -alt*ctheta - sqrt(alt*alt* (ctheta*ctheta-1.0f)+g_Rg*g_Rg);
      gtransp = transmittance(alt, ctheta, dground);
    }

    // Inner loop described in Listing 4-22
    ...
  }
}
```

Listing 4-21: Calculating multiple in-scattered light at each position (Outter loop)

As shown in Listing 4-21 the zenith-view, the zenith-sun and the sun-view angle can be used to obtain a view and a sun vector.

First it is checked if the planetary surface is visible in direction w. This is the case when the cosine of θ is smaller than the cosine of the angle to the horizon. In this case reflected light is scattered into the current position and the distance to the surface, the reflection factor and the extinction factor are calculated.

Listing 4-22 shows the inner loop and the actual calculation of the total in-scattered light at the current position.

```
for (int iphi = 0; iphi < 2 * INSCATTER_SPHERICAL_INTEGRAL_SAMPLES; ++iphi)
{
  float phi = (float(iphi) + 0.5) * dphi;
  float3 w = float3(cos(phi) * sin(theta), sin(phi) * sin(theta), ctheta);

  float nu1 = dot(s, w);
  float nu2 = dot(v, w);
  float pr2 = phaseFunctionR(nu2);
  float pm2 = phaseFunctionM(nu2);

  // compute irradiance incident to surface point in direction w
  float3 gnormal = (float3(0.0, 0.0, alt) + dground * w) / g_Rg;
  float3 girradiance = irradiance(g_texDeltaE, g_Rg, dot(gnormal, s));

  // light incident to current position from direction w
  float3 raymie1;

  // first term = light reflected from the ground and attenuated before
  // reaching current position
  raymie1 = greflectance * girradiance * gtransp;

  // second term = in-scattered light
  if (g_first == 1.0)
  {
    // first iteration => introduce phase functions
    float pr1 = phaseFunctionR(nu1);
    float pm1 = phaseFunctionM(nu1);
    float3 ray1 = texture4D(g_texDeltaSR, alt, w.z, mus, nu1).rgb;
    float3 mie1 = texture4D(g_texDeltaSM, alt, w.z, mus, nu1).rgb;
    raymie1 += ray1 * pr1 + mie1 * pm1;
  }
  else
  {
    raymie1 += texture4D(g_texDeltaSR, alt, w.z, mus, nu1).rgb;
  }

  // light coming from direction w and scattered in direction v
  float dw = dtheta * dphi * sin(theta);
  raymie += raymie1 * (betaR * exp(-(alt - g_Rg) / HR) * pr2 + betaMSca *
            exp(-(alt - g_Rg) / HM) * pm2) * dw;
}
```

Listing 4-22: Calculating multiple in-scattered light at each position (Inner loop)

Within the nested loop, the direction vector w is calculated using the angles θ and φ and the appropriate transformation rule from spherical coordinates to Cartesian coordinates. As described by Equation 2-17, multiple scattering is a summation of reflected and in-scattered light.

The reflected light can be calculated by obtaining the light incident to the planetary surface in direction w. Note that the distance to the surface point was determined in the outer loop. This allows to calculate the zenith vector and thus the appropriate sun-zenith angle to access the intermediate texture $deltaE$. Consider that the retrieved value needs proper attenuation over the path from the reflection point to the origin of the ray.

To calculate in-scattered light the scattering results of previous iterations have to be accessed by looking up the appropriate values in the tables $deltaSR$ and $deltaSM$. Recall that the phase functions are not yet applied to the stored values in the first iteration (as described in chapter 4.6.1.1) and need to be reintroduced first.

The summation of in-scattered and reflected light is the total light that is incident from direction w and is used as the light term to finally solve Equation 2-10. The results are summed up for each direction w and stored in the table $deltaJ$. This intermediate texture is parameterized like the $inscatter$ table.

Calculating Irradiance at each Position

Calculating the irradiance of multiple scattered light at each position is done by the *irradianceMultiple* shader. For this, the integral over the hemisphere, as described by Equation 2-12, has to be solved.

Similar to the $inscatterMultipleA$ shader, spherical coordinates are used to describe each direction vector w. Listing 4-23 shows how irradiance is calculated for each position.

```
static const float dphi = M_PI / float(IRRADIANCE_INTEGRAL_SAMPLES);
static const float dtheta = M_PI / float(IRRADIANCE_INTEGRAL_SAMPLES);

float4 PS_IrradianceN(PS_In input) : SV_TARGET0
{
  float alt, mus;
  getIrradianceRMuS(alt, mus, input.posH.xy);
  float3 s = float3(sqrt(1.0 - mus * mus), 0.0, mus);

  float3 result = float3(0.0, 0.0, 0.0);

  // integral over 2.PI around current position with two nested loops
  // over w directions (theta,phi)
  for (int iphi = 0; iphi < 2 * IRRADIANCE_INTEGRAL_SAMPLES; ++iphi)
  {
    float phi = (float(iphi) + 0.5) * dphi;
    for (int itheta = 0; itheta < IRRADIANCE_INTEGRAL_SAMPLES / 2; ++itheta)
    {
      float theta = (float(itheta) + 0.5) * dtheta;
      float3 w = float3(cos(phi)*sin(theta),sin(phi)*sin(theta),cos(theta));
      float nu = dot(s, w);
      float dw = dtheta * dphi * sin(theta);
```

```
        if (g_first == 1.0)
        {
          // first iteration => introduce phase functions
          float pr1 = phaseFunctionR(nu);
          float pm1 = phaseFunctionM(nu);
          float3 ray1 = texture4D(g_texDeltaSR, alt, w.z, mus, nu).rgb;
          float3 mie1 = texture4D(g_texDeltaSM, alt, w.z, mus, nu).rgb;
          result += (ray1 * pr1 + mie1 * pm1) * w.z * dw;
        }
        else
        {
          result += texture4D(g_texDeltaSR, alt, w.z, mus, nu).rgb * w.z * dw;
        }
      }
    }

    return float4(result, 1.0f);
  }
```

Listing 4-23: Calculating irradiance due to multiple scattering

The shader utilizes the intermediate tables $deltaSR$ and $deltaSM$ to retrieve the light term of Equation 2-12. In the first iteration, the phase functions have to be reintroduced here as well. After integrating over the hemisphere, the total incident light is transferred to the intermediate table $deltaE$.

Calculating Multiple In-Scattered Light along Path

As described, calculating in-scattered light is a two step process. The first step was done by executing the *inscatterMultipleA* shader, which solved Equation 2-10 and thus calculated the in-scattered light for each position within the atmosphere. In the next step, the total in-scattered light over a path is calculated by solving Equation 2-11. This is done by the *inscatterMultipleB* shader, which is shown in Listing 4-24.

```
float3 integrand(float alt, float mu, float mus, float nu, float t)
{
  float alt_i = sqrt(alt*alt + t*t + 2.0*alt*mu*t);
  float mu_i = (alt * mu + t) / alt_i;
  float mus_i = (nu * t + mus * alt) / alt_i;
  return texture4D(g_texDeltaJ, alt_i, mu_i, mus_i, nu).rgb *
         transmittance(alt, mu, t);
}

float3 inscatter(float alt, float mu, float mus, float nu)
{
  float3 raymie = float3(0.0, 0.0, 0.0);
  float dx = intersectAtmosphere(alt, mu) /
             float(INSCATTER_INTEGRAL_SAMPLES);
  float x_i = 0.0;
  float3 raymie_i = integrand(alt, mu, mus, nu, 0.0);
  for (int i = 1; i <= INSCATTER_INTEGRAL_SAMPLES; ++i)
  {
    float x_j = float(i) * dx;
    float3 raymie_j = integrand(alt, mu, mus, nu, x_j);
    raymie += (raymie_i + raymie_j) / 2.0 * dx;
    x_i = x_j;
    raymie_i = raymie_j;
  }

  return raymie;
}

float4 PS_InScatterMultipleB(PS_InLayer input) : SV_TARGET0
{
  float mu, mus, nu;

  getMuMuSNu(g_r, g_dhdH, mu, mus, nu, input.posH.xy);
  float3 color = inscatter(g_r, mu, mus, nu);

  return float4(color, 1.0f);
}
```

Listing 4-24: Calculating multiple in-scattered light along path

The shader samples the path and retrieves the in-scattered light at each position from the intermediate table $deltaJ$, where the results of the first step were stored. Consider that the in-scattered light needs to be attenuated accordingly on its path from the sample point to the current position.

The remaining light is summed up and stored in the intermediate table $deltaSR$, which acts as the input for the next iteration.

Adding the Intermediate Results to Final Textures

The final two shaders $copyInscatterMultiple$ and $copyIrradiance$ are used to add the contents of the intermediate tables $deltaSR$, $deltaSM$ and $deltaE$ to the final tables.

$copyInscatterMultiple$ retrieves the intermediate results from the $deltaSR$ and $deltaSM$ tables and adds them to the final $inscatter$ table as shown in Listing 4-25.

```
float4 PS_CopyInscatterMultiple(PS_InLayer input) : SV_TARGET0
{
  ...
  float3 color = g_texDeltaS.SampleLevel(LinearSamplerClamp, uvw, 0).rgb /
                 phaseFunctionR(nu);
  return float4(color, 0.0f);
}
```

Listing 4-25: Adding the intermediate results to final the $inscatter$ table

Recall that before storing the in-scattered light into the $inscatter$ table, it is divided by the Rayleigh phase function as described in chapter 4.6.1.1.

$copyIrradiance$ simply adds the contents of the $deltaE$ texture to the final $irradiance$ table as shown in Listing 4-26.

```
float4 PS_CopyIrradiance(PS_In input) : SV_TARGET0
{
    return g_texDeltaE.SampleLevel( LinearSamplerClamp, input.texC, 0
);
```

Listing 4-26: Adding the intermediate results to final the $irradiance$ table

Copying the intermediate results to the final tables concludes the pre-computation step.

4.6.2 Atmospheric Scattering as a Post-Processing Effect

In the following chapters it is shown how the pre-computed tables can be used to apply atmospheric scattering as a single post-processing effect to an arbitrary scene. The proposed method allows smooth and realistic transitions from space to earth, while ensuring accurate atmospheric scattering at all time.

The full source code of the atmospheric scattering shader is provided in Appendix C.

4.6.2.1 Reconstructing World Space Position from Linear Depth Map

For calculating atmospheric scattering as a post-processing effect, the original positions of the rendered pixels need to be known. Note that the proposed GBuffer does not directly store the position of each pixel. Instead, the depth buffer is used to reconstruct this information.

As described in chapter 4.3.1.2 the view frustum corners of the near and far plane are assigned to the corresponding vertices of the screen space quad. These frustum corner positions are used to calculate the $CameraToNear$ and $NearToFar$ vectors as described in Equation 4-3. Listing 4-27 shows how this is done in the vertex shader.

```
float3 frustumFarWorld = mul(float4(g_frustumFar[input.index].xyz, 1.0f),
g_cameraWorld).xyz;
float3 frustumNearWorld = mul(float4(g_frustumNear[input.index].xyz, 1.0f),
g_cameraWorld).xyz;

output.cameraToNear = frustumNearWorld - g_cameraPos;
output.nearToFar = frustumFarWorld - frustumNearWorld;
```

Listing 4-27: Obtaining the distances from near to far plane and from camera to near plane

To assign the correct frustum corner positions to each vertex, the vertices of the screen space quad are extended by an index value. As the frustum corners are defined in view space they have to be transformed to world space by the inverse view matrix or the so called *camera world matrix*, which represents the cameras position and orientation in world space. After this the $CameraToNear$ and $NearToFar$ vectors can be calculated and passed on to the pixel shader.

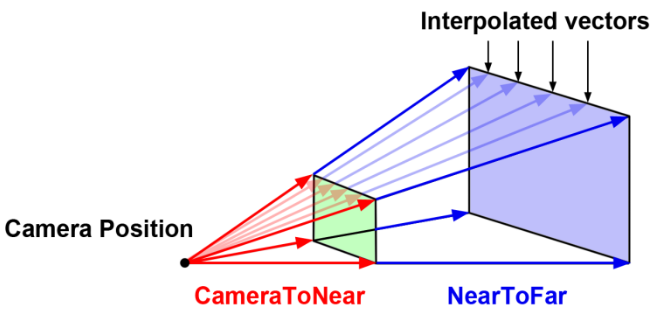

Figure 4-35: Interpolated vectors $CameraToNear$ and $NearToFar$

These vectors are interpolated over each triangle as shown in Figure 4-35 and can be used directly in the pixel shader (Listing 4-28) to reconstruct the original position as described in Equation 4-3.

```
// reconstructing world space postion by interpolation
float depthVal = g_depth.SampleLevel( PointSamplerClamp, input.texC, 0 ).r;
float3 surfacePos = g_cameraPos + input.cameraToNear + depthVal *
input.nearToFar;
```

Listing 4-28: Reconstructing world space position by interpolation between near and far plane

4.6.2.2 Obtaining the View Direction

Accessing the pre-computed tables requires knowledge of the view direction vector. The view direction vector is unique for each processed pixel and describes a vector originating from the camera position and running through each pixel in the image plane. Note that the $CameraToNear$ and $NearToFar$ vectors calculated in the previous chapter already point into this particular direction and can be reused to obtain the view direction vector as shown in Listing 4-29.

