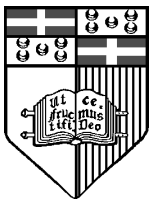
Synthesising physically correct skies for interactive rendering

Juraj Papp

Supervisor(s): Dr Sandro Spina, Dr Keith Bugeja



Institute of Digital Games

University of Malta

June 2018

Submitted in partial fulfilment of the requirements for the degree of M.Sc. in Digital Games.

Abstract:

In computer graphics, physically based rendering is concerned with rendering images indistinguishable from pictures of the real world. However, physically based approaches to rendering are traditionally implemented using numerical methods which limit their use to offline solutions due to their computational cost. In this work we explore rendering of physically correct sky models together with participating media such as clouds, and present a model suitable for interactive rendering on a wide spectrum of devices. This model supports a full day-night cycle, positionally accurate astronomical bodies, volumetric animated clouds, rainbows and image-based lighting to correctly illuminate the scene. The implementation of our model has been packaged into an extensible Java library with an accessible API: SevenSky. The model implementation is evaluated in terms of performance, visual quality and correctness, while the applicability of the API is validated through two use cases. The results show that SevenSky can be used to generate photorealistic skies at interactive rates of over 60 Hz even on modest graphics hardware such as integrated GPUs.

Acknowledgements

I would like to express my gratitude and appreciation to my supervisors Dr Sandro Spina and Dr Keith Bugeja for their continuous guidance, support and motivation. My gratitude goes also to my family for their support and encouragement.

Contents

List of Figures

1 Introduction

The state of the art in computer graphics is striving for photorealism and physical correctness. Photorealism pushes for synthesised images that are plausible renditions of the world around us. On the other hand, physical correctness is not satisfied with merely emulating how an observer perceives the world around them; rather, it takes the path of modelling and simulating the physical properties of light, materials and surfaces to the extent of generating imagery that is indistinguishable from real world photos.

The discipline that generates physically correct images is known as physically based rendering and revolves around the rendering equation, an intractable integral equation first introduced, simultaneously and independently, by Kajiya [12] and Immel [13]. The rendering equation employs radiometric quantities to model steady-state light distribution in a scene in terms of radiance. Due to its intractability, it is typically evaluated using numerical techniques such as Monte Carlo or finite element approaches, making it computationally intensive and a challenging proposition for interactive rendering systems to implement. Nonetheless, there is an ever-increasing number of rendering tools and middleware with interactive disposition that are integrating physically based techniques into their synthesis pipelines [14, 15, 16] to generate physically correct output, motivated by the level of photorealism achievable through this approach.

There is a multitude of layers of the physical world that may be modelled in a physically correct manner; for instance, even though the rendering techniques, material and models employed may be physically correct, the environment might not. In interactive settings, these concerns remain as valid as ever; applications that train people through simulation, for instance, might require physical correctness on multiple levels, more than is typically offered by popular 3d application programming interfaces (APIs) or game engines. To build on our example, software that trains users for navigation might require a correct simulation of the night sky, accurate positions of the constellations, stars and other astronomical bodies. More so, a flight simulator would want to simulate accurate atmospheric visualisation, down to proper light propagation, scattering and attenuation, time of day and, sun and moon positions, to correctly predict different visibility scenarios a pilot might encounter on duty.

The applications for interactive physically correct sky visualisation are innumerable: disciplines ranging from engineering to archaeology, from architectural walk-throughs to the interactive relighting and manipulation of sites and artefacts, from user training via virtual reality and simulation to entertainment and video games that require realism and accurate physically based environments, the ability to efficiently render a physically correct sky with phenomena like participating media would be invaluable. Furthermore, researchers and software developers from these fields in particular would greatly benefit from a lightweight API encapsulating this functionality, designed for a widespread programming technology.

1.1 Aims and Objectives

The aim of this work is to design an efficient physically correct, location-dependent sky model with day-night cycle that includes participating media effects such as clouds and rainbows for interactive rendering. Moreover, in order to reach a wide spectrum of researchers, a further goal of this work is that of packaging the model into a comprehensive, extensible and easy to use library.

In order to accomplish the goals of this dissertation, the following objectives have been set:

- Research and design:
  - an atmosphere model for light propagation that supports attenuation and scattering;
  - a sky model with day-night cycle complete with accurate positions for the main astronomical bodies (e.g. planets, natural satellites and constellations);
  - altitude-dependent participating media (e.g. clouds).
- Develop and test a portable software implementation for the model that runs on a widespread spectrum of hardware, from mobile phones to high-end desktops.
- Package software implementation into an extensible library with an easy to use API (SevenSky).