```
// obtaining the view direction vector
float3 viewDir = normalize(input.nearToFar);
```

Listing 4-29: Obtaining the view direction vector by reusing the $NearToFar$ vector

4.6.2.3 Ray-Sphere Intersection

To calculate the amount of scattering along the view ray, the path traversed within the atmosphere needs to be known. This path can be obtained by making use of an intersection test of a ray, described by the view direction against a sphere of radius R_t, which represents the outer atmospheric shell around the planet.

The intersection test is used to obtain two values: $offset$ and $maxPathLength$. $offset$ describes the distance from the camera position to the atmosphere along the view direction. If the camera is located within the atmosphere $offset$ will equal zero. $maxPathLength$ describes the actual distance traversed within the atmosphere. Figure 4-36 illustrates the values of $offset$ and $maxPathLength$ based on two examples.

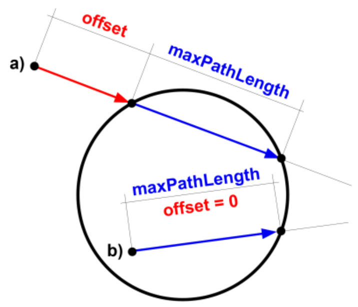

Figure 4-36: Illustrating values of $offset$ and $maxPathLength$ during ray/sphere intersection test based on two examples a) camera located in space b) camera located within the atmosphere

In example a) of Figure 4-36, the ray starts in space and intersects the atmosphere. $offset$ is set to the distance from the ray origin to the outer atmospheric boundary (which equals the first intersection point) and $maxPathLength$ describes the distance that is actually traversed within the atmosphere. The second example b) shows a ray originating within the atmosphere. Therefore $offset$ is set to zero and $maxPathLength$ is set again to the distance that is traversed inside the atmosphere (which equals the distance from the camera to the first and only intersection point).

The intersection test used to obtain these values is based on an optimized test of a ray/sphere intersection proposed by Akenine-Möller et al. [38].

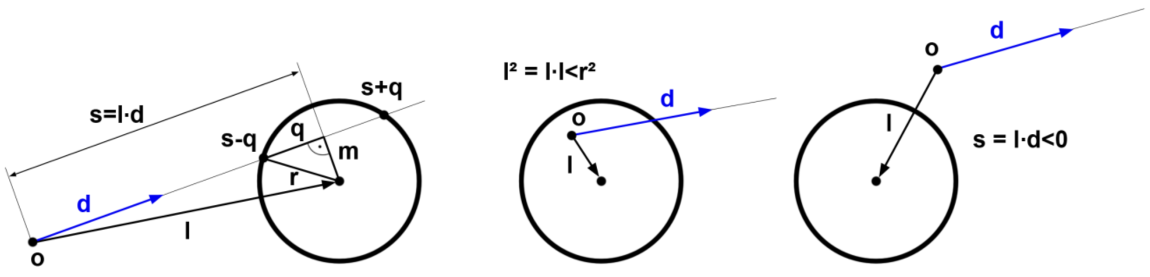

Figure 4-37: Geometrics of the ray/sphere intersection [38]

Figure 4-37 shows the notation of the geometry involved. d represents the view direction and r the radius of the sphere, which equals $R_t - \varepsilon$ (ε describes a very small value, which helps preventing artifacts at the border of the atmosphere). l denotes a vector from the camera position o to the center of the sphere (which is the origin of the coordinate system in the presented approach).

First it is tested if the camera position is inside the atmosphere (which is true if $l^2 \leq r^2$ as seen in the second example of Figure 4-37). In this case a hit is ensured and $maxPath$ is set to the distance from the camera position to the first (and only) intersection point with the outer shell of the atmosphere. This value is found by making use of the Pythagorean theorem to calculate q and the projection of l onto d denoted as s. As the camera is located within the atmosphere, $offset$ is set to zero.

If the camera is located in space an intersection will be detected if $s \geq 0$ and $m^2 \leq r^2$. Note that this allows for an early exit if $s < 0$. In this case the atmosphere is behind the camera position. If an intersection is found $offset$ will be initialized with the distance from the camera position to the first intersection point and $maxPathLength$ will be set to the distance between the first and the second intersection point.

Listing 4-30 shows how the described intersection test is implemented in the pixel shader.

```
bool intersectAtmosphere(in float3 d, out float offset, out float
maxPathLength)
{
	offset = 0.0f;
	maxPathLength = 0.0f;

	// vector from ray origin to center of the sphere
	float3 l = -g_cameraPos;
	float l2 = dot(l,l);
	float s = dot(l,d);
	// adjust top atmosphere boundary by small epsilon to prevent
artifacts
	float r = g_Rt - EPSILON;
	float r2 = r*r;

	if(l2 <= r2)
	{
		// ray origin inside sphere, hit is ensured
		float m2 = l2 - (s * s);
		float q = sqrt(r2 - m2);
		maxPathLength = s + q;

		return true;
	}
	else if(s >= 0)
	{
		// ray starts outside in front of sphere, hit is possible
		float m2 = l2 - (s * s);

		if(m2 <= r2)
		{
			// ray hits atmosphere definitely
			float q = sqrt(r2 - m2);
			offset = s - q;
			maxPathLength = (s + q) - offset;

			return true;
		}
	}

	return false;
}
```

Listing 4-30: Intersecting a sphere of radius R_t with the view ray

Note that if the intersection test returns false, the atmosphere will not be seen by the current view ray. In this case the calculations of in-scattered light can be omitted.

4.6.2.4 Obtaining a Correct Starting Position for the Scattering Path

To calculate light scattering, the path traversed within the atmosphere needs to be determined. Initially, the start and end position are set to the camera and the surface position respectively (the surface position denotes the position that was reconstructed from the depth buffer as described in chapter 4.6.2.1). However, these values may not represent the actual path that is traversed within the atmosphere. In fact, this path is only correct if the camera is located inside the atmosphere and the corresponding view ray hits a surface within it.

Accessing the pre-computed tables requires an altitude of the ray origin, which has to be somewhere between R_g and R_t. However, cameras located in space have an altitude greater than R_t. In this case, these tables have to be accessed by the altitude of the first intersection point as this represents the altitude, at which the traversal of the atmosphere actually starts.

This altitude can be found by offsetting the camera position along the view direction by the distance stored in the $offset$ value as shown by example a) in Figure 4-38. For this, no special handling is required. It is safe to offset the camera position in every case, even when the camera is already located within the atmosphere as the corresponding $offset$ value equals zero in this case.

Additional care has to be taken when an object occludes the atmosphere as shown by example b) in Figure 4-38. This case can be easily determined when the distance stored in $offset$ is greater than the distance to the surface position.

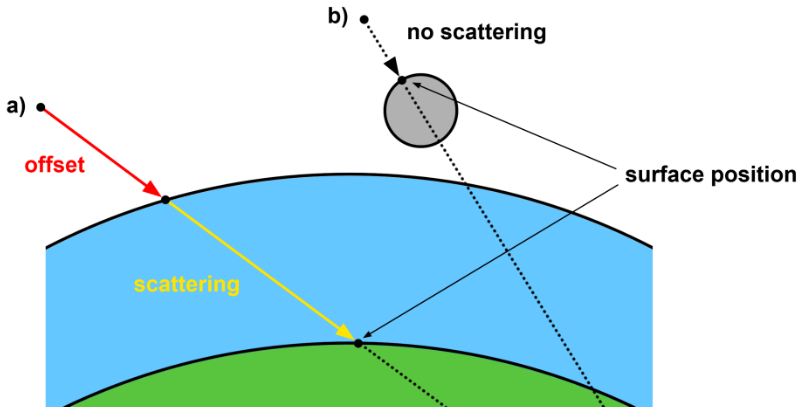

Figure 4-38: Camera located in space: a) offsetting to the first intersection point b) atmosphere occluded by an object

Listing 4-31 shows the offsetting of the camera position as well as handling cases where the atmosphere is occluded.

```
if(intersectAtmosphere(viewDir, offset, maxPathLength))
{
  float pathLength = distance(g_cameraPos, endPos);
  // check if object occludes atmosphere
  if(pathLength > offset)
  {
    // offsetting camera
    float3 startPos = g_cameraPos + offset * viewDir;
    float startPosHeight = length(startPos);
    pathLength -= offset;

    // calculate scattering
     ...

  }
}
```

Listing 4-31: Determining occlusion and offsetting the camera

After passing the ray/sphere intersection test, it is first checked whether an object is in front of the atmosphere or not, by comparing the path length from the camera to the surface position with the value $offset$ obtained during the intersection test. Recall that $offset$ is set to the distance from the ray origin to the atmosphere. If the camera is already within the atmosphere $offset$ equals zero. This means that the atmosphere will be occluded if the distance to the surface position is smaller than the distance stored in $offset$. In this case no atmospheric scattering will happen and further calculations will be omitted.

As described, offsetting the camera position along the view ray ensures a correct start position of the scattering path and has no effect if the camera is located within the atmosphere.

4.6.2.5 Calculating In-scattered Light

Once the correct start position of the scattering path is obtained, the cosine angles between the view direction and the zenith, the view direction and the sun direction and the sun direction and the zenith vector need to be determined. After this, the in-scattered light for the given path can be calculated.

At first the in-scattered light for an infinite ray originating from the start position is retrieved from the $inscatter$ table. Note that this value will only be correct if the view ray traverses the atmosphere entirely, which means it does not hit an object in between.

If this is not the case and the view ray hits an object inside the atmosphere, the in-scattered light needs to be adjusted to simulate a finite ray as shown in Figure 4-39. For this the in-scattered light at the surface position is retrieved and subtracted from the in-scattered light at the start position (as described in chapter 4.6.1.1).

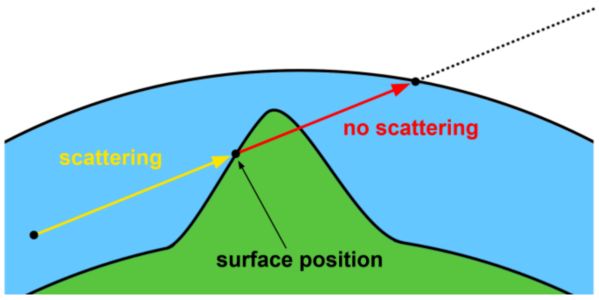

Figure 4-39: The view ray hits an object inside the atmosphere. In-scattered light for a finite ray (yellow) needs to be determined by subtracting in-scattered light at the surface position (red)

Listing 4-32 shows the calculation of the view angles and the in-scattered light.

```
// starting position of path is now ensured to be inside atmosphere
// was either originally there or has been moved to top boundary
float muStartPos = dot(startPos, viewDir) / startPosHeight;
float nuStartPos = dot(viewDir, g_sunVector);
float musStartPos = dot(startPos, g_sunVector) / startPosHeight;

// in-scattering for infinite ray (light in-scattered when no surface hit or
object behind atmosphere)
float4 inscatter = max(texture4D(g_texInscatter, startPosHeight, muStartPos,
musStartPos, nuStartPos), 0.0f);

// check if object hit is inside atmosphere
if(pathLength < maxPathLength)
{
  // reduce total in-scattered light when surface hit within atmosphere
  attenuation = transmittance(startPosHeight, muStartPos, pathLength);

  float surfacePosHeight = length(surfacePos);
  float musSurfacePos = dot(surfacePos, g_sunVector) / surfacePosHeight;
  float muSurfacePos = dot(surfacePos, viewDir) / surfacePosHeight;
  float4 inscatterSurface = texture4D(g_texInscatter, surfacePosHeight,
muSurfacePos,                                 musSurfacePos, nuStartPos);
  inscatter = max(inscatter - attenuation.rgbr *   inscatterSurface, 0.0f);
  irradianceFactor = 1.0f;
}
else
{
  // retrieve extinction factor for inifinte ray
  attenuation = transmittance(startPosHeight, muStartPos);
}

float phaseR = phaseFunctionR(nuStartPos);
float phaseM = phaseFunctionM(nuStartPos);
inscatteredLight = max(inscatter.rgb * phaseR + getMie(inscatter) * phaseM,
0.0f);
```

```
inscatteredLight *= sunIntensity;
```

Listing 4-32: Calculating in-scattered light

As shown in Listing 4-32, an additional irradiance factor variable is set when a surface within the atmosphere is hit. This variable indicates that this pixel is affected by irradiance and saves an additional if-statement when calculating the reflected light.

Note that retrieving an extinction factor is only needed for finite rays when determining the in-scattered light. However, this value is required when calculating the reflected light, so it is retrieved for infinite rays as well.

After obtaining the appropriate values from the tables, the phase functions are applied. Note that the Mie channels get reconstructed by making use of the proportion rule described in chapter 4.6.1.1. Finally, the result is scaled by the intensity of the sunlight.

4.6.2.6 Calculating Reflected Light

In order to access the correct value in the $irradiance$ table, the altitude of the surface position as well as the cosine angle between the sun direction and the zenith vector at this particular position needs to be determined.

Recall that the $irradiance$ table does not consider direct sunlight that hits a point within the atmosphere (as described in chapter 4.6.1.1). To calculate this value the surface normal stored in the GBuffer is retrieved and used to obtain the $lightScale$ term as described in Equation 4-10. Consider that this value needs to be scaled by the extinction factor from the position where the sunlight enters the atmosphere to the position of the surface.

Both, the irradiance value stored in the table and the direct sunlight hitting a surface, are reflected. The reflected light is calculated by retrieving the reflectance value and the color value stored in the GBuffer.

Note that the reflected light needs to be attenuated on its way from the surface to the camera. For this, the attenuation value that was retrieved during the calculation of in-scattered light, is reused.

Listing 4-33 shows the calculation of reflected light.