1.2 Dissertation Outline

This dissertation is comprised of the following chapters:

Background This chapter provides an overview of the concepts employed in this work, including dates and celestial coordinate systems, measurement and calculation of radiometric quantities, blackbody radiation, participating media and image synthesis.

Literature review This chapter reviews the state of the art in photorealistic rendering of skies, clouds and other atmospheric phenomena.

Sky Model This chapter describes the design and implementation of the presented sky model, including the day-night cycle and astronomical body placements. The model is tested and validated.

Cloud Model This chapter introduces a method for rendering volumetric clouds in real-time, together with a novel technique for realistically animating cloud motion. Clouds are modelled as participating media with a density function obtained from a weather map. The model supports soft shadows, cast by the clouds and the visualisation of rainbows. The model is tested and validated under different conditions and system performance is measured.

SevenSky Library This chapter introduces the SevenSky library; it also gives an overview into its design and implementation together with a number of use cases for its application.

Conclusion This chapter concludes the dissertation by summarising the main contributions, outlining the limitations and proposing avenues for future work.

2 Background

This chapter provides an overview of the fundamental concepts related to this work. In order to synthesize a physically correct sky, laws of physics and real world measurements are used in the development of our sky model. These include astronomical bodies and the Celestial coordinate systems associated with them. Following on this, time standards are then presented which are essential in determining the positions of Celestial bodies given a particular time. In order to be able to visualise these bodies a measure of brightness using radiometric quantities needs to be computed for the main bodies. Finally, an overview of computer graphics including a brief description of the rasterisation rendering pipeline is given. This is essential to understand how the different sky phenomena (e.g. clouds) are rendered.

2.1 Astronomical Bodies

VSOP87 (Variations Sculaires des Orbites Plantaires) [17] presents a Mathematical model describing the positions of planets. In SevenSky, the VSOP87 model is chosen for its accuracy and simplicity in implementation, which provides functions to calculate the position of major planets in the Cartesian and spherical coordinate systems and their implementation in several programming languages including Java. VSOP87 is regarded as the most popular source for planetary calculations and is used by several astronomy programs. In the following section the coordinate systems within which these positions are computed is described.

2.2 Coordinate Systems

Coordinate systems are systems that use numbers, referred to as coordinates, to describe points in space with respect to an origin. There are different types of coordinate systems such as Cartesian, polar, spherical and others. In addition, there are coordinate systems specifically designed to be used for celestial bodies where position can

be expressed in horizontal, equatorial, ecliptic, galactic or super-galactic coordinate systems.

2.2.1 Ecliptic Coordinate System

Ecliptic is a path on the Celestial Sphere that the Sun appears to follow over a timespan of a year (Fig. 2.1). It also refers to the plane which contains this path. The plane of orbit of the main planets in the solar system is close to the plane of the ecliptic. The origin of the coordinate system is either the Sun or the Earth. In the first case, the coordinate system is referred to as heliocentric and in the latter geocentric. The primary direction is towards the vernal equinox.

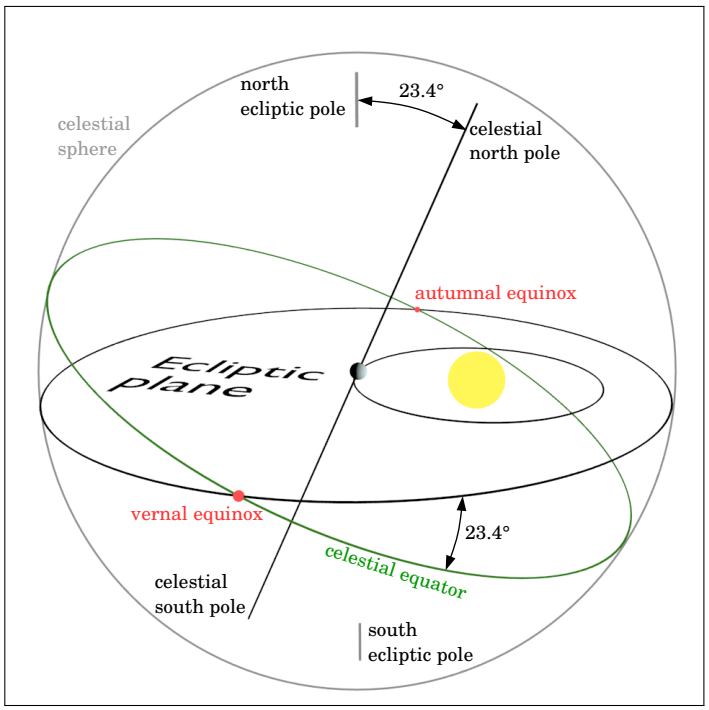


Figure 2.1: Celestial sphere and the plane of the ecliptic. The Earth is shown orbiting the Sun in the ecliptic plane.