```
// read contents of GBuffer
float4 normalData = g_normal.SampleLevel(PointSamplerClamp, texC, 0);
float3 surfaceColor = g_color.SampleLevel(PointSamplerClamp, texC, 0).rgb;

// decode normal and determine intensity of refected light at surface
postiion
float3 normal = 2.0f * normalData.xyz - 1.0f;
float lightIntensity = sunIntensity * normalData.w;
float lightScale = max(dot(normal, g_sunVector), 0.0f);

// irradiance at surface position due to sky light
float3 irradianceSurface = irradiance(g_texIrradiance, surfacePosHeight,
musSurfacePos) * irradianceFactor;

// attenuate direct sunlight on its path from top of atmosphere to surface
position
float3 attenuationSunLight = transmittance(surfacePosHeight, musSurfacePos);
float3 reflectedLight = surfaceColor * (lightScale * attenuationSunLight +
irradianceSurface) * lightIntensity;

// attenuate again on path from surface position to camera
reflectedLight *= attenuation;
```

Listing 4-33: Calculating reflected light

4.6.2.7 Composing the Final Image

The final color is then calculated by a simple summation of direct sunlight, in-scattered light and reflected light as described in Equation 2-8. Recall that the direct sunlight is basically part of the reflected light (as described in chapter 4.5.3) and therefore does not need to be considered explicitly.

Listing 4-34 shows the composing of the final image.

```
float3 inscatteredLight = GetInscatteredLight(surfacePos, viewDir,
attenuation, irradianceFactor);
float3 reflectedLight = GetReflectedLight(surfacePos, input.texC,
attenuation, irradianceFactor);

return float4(HDR(reflectedLight + inscatteredLight), 1.0f);
```

Listing 4-34: Composing the final color

The function HDR is a simple color scaling function to prevent color banding and is described in the next chapter in further detail.

Figure 4-40 shows each component of three different scenes with varying altitude and time of the day and their final composition.

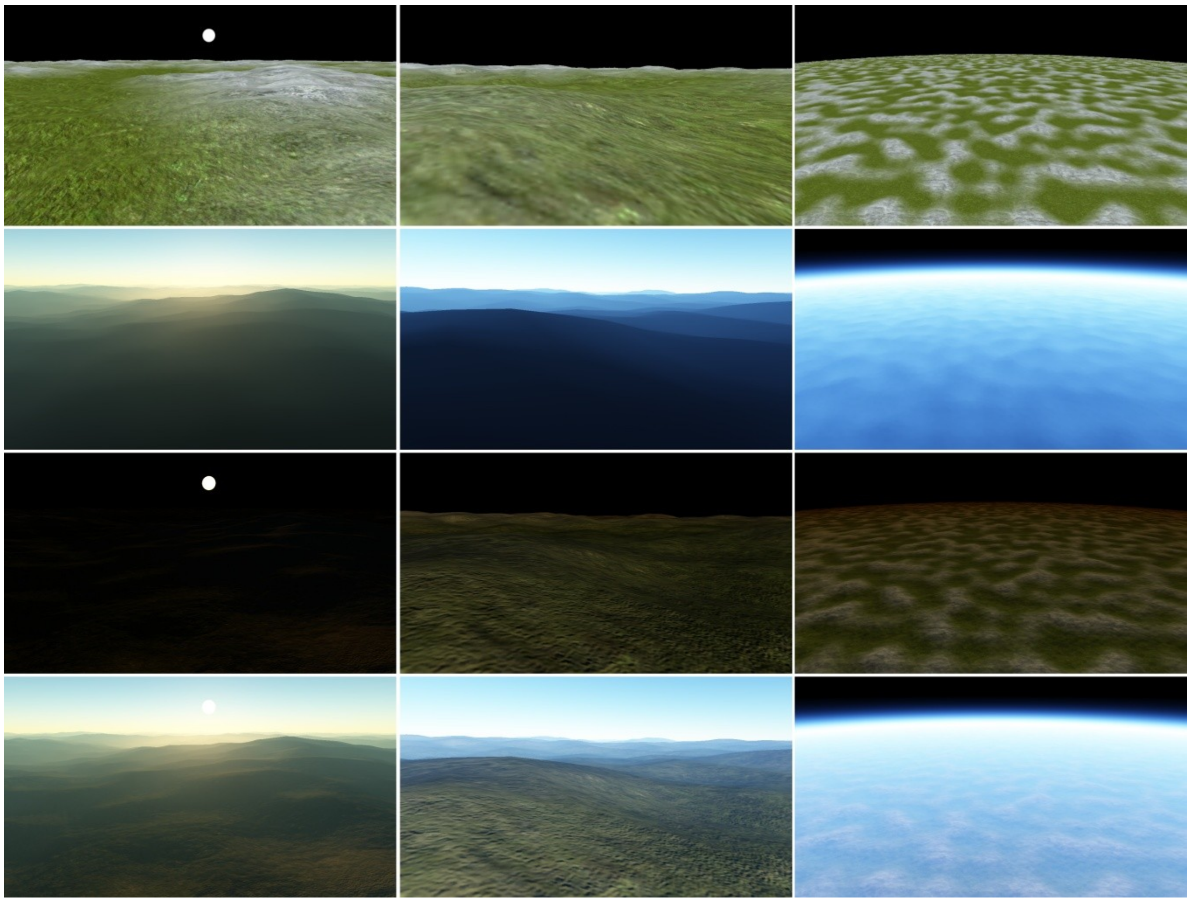

Figure 4-40: Anatomy of three scenes (from top to bottom): Color value stored in the GBuffer, in-scattered light (colors scaled for better visualization), reflected light and final composed image

4.6.3 High Dynamic Range Rendering

An issue that needs proper handling in nearly all atmospheric scattering models is scaling of colors to prevent color banding artifacts. Color banding refers to the replacement of smooth color gradients by a set of visible color bands. This effect is usually the result when an image is displayed by a reduced palette.

Common displays offer a color palette of 8 bit per channel, which results in approximately 16.7 million discrete values. On the graphics hardware these 8 bit are mapped to the range $[0, 1]$. An image is said to have a *high dynamic range* (HDR) if parts of it exceed this limited palette. Displaying these images without proper handling then may result in color banding artifacts. HDR Rendering refers to rendering of HDR images, by mapping these colors to a displayable color range without producing visible artifacts due to the reduced palette. This color scaling is usually referred to as *tone mapping*. Tone mapping is done by applying certain *tone mapping operators*. One of the most famous tone mapping operator is proposed by Reinhard et al. [39].

Atmospheric scattering is prone to color banding, because the equations can easily produce images that are too bright [6]. The resulting image is then often not properly displayable due to the limitation of 8 bit per channel as shown in Figure 4-41.

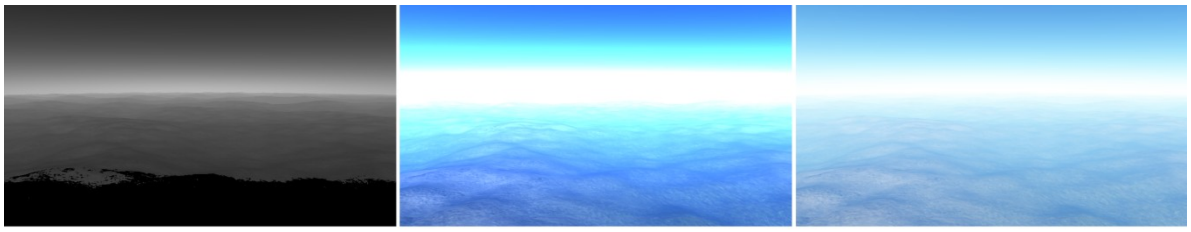

Figure 4-41: Importance of considering HDR: (left) parts of the scene where at least one channel exceeds the range $[0,1]$ (middle) without tone mapping visible color banding artifacts occur (right) remapped colors to prevent color banding

Equation 4-19 shows the tone mapping operator [6] used to scale the images to a displayable color range.

$$scaledColor = 1 - e^{(-exposure \, * unscaledColor \;)} \qquad \text{Equation 4-19}$$

Where $scaledColor$ describes the resulting color that is displayed, $exposure$ a constant that controls the steepness of the scaling curve and $unscaledColor$ the unmodified color value that has to be remapped.

Figure 4-42 shows a scene where this tone mapping operator is applied using varying values for exposure.

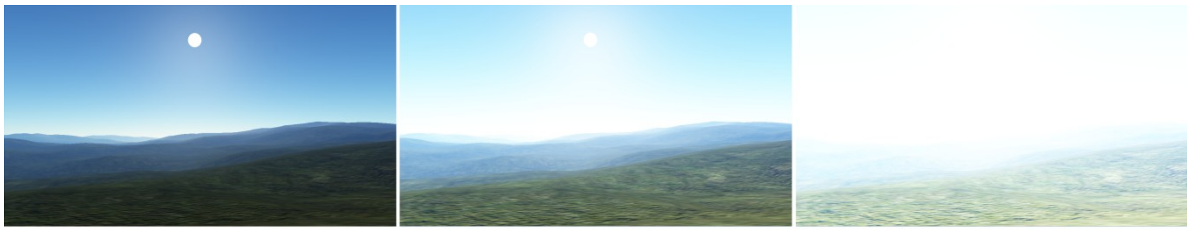

Figure 4-42: Tone mapping with different settings for exposure value: (left) exposure = 1 (middle) exposure = 3 (right) exposure = 10.

The corresponding color scaling curves are shown in Figure 4-43.

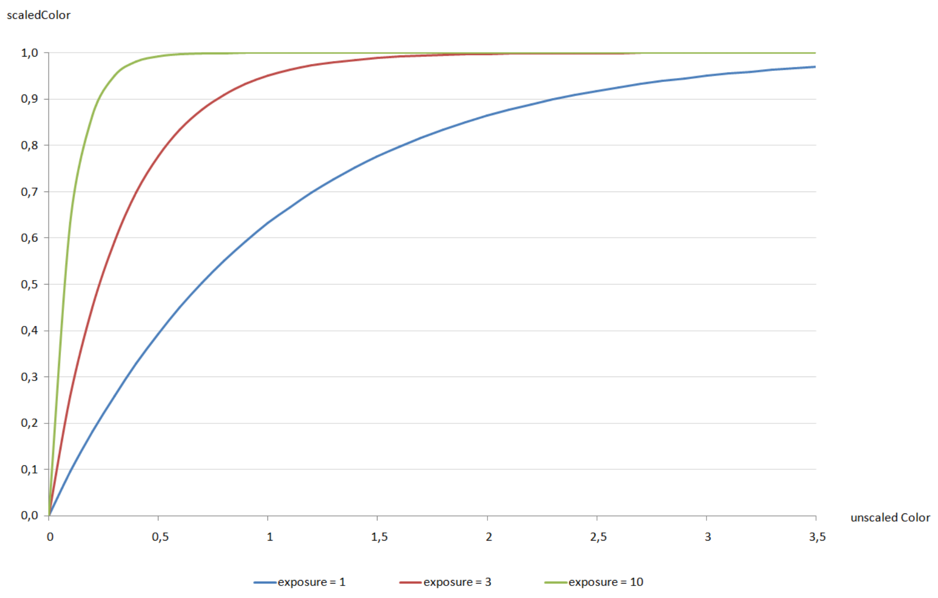

Figure 4-43: Color scaling curves with varying values for exposure

Usually, tone mapping operators scale the colors of an HDR image based on its luminance. However, for simplicity this is not the case in the presented approach and therefore uses the same exposure value for every scene regardless of its average luminance. Listing 4-35 shows the implementation of Equation 4-19 in the pixel shader.

```
float3 HDR(float3 color)
{
  return 1.0f - exp(-EXPOSURE * color);
}
```

Listing 4-35: Tone mapping function used in the presented model

HDR rendering is a vastly researched topic in computer graphics and this chapter is primarily meant to raise awareness of its importance. Additionally, a simple yet effective solution is presented how color banding can be avoided.

A comprehensive introduction to this topic is given by Akenine-Möller et al. [38]. Hable [40] provides an in-depth explanation of various aspects regarding HDR Rendering and so called filmic tone mapping. An extensive comparison of various tone mapping operators is presented by Čadík [41].

4.7 Results

The following chapters show the results of the presented approach. This includes seamless transitions from space to the planetary surface, differences between single and multiple scattering, the results of varying Rayleigh and Mie scattering parameters and the correct handling of objects that occlude and partially intersect the atmosphere. Unless explicitly stated otherwise, the rendering parameters were defined as shown in Table 4-1 and Table 4-2.

$Blocksize$:	33
C:	50
$Max. Elevation$:	4
$Max. Recursion$:	9

Table 4-1: Terrain parameters used for presenting the results

$\beta_R^s(red, green, blue)$:	$5.8 * 10^{-3}, 1.35 * 10^{-2}, 3.31 * 10^{-2}$
$\beta_R^e(red, green, blue)$:	$same\ as\ \beta_R^s$
H_R:	8.0
$\beta_M^s(red, green, blue)$:	$4.0 * 10^{-3}, 4.0 * 10^{-3}, 4.0 * 10^{-3}$
$\beta_M^e(red, green, blue)$:	$\frac{\beta_M^s(red, green, blue)}{0.9}$
H_M:	1.2
g:	0.8
$sun\ intensity$:	30
α:	0.1
R_g:	6360
R_t:	6420
$Exposure$:	2
$size\ transmittance\ table$:	$256x64$
$size\ irradiance\ table$:	$64x16$
$size\ inscatter\ table$:	$256x128x32$
$table\ format$:	$16\ bit\ floating\ point$
$multiple\ scattering\ iterations$:	3

Table 4-2: Scattering parameters used for presenting the results

4.7.1 Seamless Transition from Space to Planetary Surface

The presented model allows seamless transitions from space to the planetary surface while ensuring correct atmospheric scattering at all time. Figure 4-44 shows an approach from outer space in a sunset and midday scenario.

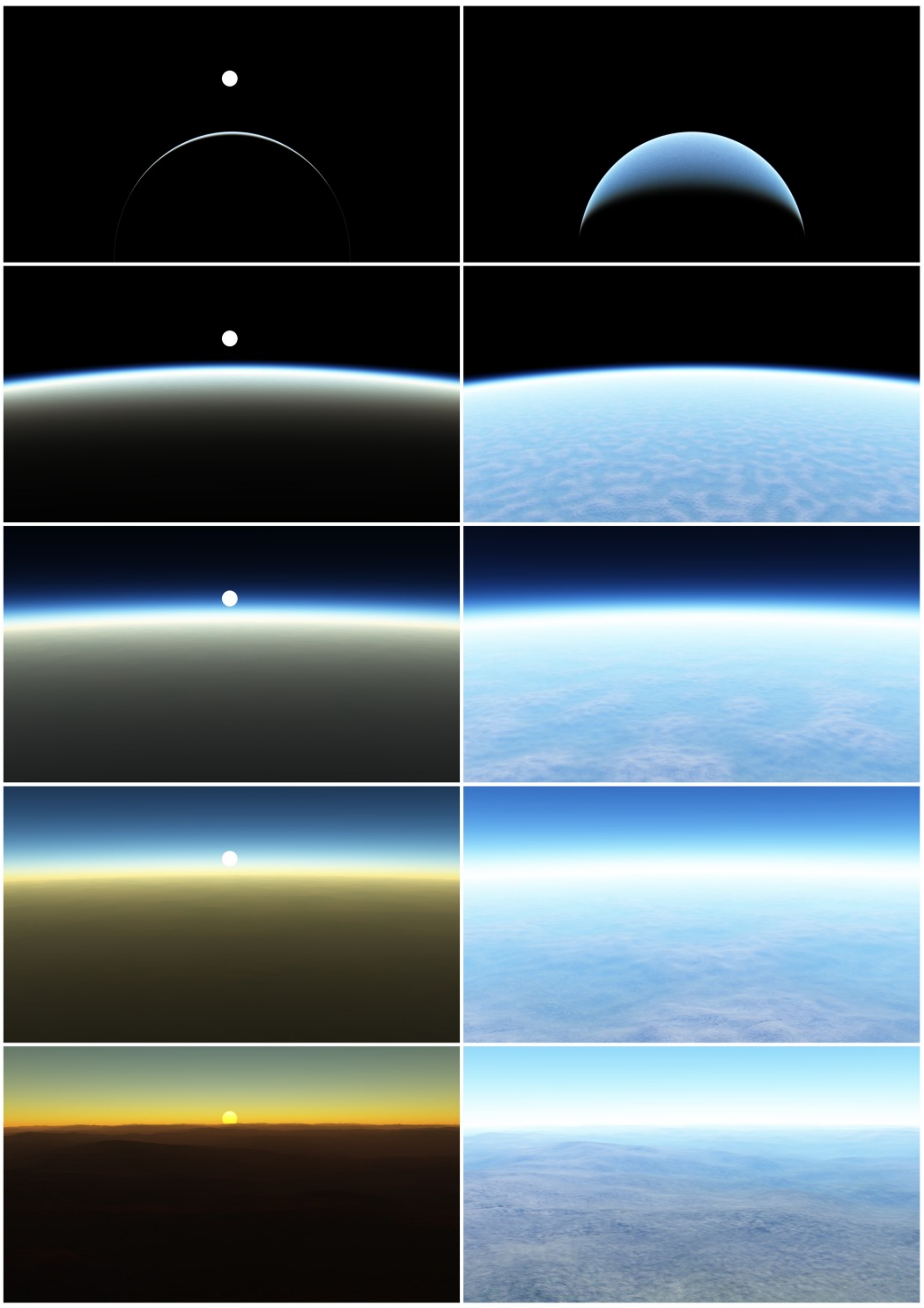

Figure 4-44: Seamless transitions from space to planetary surface: (left) at sunset (right) at midday

4.7.2 Comparing Single and Multiple Scattering

Although contributions of multiple scattering are subtle, it greatly enhances believability and realism. Figure 4-45 shows the differences between single scattering and multiple scattering. For this various scenes are considered. As can be seen, considering just single scattering results in a too dark scene. By taking single and multiple scattering into account, the density of the atmospheric media appears thicker and colors are stronger shifted towards blue.

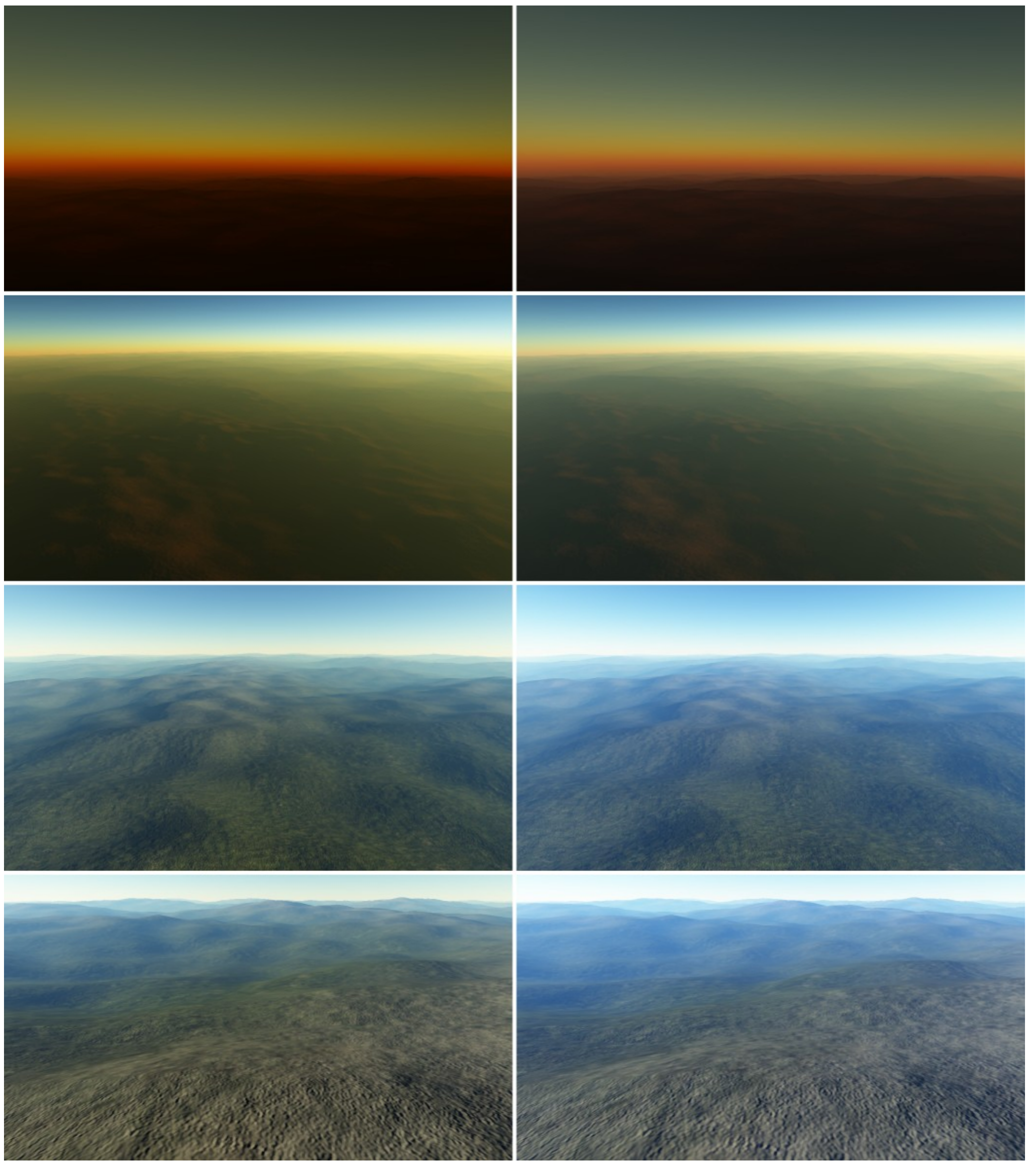

Figure 4-45: Contributions of multiple scattering:
(left) single scattering only (right) single and multiple scattering

4.7.3 Rayleigh Scattering

As described in chapter 2.7, Rayleigh scattering is responsible for the color of the sky. However, it also highly influences perception of reflected light. Thus, changing the parameters of Rayleigh scattering has a great impact on the resulting image.

Figure 4-46 shows the effects of adjusting the scattering coefficient β_R^s (which equals the appropriate extinction coefficient β_R^e as described in chapter 2.2) on the resulting image. As scattering varies with altitude, three different scenes are considered. As can be seen, reducing the amount of scattering darkens the color of the sky and lowers the shift of reflected light towards blue.

Adjusting the scale height H_R is shown in Figure 4-47. This parameter primarily affects the perceived thickness of the atmospheric layer surrounding the planet.

4.7.4 Mie Scattering

The visible halo surrounding the sun is a direct result of Mie scattering (as described in chapter 2.7). Thus the effects of Mie scattering are best observed when looking directly in the direction of the sun.

Figure 4-48 shows the impact of variations on the Mie scattering coefficient β_M^s (the extinction coefficient β_M^e remains $\frac{\beta_M^s}{0.9}$). As can be seen, increasing this value results in a larger halo effect when near the ground. The impact of adjusting the Mie scattering coefficient has nearly no effect when the camera is high above the ground as the density of aerosols decreases rapidly with altitude.

Figure 4-49 shows the results of varying the scale height H_M. As can be seen the effects of Mie scattering are now clearly visible even at higher altitudes.

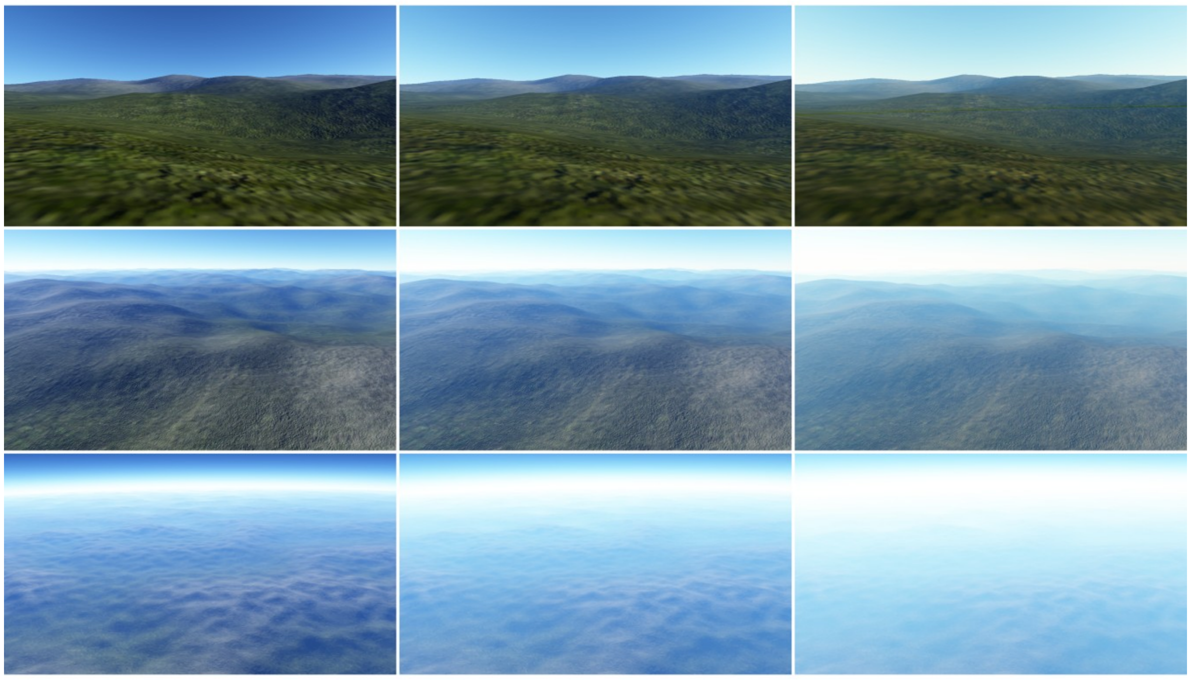

Figure 4-46: Varying the scattering coefficient β_R^s

(top row) altitude: $6362km$ (middle row) altitude: $6365km$ (bottom row) altitude: $6380km$

(left) β_R^s: $2.9 * 10^{-3}, 0.65 * 10^{-2}, 1.67 * 10^{-2}$

(middle) β_R^s: $5.8 * 10^{-3}, 1.35 * 10^{-2}, 3.31 * 10^{-2}$ (default)

(right) β_R^s: $1.16 * 10^{-2}, 2.7 * 10^{-2},\ 6.62 * 10^{-2}$

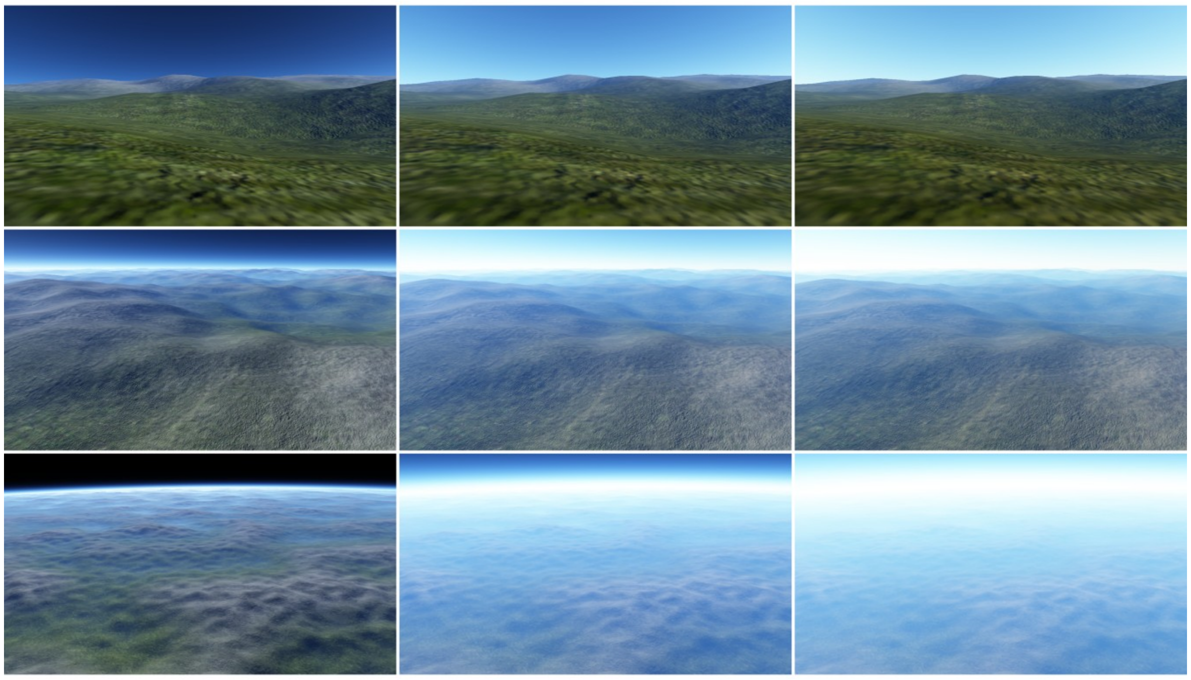

Figure 4-47: Varying the scale height H_R

(top row) altitude: $6362km$ (middle row) altitude: $6365km$ (bottom row) altitude: $6380km$

(left) H_R: 2 (middle) H_R: 6 (right) H_R: 10

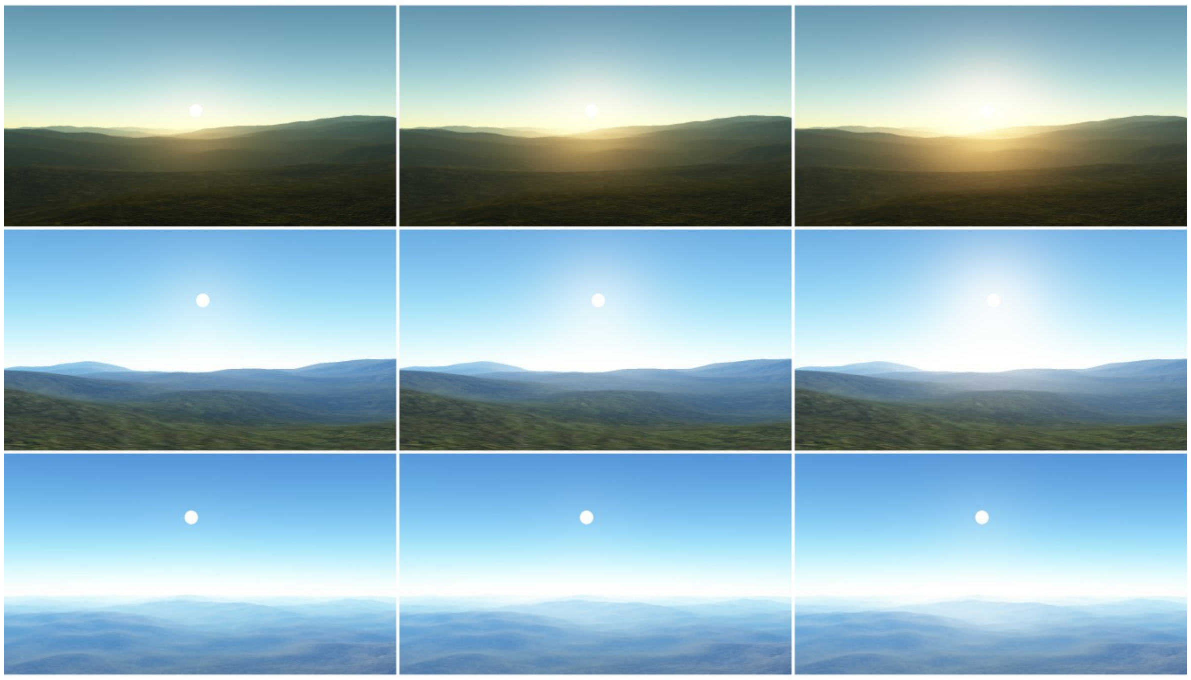

Figure 4-48: Varying the scattering coefficient β_M^s

(top row) altitude: $6361km$ (middle row) altitude: $6363km$ (bottom row) altitude: $6365km$

(left) $\beta_M^s: 2.0 * 10^{-3}, 2.0 * 10^{-3}, 2.0 * 10^{-3}$

(middle) $\beta_M^s: 4.0 * 10^{-3}, 4.0 * 10^{-3}, 4.0 * 10^{-3}$ (default)

(right) $\beta_M^s: 8.0 * 10^{-3}, 8.0 * 10^{-3}, 8.0 * 10^{-3}$

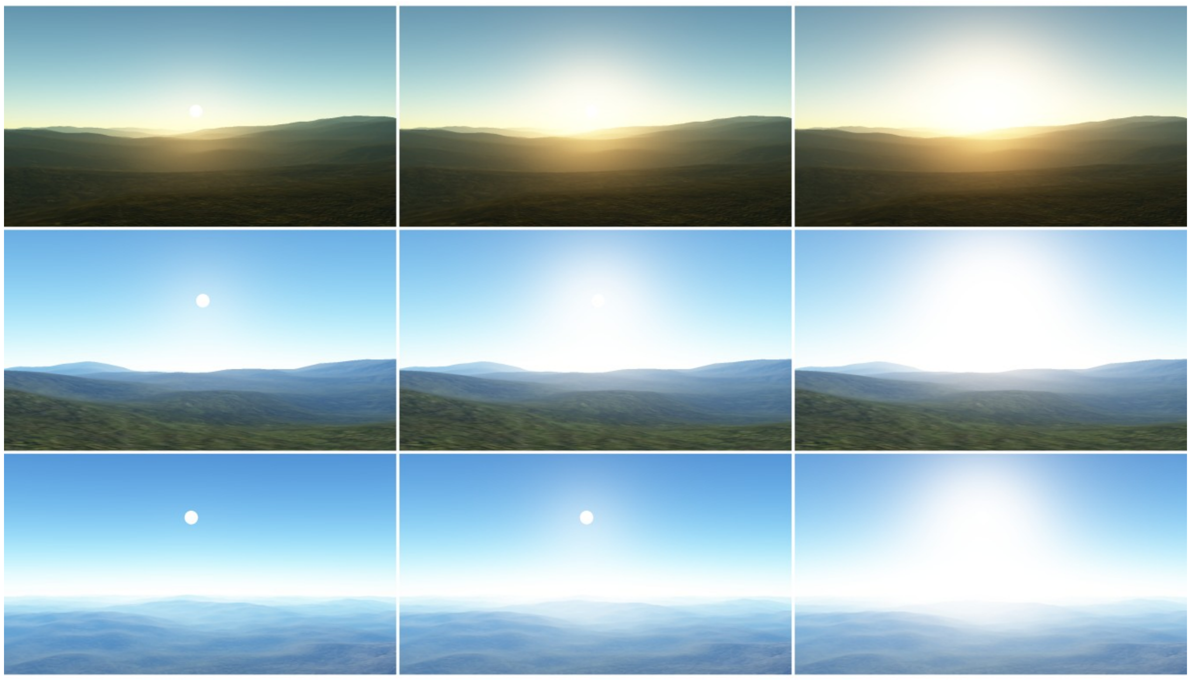

Figure 4-49: Varying the scale height H_M

(top row) altitude: $6361km$ (middle row) altitude: $6363km$ (bottom row) altitude: $6365km$

(left) $H_M: 1$ (middle) $H_M: 2$ (right) $H_R: 4$

4.7.5 Objects Occluding and Partially Intersecting the Atmosphere

Finally it is demonstrated that the proposed approach can be applied to arbitrary scenes and thus correctly handles occlusions and partial intersections of the atmosphere as shown in Figure 4-50.

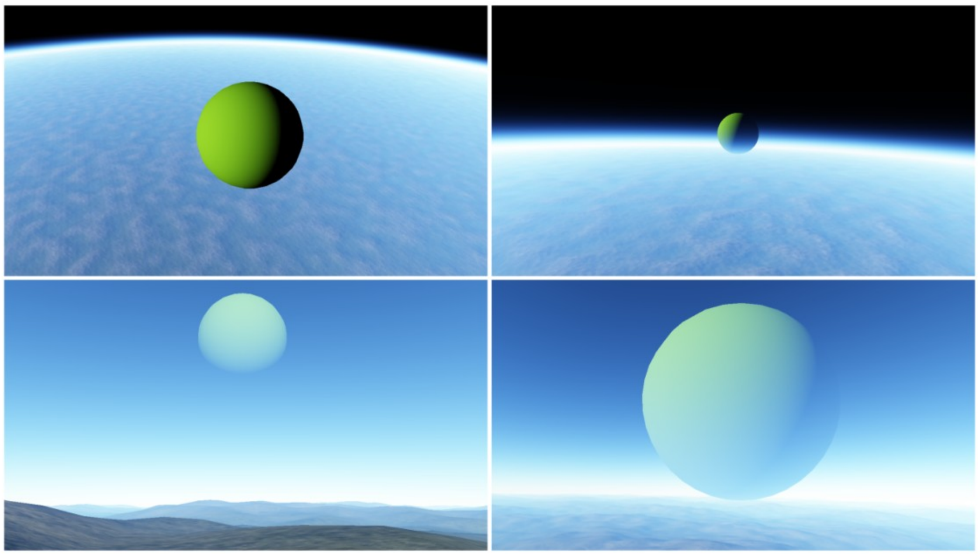

Figure 4-50: Correct scattering of occluding and intersecting objects:
(left) object completely outside atmosphere (right) object partially inside atmosphere

5 Limitations and Future Work

The following chapters reveal the limitations of the presented approach and discuss possible improvements to overcome these.

5.1 Precision Issues

Correct atmospheric scattering is highly dependent on accurate calculations. However, view angles near the horizon can quickly result in calculations with very high values, which can yield precision problems. These precision problems are reinforced by the fact, that the results are stored in a 16 bit floating point table, which offers only half the precision of floating point values in the shader. These issues can lead to slight artifacts near the horizon.

However, the provided source code of Bruneton et al. [37] offers various fixes to overcome these precision problems. The following chapters briefly present these methods. The post-processing shader provided in Appendix C already contains these fixes.

5.1.1 Analytic Transmittance

Using the transmittance texture to obtain the extinction factor causes noise effects near the horizon. This can be avoided by replacing the transmittance texture by an analytic formula, which approximates the extinction factor at run-time. Figure 5-1 shows that this slightly reduces the occurring noise.

Calculating the extinction factor at runtime, results in a minor performance drop of approximately 2%.

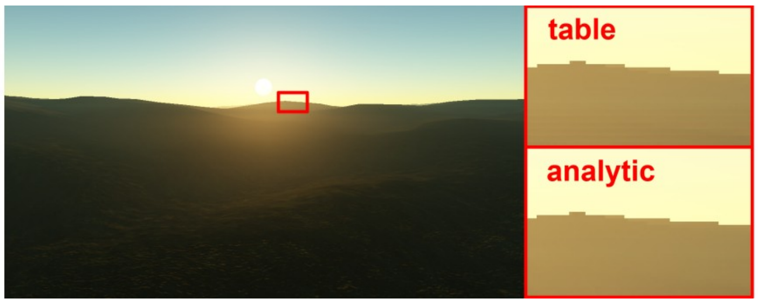

Figure 5-1: Imprecision problems of $transmittance$ table are solved using an analytic formula

5.1.2 Artifacts at In-scattered Light

The imprecision problems near the horizon also affect calculations of the in-scattered light, which can cause serious artifacts. However, the provided source code handles these problems by sampling the $inscatter$ texture above and below the horizon. These samples are then interpolated accordingly. Basically, this just prevents calculations with imprecise data. Figure 5-2 shows that this removes the artifacts considerably.

Although the actual performance drop is highly dependent on how fast the 3D texture can be sampled, the expected loss in speed is about 5%.

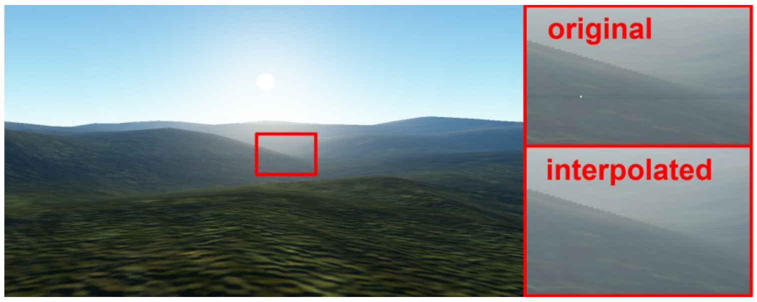

Figure 5-2: Imprecision problems of $inscatter$ table are solved by interpolating between two sample points above and below horizon

5.1.3 Artifacts at Mie Scattering

Imprecision problems of Mie scattering appear when the sun is slightly below the horizon. The provided source code handles this issue by avoiding Mie scattering entirely in this case. For this the effect of Mie scattering quickly fades out when the sun passes the horizon as shown in Figure 5-3.

The performance costs of this are insignificant and clearly below 1%.

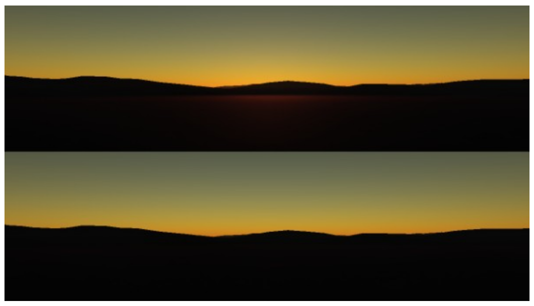

Figure 5-3: Artifacts of Mie scattering can be avoided by removing Mie scattering entirely when the sun is below horizon

5.2 Reducing Memory Consumption

Using pre-computed tables is a typical trade-off between speed and memory consumption. In the presented model memory requirements are primarily affected by the dimensions of the 3D $inscatter$ texture. In this chapter three approaches of reducing memory requirements are proposed.

5.2.1 Reducing Variable Parameters

The most obvious approach to minimize memory consumption is to reduce the variable parameters of the final tables.

The altitude may be a good candidate for such an optimization. By assuming a fixed altitude of the camera, the dimension of the pre-computed tables can be reduced dramatically. However, this won't allow seamless transitions from space to the planetary surface anymore. Note that reducing the corresponding dimension to a single entry is not recommended, as calculating in-scattered and reflected light on surfaces with higher or lower altitudes than the camera may be required. Instead, the corresponding dimensions should be reduced to a reasonable range. A similar approach is to assume a fixed time of the day.

Elek [42] proposes a reduction of the $inscatter$ table to ignore the shadow of the planet itself and thus a removal of the sun-view angle parameter. This results in a table similar to the one proposed by Schafhitzel et al. [11].

5.2.2 Compression of Look-up Tables

Another possibility to reduce memory consumption is to store the pre-computed tables in a different texture format. However, this may require proper encoding as texture formats below 16 bit usually clamp the stored values between the range $[0,1]$. In certain cases, this will be a viable option if memory requirements are a critical factor.

For demonstration purposes, the $inscatter$ texture is converted to an 8 bit unsigned integer format. The values of the original table are scaled appropriately to utilize the whole bit range as well as to allow storage in the normalized range. The appropriate scaling parameters depend on the values that are actually stored in the table. In the presented example, the red, green and blue channels store maximum values of approximately 1.6. Thus, they are scaled by $\frac{1}{1.6}$, which maps them to a range between 0 and 1. The Mie channel stores very low values between 0 and 0.256. To utilized the whole bit range, they are scaled by $\frac{1000}{256}$. Listing 5-1 shows the corresponding encoding and decoding functions. Recall that Mie scattering is stored in the alpha channel of the $inscatter$ texture and is represented by a single floating point value.

```
float3 encodeRayleigh(float3 rayleigh)
{
        return rayleigh /= 1.6f;
}

float3 decodeRayleigh(float3 rayleighEncoded)
{
        return rayleighEncoded *= 1.6f;
}

float encodeMie(float mie)
{
    return (mie * 1000.0f) / 256.0f;
}

float decodeMie(float mie)
{
        return (mie * 256.0f) / 1000.0f;
}
```

Listing 5-1: Encoding and Decoding functions used to store the original 16 bit floating point values in a normalized unsigned integer table format

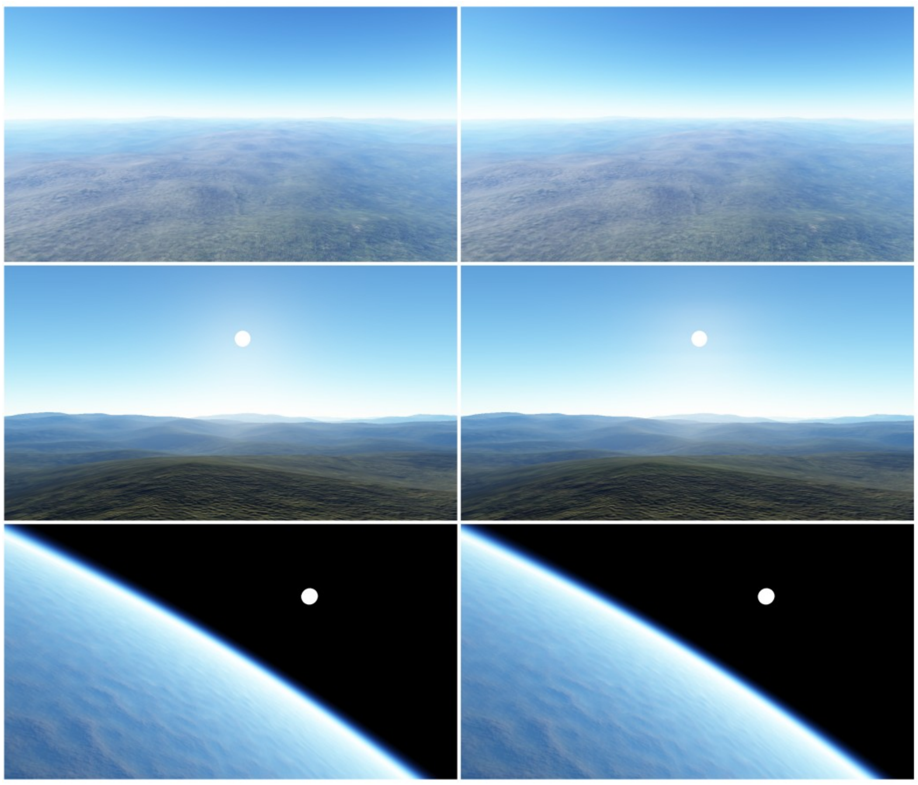

Figure 5-4: Comparing results of using 16 bit and 8 bit table in a midday and afternoon scenario
(left) *inscatter* table using 16 bit floating point format
(right) *inscatter* table using 8 bit unsigned integer format

As can be seen in Figure 5-4, the differences are marginal in a midday and afternoon scenario. However, due to the limited precision, color banding artifacts appear when Mie scattering sets in. As can be seen in Figure 5-5 these artifacts are getting more serious when Mie scattering is prevailing at sunset.

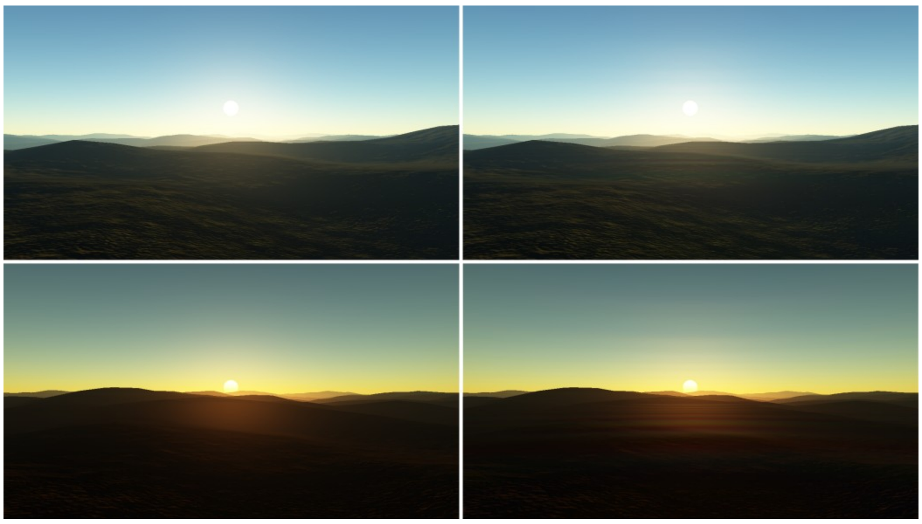

Figure 5-5: Color banding effects at Mie scattering when using 8 bit table
(left) $inscatter$ table using 16 bit floating point format
(right) $inscatter$ table using 8 bit unsigned integer format

Note that the presented encoding/decoding functions can be optimized considerably and were only used to demonstrate the possibility of storing the $inscatter$ texture into an 8 bit unsigned integer format.

5.2.3 Analytic transmittance

As described in chapter 5.1.1 an analytic formula can be used to replace the $transmittance$ table during runtime. However, this is only mentioned here for completeness as the memory consumption of the $transmittance$ table is insignificant.

5.3 Shadows and Lightshafts

As emphasis of the proposed model is laid on the actual integration of atmospheric scattering into a deferred rendering pipeline, not all of the features proposed by Bruneton and Neyret [1] are actually implemented.

This concerns especially the lack of shadows due to occlusion of the sun. In their paper, Bruneton and Neyret propose a shadow volume algorithm that is used to limit the in-scattered light to the path from the camera to the outer boundary where the shadowed region starts. This approach not only simulates the shadows of the terrain, it also implicitly results in lightshafts.

Note that shadows imply that parts of the terrain which are clearly outside the view frustum may contribute to the final image. Thus, adapting this technique to the presented model requires adjustments to the LOD functionality of the terrain.

5.4 Level of Detail Artifacts

Depending on the LOD constant C and the triangle count of a single terrain block, seams may appear between different LOD stages. To a certain degree these artifacts are hidden by the scattering of light. However, in certain cases, especially when the camera looks straight down on the planetary terrain, atmospheric effects are minimized and these seams may be clearly visible. Adjusting the vertices near the outer boundary can be a solution to create a seamless transition to the next LOD stage.

A slightly different problem is the fact that lighting of a terrain block suddenly changes when switching between different LOD stages. This happens because the tangent vectors used to obtain the normal are calculated in the vertex shader and thus dependent on the refinement of the terrain. Moving these calculations to the pixel shader may result in a huge performance drop, as the computational expensive Perlin noise function would need to be executed several times for each pixel. Instead it is recommended to hide these artifacts by slowly blending between different LOD stages based on a morphing parameter, as proposed by Vines [19].

5.5 Lack of Artistic Control

Atmospheric scattering in the presented model follows predefined rules, which ensure physical correctness at all times. This means that artistic control is almost non-existent. Hence, generating artificial atmospheres with a specific behavior can be difficult and is to a certain degree even impossible.

6 Conclusion

This thesis presented an approach to deferred rendering that allows applying atmospheric scattering to a planetary terrain as a post-processing technique.

The first chapters covered the significance of atmospheric scattering in outdoor scenarios and introduced the reader to the appropriate physical model by presenting the most important light transfer equations. In addition, various phenomena due to the scattering of light in the atmospheric media were examined in further detail, like the shifting colors of distant objects, the color of the sky during midday and sunset as well as the visible halo that surrounds the sun.

The main part of this thesis first presented an approach of rendering a planetary terrain on various scales. Allowing seamless transitions from space to ground was achieved by implementing a tiled block algorithm that was extended with LOD and frustum culling capabilities. The LOD algorithm allowed a gradient refinement of the planetary surface based on the distance to the camera and the frustum culling functionality reduced rendering costs to a minimum when near the ground. Furthermore, it was shown how planetary terrains can be generated procedurally by using noise functions. To preserve details near the ground it was shown how normal mapping and multi-texturing can be applied as well.
After this it was described how the sun is rendered and how direct sunlight can be expressed as reflected light. This helped reducing complexity of the final post-processing shader.
The next part finally showed how atmospheric scattering is computed. For this, the pre-computation process proposed by Brunteon and Neyret [1] was examined in further detail. The generated tables were then used to apply atmospheric scattering as a post-processing effect to the planetary terrain. Similar to the corresponding equations, the final image was composed as a summation of in-scattered, reflected and direct sunlight.
In the last part the results of the proposed approach were presented. For this it was shown how a variation of parameters affects the representation of the atmosphere. In addition, the correct behavior of the presented approach was demonstrated under various conditions.

The last chapter revealed the limitations of the proposed approach and suggested various improvements for future work.

Bibliography

[1]. **Bruneton, Eric and Neyret, Fabrice.** Precomputed Atmospheric Scattering. *Eurographics Symposium on Rendering 2008.* Vol. 27, 4.

[2]. **Rayleigh, Lord.** On the scattering of light by small particles. *Philosophical Magazine.* 1871, 41, pp. 447-451.

[3]. **Mie, Gustav.** Beiträge zur Optik trüber Medien, speziell kolloidaler Metallösungen. *Annallen der Physik.* 1908, Vol. 25, 3, p. 377.

[4]. **Nishita, Tomoyuki, et al.** Display of the earth taking into account atmospheric scattering. *SIGGRAPH '93 Proceedings.* 1993, pp. 175-182.

[5]. **Cornette, William M. and Shanks, Joseph G.** Physical reasonable analytic expression for the single-scattering phase function. *Applied Optics.* 1992, Vol. 31, 16, pp. 3152-3160.

[6]. **O'Neil, Sean.** Accurate Atmospheric Scattering. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Boston : Addison-Wesley Professional, 2005.

[7]. **Hoffmann, Naty and Preetham, Arcot J.** Rendering Outdoor Light Scattering in Real Time. *Game Developer Conference 2002.*

[8]. **Thomas, Gary E. and Stamnes, Knut.** *Radiative transfer in the atmosphere and ocean.* Cambridge : Cambridge University Press, 2002.

[9]. **Goldstein, Bruce E.** *Sensation and perception.* Belmont : Wadsworth Publishing, 2009.

[10]. **Nielsen, Ralf Stokholm.** *Real Time Rendering of Atmospheric Scattering Effects for Flight Simulators.* Kongens Lyngby : Technical University of Denmark, 2003.

[11]. **Schafhitzel, Tobias, Falk, Martin and Ertl, Thomas.** Real-Time Rendering of Planets with Atmospheres. *WSCG International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision.* 2007.

[12]. **Duchaineau, Mark, et al.** ROAMing Terrain: Real-time Optimally Adapting Meshes. *IEEE Visualization '97 Proceedings.* October 1997, pp. 81-88 .

[13]. **O'Neil, Sean.** Gamasutra.com. *A Real-Time Procedural Universe, Part Two: Rendering Planetary Bodies.* [Online] [Cited: April 18, 2011.] http://www.gamasutra.com/view/feature/3042/a_realtime_procedural_universe_.php.

[14]. **Hill, David.** An efficient, hardware-accelerated, level-of-detail rendering technique for large terrains. *Master's thesis, Graduate Department of Computer Science, University of Toronto.* 2002.

[15]. **Cignoni, Paolo, et al.** BDAM – Batched Dynamic Adaptive Meshes for High Performance Terrain Visualization. *Computer Graphics Forum.* September 2003, Vol. 22, 3, pp. 505-514.

[16]. —. Planet-sized batched dynamic adaptive meshes (P-BDAM). *IEEE Visualization '03 Proceedings.* 2003.

[17]. **Clasen, Malte and Hege, Hans-Christian.** Terrain Rendering using Spherical Clipmaps. *Eurographics Symposium on Visualization 2006.* pp. 91-98.

[18]. **Asirvatham, Arul and Hoppe, Hugues.** Terrain Rendering Using GPU-Based Geometry Clipmaps. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Boston : Addison-Wesley Professional, 2005.

[19]. **Vistnes, Harald.** GPU Terrain Rendering. *Game Programming Gems 6.* New York : Charles River Media, 2006.

[20]. **Deering, Michael F., et al.** The triangle processor and normal vector shader: A VLSI system for high performance graphics. *SIGGRAPH '88 Proceedings.* 1988, Vol. 22, 4, pp. 21-30.

[21]. **Saito, Takafumi and Tokiichiro, Takahashi.** Comprehensible rendering of 3-D shapes. *SIGGRAPH '90 Proceedings.* 1990, Vol. 24, 4, pp. 197-206.

[22]. **Hargreaves, Shawn.** Deferred Shading. [Online] [Cited: April 20, 2011.] http://www.talula.demon.co.uk/DeferredShading.pdf.

[23]. **Pranckevičius, Aras.** Compact Normal Storage for small G-Buffers. [Online] [Cited: April 22, 2011.] http://aras-p.info/texts/CompactNormalStorage.html.

[24]. **Lee, Mark.** Pre-lighting in Resistance 2. *Game Developers Conference 2009.* [Online] [Cited: April 22, 2011.] http://www.insomniacgames.com/tech/articles/0409/files/GDC09_Lee_Prelighting.pdf.

[25]. **Mittring, Martin.** A bit more deferred - CryEngine3. *Triangle Game Conference 2009.* [Online] [Cited: April 22, 2011.] http://www.crytek.com/sites/default/files/A_bit_more_deferred_-_CryEngine3.ppt.

[26]. **Röttger, Stefan, Heidrich, Wolfgang and Slusallek, Philipp, Seidel, Hans-Peter.** Real-Time Generation of Continuous Levels of Detail for Height Fields. *Universität Erlangen-Nürnberg.* 1998.

[27]. **Ulrich, Thatcher.** Rendering Massive Terrains using Chunked Level of Detail Control. *SIGGRAPH '02 Course Notes.* 2002.

[28]. **Perlin, Ken.** An image synthesizer. *SIGGRAPH '85 Proceedings.* 1985, Vol. 19, 3, pp. 287-296.

[29]. **Jänich, Klaus and Kay, Leslie D.** *Vector analysis.* New York : Springer, 2001.

[30]. **Rosa, Marek.** Destructible Volumetric Terrain. *GPU Pro.* Natick : A K Peters, 2010, pp. 597-609.

[31]. **Cohen, Jonathan, Olano, Marc and Manocha, Dinesh.** Appearance-Preserving Simplification. *SIGGRAPH '98 Proceedings.* 1998, pp. 115--122.

[32]. **Lengyel, Eric.** *Mathematics for 3D game programming and computer graphics.* Second Edition. Boston : Charles River Media, 2004.

[33]. **Ericson, Christer.** *Real-Time Collision Detection.* San Francisco : Morgan Kaufmann, 2005.

[34]. **Assarsson, Ulf and Möller, Thomas.** Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools.* September 2000, Vol. 5, 1, pp. 9-22.

[35]. **Greene, Ned.** Detecting Intersection of a Rectangular Solid and a Convex Polyhedron. *Graphics gems IV.* San Diego : Academic Press Professional, 1994, pp. 74-82.

[36]. **O'Neil, Sean.** Real-Time Atmospheric Scattering. *GameDev.net.* [Online] [Cited: August 27, 2011.] http://www.gamedev.net/page/resources/_/reference/programming/special-effects/clouds/real-time-atmospheric-scattering-r2093.

[37]. **Bruneton, Eric.** Source code: Precomputed Atmospheric Scattering. [Online] [Cited: August 11, 2011.] http://www-evasion.imag.fr/Membres/Eric.Bruneton/PrecomputedAtmosphericScattering2.zip.

[38]. **Akenine-Möller, Tomas, Haines, Eric and Hoffman, Naty.** *Real-Time Rendering.* Wellesley : A. K. Peters, 2008.

[39]. **Reinhard, Erik, et al.** Photographic tone reproduction for digital images. *SIGGRAPH '02 Proceedings.* 2002, Vol. 21, 3, pp. 267-276.

[40]. **Hable, John.** Uncharted 2: HDR Lighting. *Game Developers Conference 2010.* [Online] [Cited: August 26, 2011.] http://www.gdcvault.com/play/1012459/Uncharted-2-HDR.

[41]. **Čadík, Martin, et al.** Evaluation of tone mapping operators. [Online] [Cited: August 25, 2011.] http://www.cgg.cvut.cz/~cadikm/tmo.

[42]. **Elek, Oskar.** *Rendering Parametrizable Planetary Atmospheres with Multiple Scattering in Real-Time.* Prague : Charles University, 2009.

Listings

List of Figures

List of Equations

List of Tables

List of Abbreviations

AABB	Axis aligned bounding box
GBuffer	Geometry buffer
HDR	High dynamic range
HLSL	High Level Shading Language
LOD	Level of detail
MRT	Multiple render targets

Appendix A – Planetary Terrain Shader

```
cbuffer perFrame
{
  float4x4 g_viewProj;
  float4x4 g_cube;
  float4x4 g_view;
  float3 g_cameraPos;
  float4x4 g_invTransView;
}

cbuffer once
{
  float g_vertexDistance;
  float g_invFarPlane;
}

cbuffer perObject
{
  float g_uMin;
  float g_vMin;
  float g_sizeBlock;
  bool g_culled;
}

Texture2D g_grass;
Texture2D g_stone;
Texture2D g_normal;

SamplerState PointSamplerClamp
{
  Filter = MIN_MAG_MIP_POINT;
  AddressU = CLAMP;
  AddressV = CLAMP;
};

SamplerState LinearSamplerWrap
{
  Filter = MIN_MAG_MIP_LINEAR;
  AddressU = WRAP;
  AddressV = WRAP;
};

struct VS_IN
{
  float3 pos    : POSITION;
};

struct VS_OUT
{
  float4 posH      : SV_POSITION;
  float3 posW      : POSITION;
  float3 tangent   : TANGENT0;
  float3 bitangent : TANGENT1;
  float2 texC      : TEXCOORD0;
};

struct PS_OUT
{
  float4 color  : SV_TARGET0;
  float4 normal : SV_TARGET1;
};

// input - posBlock: position within the local coordinate system of the block
// output - return value: position after transformation to uvw space
float3 getUVW(in float3 posBlock)
{
  float3 posUVW;
  posUVW.x = posBlock.x * g_sizeBlock + g_uMin;
  posUVW.y = 0.0f;
  posUVW.z = posBlock.z * g_sizeBlock + g_vMin;

  return posUVW;
}

// input - posBlock: position within the vertex buffer of the block
// output - posS: position of south neighbor in world space
// output - posN: position of north neighbor in world space
// output - posW: position of west neighbor in world space
// output - posE: position of east neighbor in world space
void GetNeighborPos(in float3 posBlock, out float3 posS, out float3 posN, out
float3 posW, out float3 posE)
{
  posS = getUVW(posBlock - float3(0.0f, 0.0f, g_vertexDistance));
  posN = getUVW(posBlock + float3(0.0f, 0.0f, g_vertexDistance));
  posW = getUVW(posBlock - float3(g_vertexDistance, 0.0f, 0.0f));
  posE = getUVW(posBlock + float3(g_vertexDistance, 0.0f, 0.0f));

  posS = normalize(mul( float4(posS, 1.0f), g_cube ).xyz);
  posN = normalize(mul( float4(posN, 1.0f), g_cube ).xyz);
  posW = normalize(mul( float4(posW, 1.0f), g_cube ).xyz);
  posE = normalize(mul( float4(posE, 1.0f), g_cube ).xyz);

  posS *= g_Rg + getElevation(posS);
  posN *= g_Rg + getElevation(posN);
  posW *= g_Rg + getElevation(posW);
  posE *= g_Rg + getElevation(posE);
}

// vertex shader
VS_OUT VS(VS_IN input)
{
  VS_OUT output;

  // transform position to uvw space
  float3 posUVW = getUVW(input.pos);

  // transform to cube and normalize to obtain position on unit sphere
  float3 posCube = mul(float4(posUVW, 1.0f), g_cube).xyz;
  float3 posUnitSphere = normalize(posCube);

  // extrude by planetary radius and an elevation factor to obtain
  // position in world space
  float elevation   = getElevation(posUnitSphere) * input.pos.y;
  float3 posWorld = posUnitSphere * (g_Rg + elevation);

  // obtain tangents
  float3 neighborS, neighborN, neighborW, neighborE;
  GetNeighborPos(input.pos, neighborS, neighborN, neighborW, neighborE);
  output.tangent   = normalize(neighborW - neighborE);
  output.bitangent = normalize(neighborS - neighborN);
  // output
  output.texC = posUVW.xz;
  output.posW   = posWorld;
  output.posH   = mul(float4(posWorld, 1.0f), g_viewProj);
  output.posH.z = output.posH.z * output.posH.w * g_invFarPlane;

  return output;
}

// controls which distance is considered as Near or Far
static const float g_textureDistNear = 100;
static const float g_textureDistFar = 1000;
// scale factors for textures
static const float g_grassTexScaleNear = 600.0f;
static const float g_grassTexScaleFar = 8.0f;
static const float g_stoneTexScaleNear = 600.0f;
static const float g_stoneTexScaleFar = 32.0f;
static const float g_normalTexScale = 1100.0f;
// value between 0 and 1 that controls the height
// where the stone texture starts to blend in
static const float g_stoneColorStart = 0.4f;
// factor that controls the softness of
// transition from grass to stone texture
static const float g_stoneColorTransition = 3.5f;

// input - texC: texture coordinates in the range [0,1]
// input - positionWorld: position in world space
// output - return value: color of surface
float3 getSurfaceColor(in float2 texC, in float3 positionWorld)
{
  float distCamToSurface  = length(g_cameraPos - positionWorld);
  float distForTextures = saturate((distCamToSurface - g_textureDistNear) /
(g_textureDistFar - g_textureDistNear));

  // lerping between near and far texture color
  float4 grassNear = g_grass.Sample(LinearSamplerWrap, texC *
                     g_grassTexScaleNear);
  float4 grassFar  = g_grass.Sample(LinearSamplerWrap, texC *
                     g_grassTexScaleFar);
  float4 grass = lerp(grassNear, grassFar, distForTextures);

  float4 stoneNear = g_stone.Sample(LinearSamplerWrap, texC *
                     g_stoneTexScaleNear);
  float4 stoneFar  = g_stone.Sample(LinearSamplerWrap, texC *
                     g_stoneTexScaleFar);
  float4 stone = lerp(stoneNear, stoneFar, distForTextures);

  // lerping between grass and stone texture
  float n = Perlin(normalize(positionWorld) * g_lowFrequencyScale);
  n = shift(n);
  n = saturate(n - g_stoneColorStart);
  n = saturate(g_stoneColorTransition * n);

  return lerp(grass.rgb, stone.rgb, n);
}

// input - tangent: tangent vector in world space
// input - bitangent: bitangent vector in world space
// output - return value: perturbed normal from normal map
float3 getNormalVector(in float3 tangent, in float3 bitangent, in float2
texC)
```

```
{
  // set up tangent-to-world matrix
  tangent   = normalize(tangent);
  bitangent = normalize(bitangent);
  bitangent = normalize(bitangent - dot(bitangent, tangent) * tangent);
  float3 normal = normalize(cross(bitangent, tangent));
  float3x3 TBN = float3x3(tangent, bitangent, normal);

  float3 normalTangent = g_normal.Sample( LinearSamplerWrap, texC *
                         g_normalTexScale).xyz;
  normal = normalize(mul(2.0f * normalTangent - 1.0f, TBN));

  return normal;
}

// pixel shader
PS_OUT PS(VS_OUT input)
{
  PS_OUT output;

  float3 color = getSurfaceColor(input.texC, input.posW);
  float3 normal = getNormalVector(input.tangent, input.bitangent,input.texC);

  output.color.rgb  = color;
  output.color.a    = 1.0f;
  output.normal.xyz = 0.5f * normal + 0.5f;
  output.normal.w   = AVERAGE_GROUND_REFLECTANCE / M_PI;

  return output;
}

technique10 RenderPlanet
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS() ) );
    }
}
```

Appendix B – Sun Shader

```
cbuffer cbSun
{
  float3   g_cameraPos;
  float3   g_sunDirection;
  float    g_sunDistance;
  float4x4 g_viewProj;
  float       g_invFarPlane;
}

Texture2D g_texture;

struct VS_IN
{
  float3 posL    : POSITION;
  float2 texC    : TEXCOORD;
};

struct VS_OUT
{
  float4 posH    : SV_POSITION;
  float2 texC    : TEXCOORD;
};

struct PS_OUT
{
  float4 color    : SV_TARGET0;
  float4 normal   : SV_TARGET1;
};


// output - rotationMatrix: matrix that lets the quadrilateral face the
camera at all times
void generateRotationMatrix(out float4x4 rotationMatrix)
{
  float3 lookAtDirection = normalize(-g_sunDirection);

  float3 baseVector0, baseVector1, baseVector2;

  // first base vector equals lookAtDirection
  baseVector0 = lookAtDirection;

  // second base vector found by crossing first base vecotr with
  // cardinal basis vector corresponding to element with least magnitude
  float3 crossVector = float3(1, 0, 0);
  float absX = abs(lookAtDirection.x);
  float absY = abs(lookAtDirection.y);
  float absZ = abs(lookAtDirection.z);

  if((absY <= absX) && (absY <= absZ))
    crossVector = float3(0.0f, 1.0f, 0.0f);
  else if((absZ <= absX) && (absZ <= absY))
    crossVector = float3(0.0f, 0.0f, 1.0f);

  baseVector1 = normalize(cross(baseVector0, crossVector));

  // third base vector equals crossing first and second base vector
  baseVector2 = normalize(cross(baseVector0, baseVector1));
  rotationMatrix[0] = float4(baseVector2, 0.0f);
  rotationMatrix[1] = float4(baseVector1, 0.0f);
  rotationMatrix[2] = float4(baseVector0, 0.0f);
  rotationMatrix[3] = float4(0.0f, 0.0f, 0.0f, 1.0f);
}

// input - sunPosition: postion of the sun in world space
// output - translationMatrix: matrix that translates quadrilateral to world
space position
void generateTranslationMatrix(in float3 sunPosition, out float4x4
translationMatrix)
{
  translationMatrix[0] = float4(1.0f, 0.0f, 0.0f, 0.0f);
  translationMatrix[1] = float4(0.0f, 1.0f, 0.0f, 0.0f);
  translationMatrix[2] = float4(0.0f, 0.0f, 1.0f, 0.0f);
  translationMatrix[3] = float4(sunPosition, 1.0f);
}

// vertex shader
VS_OUT VS_SPHERICAL_BILLBOARD(VS_IN input)
{
  VS_OUT output;

  // calculates position of sun relative to camera position
  float3 sunPosition = g_cameraPos + (g_sunDirection * g_sunDistance);

  float4x4 rotationMatrix, translationMatrix;
  generateRotationMatrix(rotationMatrix);
  generateTranslationMatrix(sunPosition, translationMatrix);

  float4x4 worldMatrix = mul(rotationMatrix, translationMatrix);
  float4x4 worldViewProjMatrix = mul(worldMatrix, g_viewProj);
  output.posH  = mul( float4(input.posL,1.0f), worldViewProjMatrix );
  output.posH.z = output.posH.z * output.posH.w * g_invFarPlane;

  output.texC  = input.texC;

  return output;
}

// pixel shader
PS_OUT PS_SUN(VS_OUT input)
{
  PS_OUT output;

  // sample texture with alpha channel
  output.color = g_texture.Sample( LinearSamplerClamp, input.texC );

  // set normal vector to sun direction vector
  float3 normal = g_sunDirection;
  output.normal.xyz = 0.5f * normal + 0.5f;

  // set sun reflectance value to 1.0f
  output.normal.w = 1.0f;

  return output;
}

technique10 RenderBillboardSun
{
    pass P0
```

```
        {
            SetVertexShader( CompileShader( vs_4_0, VS_SPHERICAL_BILLBOARD() ) );
            SetGeometryShader( NULL );
            SetPixelShader( CompileShader( ps_4_0, PS_SUN() ) );
        }
    }
```

Appendix C – Atmospheric Scattering Shader

```
static const float EPSILON_ATMOSPHERE = 0.002f;
static const float EPSILON_INSCATTER = 0.004f;

Texture2D g_depth;
Texture2D g_color;
Texture2D g_normal;
Texture2D g_texIrradiance;
Texture3D g_texInscatter;

float3 g_cameraPos;
float3 g_sunVector;
float4x4 g_cameraWorld;
float4 g_frustumFar[4];
float4 g_frustumNear[4];
float sunIntensity = 30.0f;

struct VS_IN
{
  float3 posL    : POSITION;
  float2 texC    : TEXCOORD0;
  uint   index   : TEXCOORD1;
};

struct VS_OUT
{
  float4 posH         : SV_POSITION;
  float2 texC           : TEXCOORD0;
  float3 nearToFar    : TEXCOORD2;
  float3 cameraToNear : TEXCOORD3;
};

SamplerState PointSamplerClamp
{
  Filter = MIN_MAG_MIP_POINT;
  AddressU = Clamp;
  AddressV = Clamp;
};


// vertex shader
VS_OUT VS(VS_IN input)
{
  VS_OUT output;

  output.posH   = float4(input.posL,1.0f);
  output.texC   = input.texC;

  float3 frustumFarWorld = mul(float4(g_frustumFar[input.index].xyz, 1.0f),
                           g_cameraWorld).xyz;
  float3 frustumNearWorld = mul(float4(g_frustumNear[input.index].xyz, 1.0f),
                             g_cameraWorld).xyz;

  output.cameraToNear = frustumNearWorld - g_cameraPos;
  output.nearToFar = frustumFarWorld - frustumNearWorld;

  return output;
}
```

```
// input - d: view ray in world space
// output - offset: distance to atmosphere or 0 if within atmosphere
// output - maxPathLength: distance traversed within atmosphere
// output - return value: intersection occurred true/false
bool intersectAtmosphere(in float3 d, out float offset, out float
maxPathLength)
{
  offset = 0.0f;
  maxPathLength = 0.0f;

  // vector from ray origin to center of the sphere
  float3 l = -g_cameraPos;
  float l2 = dot(l,l);
  float s = dot(l,d);
  // adjust top atmosphere boundary by small epsilon to prevent artifacts
  float r = g_Rt - EPSILON_ATMOSPHERE;
  float r2 = r*r;

  if(l2 <= r2)
  {
    // ray origin inside sphere, hit is ensured
    float m2 = l2 - (s * s);
    float q = sqrt(r2 - m2);
    maxPathLength = s + q;

    return true;
  }
  else if(s >= 0)
  {
    // ray starts outside in front of sphere, hit is possible
    float m2 = l2 - (s * s);

    if(m2 <= r2)
    {
      // ray hits atmosphere definitely
       float q = sqrt(r2 - m2);
       offset = s - q;
       maxPathLength = (s + q) - offset;

       return true;
    }
  }

  return false;
}

// input - surfacePos: reconstructed position of current pixel
// input - viewDir: view ray in world space
// input/output - attenuation: extinction factor along view path
// input/output - irradianceFactor: surface hit within atmosphere 1.0f
// otherwise 0.0f
// output - return value: total in-scattered light
float3 GetInscatteredLight(in float3 surfacePos, in float3 viewDir, in out
float3 attenuation, in out float irradianceFactor)
{
  float3 inscatteredLight = float3(0.0f, 0.0f, 0.0f);

  float offset;
  float maxPathLength;
  if(intersectAtmosphere(viewDir, offset, maxPathLength))
  {
    float pathLength = distance(g_cameraPos, surfacePos);
    // check if object occludes atmosphere
    if(pathLength > offset)
    {
      // offsetting camera
      float3 startPos = g_cameraPos + offset * viewDir;
      float startPosHeight = length(startPos);
      pathLength -= offset;

      // starting position of path is now ensured to be inside atmosphere
      // was either originally there or has been moved to top boundary
      float muStartPos = dot(startPos, viewDir) / startPosHeight;
      float nuStartPos = dot(viewDir, g_sunVector);
      float musStartPos = dot(startPos, g_sunVector) / startPosHeight;

      // in-scattering for infinite ray (light in-scattered when
       // no surface hit or object behind atmosphere)
      float4 inscatter = max(texture4D(g_texInscatter, startPosHeight,
                         muStartPos, musStartPos, nuStartPos), 0.0f);

      float surfacePosHeight = length(surfacePos);
      float musEndPos = dot(surfacePos, g_sunVector) / surfacePosHeight;


      // check if object hit is inside atmosphere
      if(pathLength < maxPathLength)
      {
        // reduce total in-scattered light when surface hit
         // within atmosphere
        // fix described in chapter 5.1.1
        attenuation = analyticTransmittance(startPosHeight, muStartPos,
                      pathLength);

        float muEndPos = dot(surfacePos, viewDir) / surfacePosHeight;
        float4 inscatterSurface =
                             texture4D(g_texInscatter, surfacePosHeight,
                             muEndPos, musEndPos, nuStartPos);
        inscatter = max(inscatter-attenuation.rgbr*inscatterSurface, 0.0f);
        irradianceFactor = 1.0f;
      }
      else
      {
        // retrieve extinction factor for inifinte ray
        // fix described in chapter 5.1.1
        attenuation = analyticTransmittance(startPosHeight, muStartPos,
                      pathLength);
      }

      // avoids imprecision problems near horizon by interpolating between
       // two points above and below horizon
      // fix described in chapter 5.1.2
      float muHorizon = -sqrt(1.0 - (g_Rg / startPosHeight) * (g_Rg /
                        startPosHeight));
      if (abs(muStartPos - muHorizon) < EPSILON_INSCATTER)
      {
        float mu = muHorizon - EPSILON_INSCATTER;
        float samplePosHeight = sqrt(startPosHeight*startPosHeight
                                +pathLength*pathLength+2.0f*startPosHeight*
                                pathLength*mu);

        float muSamplePos = (startPosHeight * mu + pathLength)/
                             samplePosHeight;
        float4 inScatter0 = texture4D(g_texInscatter, startPosHeight, mu,
                                      musStartPos, nuStartPos);
        float4 inScatter1 = texture4D(g_texInscatter, samplePosHeight,
                                      muSamplePos, musEndPos, nuStartPos);
        float4 inScatterA = max(inScatter0-attenuation.rgbr*inScatter1,0.0);

        mu = muHorizon + EPSILON_INSCATTER;
        samplePosHeight = sqrt(startPosHeight*startPosHeight
                          +pathLength*pathLength+2.0f*
                          startPosHeight*pathLength*mu);
        muSamplePos = (startPosHeight * mu + pathLength) / samplePosHeight;

        inScatter0 = texture4D(g_texInscatter, startPosHeight, mu,
                               musStartPos, nuStartPos);
        inScatter1 = texture4D(g_texInscatter, samplePosHeight, muSamplePos,
                               musEndPos, nuStartPos);
        float4 inScatterB = max(inScatter0 - attenuation.rgbr * inScatter1,
                                      0.0f);
       float t = ((muStartPos - muHorizon) + EPSILON_INSCATTER) /
                  (2.0 * EPSILON_INSCATTER);

        inscatter = lerp(inScatterA, inScatterB, t);
      }

      // avoids imprecision problems in Mie scattering when sun is below
       //horizon
      // fix described in chapter 5.1.3
      inscatter.w *= smoothstep(0.00, 0.02, musStartPos);
      float phaseR = phaseFunctionR(nuStartPos);
      float phaseM = phaseFunctionM(nuStartPos);
      inscatteredLight = max(inscatter.rgb * phaseR + getMie(inscatter)*
                         phaseM, 0.0f);
      inscatteredLight *= sunIntensity;
    }
  }

  return inscatteredLight;
}

// input - surfacePos: reconstructed position of current pixel
// input - texC: texture coordinates
// input - attenuation: extinction factor along view path
// input - irradianceFactor: surface hit within atmosphere 1.0f
// otherwise 0.0f
// output - return value: total reflected light + direct sunlight
float3 GetReflectedLight(in float3 surfacePos, in float2 texC, in float3
attenuation, in float irradianceFactor)
{
  // read contents of GBuffer
  float4 normalData = g_normal.SampleLevel(PointSamplerClamp, texC, 0);
  float3 surfaceColor = g_color.SampleLevel(PointSamplerClamp, texC, 0).rgb;

  // decode normal and determine intensity of refected light at
  // surface postiion
  float3 normal = 2.0f * normalData.xyz - 1.0f;
  float lightIntensity = sunIntensity * normalData.w;
  float lightScale = max(dot(normal, g_sunVector), 0.0f);

  // irradiance at surface position due to sky light
  float surfacePosHeight  = length(surfacePos);
  float musSurfacePos = dot(surfacePos, g_sunVector) / surfacePosHeight;
  float3 irradianceSurface = irradiance(g_texIrradiance, surfacePosHeight,
                             musSurfacePos) * irradianceFactor;

  // attenuate direct sun light on its path from top of atmosphere to
  // surface position
  float3 attenuationSunLight = transmittance(surfacePosHeight,musSurfacePos);
  float3 reflectedLight = surfaceColor * (lightScale * attenuationSunLight +
                          irradianceSurface) * lightIntensity;
  // attenuate again on path from surface position to camera
  reflectedLight *= attenuation;

  return reflectedLight;
}

// pixel shader
float4 PS_PLANET_DEFERRED(VS_OUT input) : SV_TARGET0
{
  // reconstructing world space postion by interpolation
  float depthVal = g_depth.SampleLevel( PointSamplerClamp, input.texC, 0 ).r;
  float3 surfacePos = g_cameraPos + input.cameraToNear + depthVal *
                      input.nearToFar;

  // obtaining the view direction vector
  float3 viewDir = normalize(input.nearToFar);

  float3 attenuation        = float3(1.0f, 1.0f, 1.0f);
  float irradianceFactor = 0.0f;

  float3 inscatteredLight = GetInscatteredLight(surfacePos, viewDir,
                            attenuation, irradianceFactor);
  float3 reflectedLight = GetReflectedLight(surfacePos, input.texC,
                          attenuation, irradianceFactor);

  return float4(HDR(reflectedLight + inscatteredLight), 1.0f);
}


technique10 RenderPlanetDeferred
{
    pass P0
    {
        SetVertexShader( CompileShader( vs_4_0, VS() ) );
        SetGeometryShader( NULL );
        SetPixelShader( CompileShader( ps_4_0, PS_PLANET_DEFERRED() ) );
    }
}
```