The coordinates of the system may be spherical or rectangular. In the spherical coordinate system the position is represented by ecliptic longitude and latitude. The longitude measures angular distance along the ecliptic from the primary direction. The latitude measures the angular distance from the ecliptic.

2.2.2 Equatorial Coordinate System

The equatorial coordinate system uses the plane of the Earth's equator as its fundamental plane. Its primary direction is towards the vernal equinox. The system's origin is either the Sun or the Earth. Its coordinates can be spherical or rectangular. In the first case, the coordinates are represented by the right ascension(or hour angle) and declination. The right ascension measures the angular distance along the celestial equator, whereas the declination measures the angular distance from the celestial equator.

2.2.3 Horizontal Coordinate System

The horizontal coordinate system uses the observer's horizon as its fundamental plane. The spherical coordinates are represented by altitude (or elevation) and azimuth. The azimuth measures the angular distance along the horizon and altitude measures the angular distance from the horizon.

2.2.4 Ecliptic Coordinates of the Sun

The mathematical model VSOP87 can be used to compute the heliocentric ecliptic coordinates of planets in our solar system. Equations for spherical and rectangular coordinates are provided by an implementation of VSOP87. The spherical coordinate system is used in this work. Since the model is heliocentric, in order to calculate the position of the Sun the position of the Earth is calculated instead. An implementation of VSOP87 defines three functions $E_{lon}(t), E_{lat}(t), E_{rad}(t)$ which accept a parameter t which is a Julian time (see Section 2.3.2) in millennia. The three functions return longitude, latitude and distance between the Sun and the Earth respectively. The longitude and latitude is expressed in ecliptic coordinates and the distance is expressed in astronomical units. One astronomical unit is equal to the average distance between the Sun and the Earth and is defined at 149,597,870,700m.

2.2.5 Converting between Ecliptic and Equatorial Coordinates

Function $C_{EclipEquat}(s, lon, lat, e)$ is used to convert between ecliptic and equatorial coordinate systems. The first parameter s is set to 1 to convert ecliptic coordinates to equatorial and to -1 to convert equatorial to ecliptic coordinates. lon, lat is the longitude and latitude to be converted and finally e is the obliquity of the ecliptic, also known as the axial tilt. It is the angle between Earth's rotational axis and the ecliptic plane. This value is not constant and on average it is equal to 23.4°. A more accurate value is computed by using the current Julian date. This calculation has not been included due to its complexity. The function $C_{EclipEquat}(s, lon, lat, e)$ is calculated as follows:

$$C_{EclipEquat}(s, lon, lat, e) = \begin{bmatrix} lon_a & lat_a \end{bmatrix} \tag{2.1}$$

where,

$x = cos(lon)$

$y = cos(e)sin(lon) - tan(lat)sin(e)s$

$lon_a = 2\pi \lfloor atan2(y, x)/(2\pi) \rfloor$

$lat_a = asin(sin(lat)cos(e) + cos(lat)sin(e)sin(lon)s).$

2.3 Time Standards

Time standards refer to specifications for measuring time and date. Time can be determined by the rotation of the Earth, the position of the Sun, position of stars or by an atomic clock. The next sections describe different type standards used in the process of computing the position of celestial bodies.

2.3.1 Universal Time (UT1)

Universal Time (UT1), previously known as Greenwich Mean Time (GMT) measures the rotation of the Earth with respect to the position of the Sun. Universal Time depends on the Earth's rotation and as the Earth's rotation varies so does the Universal Time vary and is thus not uniform.

2.3.2 Julian Date and UT Conversion

Julian day is represented by a number of days that have passed since the beginning of the Julian Period on the 24 November 4714 BC in the proleptic Gregorian calendar [18]. Julian days are widely used in astronomical calculations. Let $jdtt$ represent the Julian Day number which corresponds to a given day and time and t represents Julian millennia starting at the year 2000. Julian day and time can be calculated by using Universal Time:

$$t = \frac{jdtt - 2451545}{365250} \tag{2.2}$$

$$jdtt = JDN(UT_{day}, UT_{month}, UT_{year}) + JDTT(UT_{hour}, UT_{minute}, UT_{seconds}) \tag{2.3}$$

$$JDTT(h, m, s) = \frac{h - 12.0}{24} + \frac{m}{1440} + \frac{s}{86400} \tag{2.4}$$

$$JDN(day, month, year) = day + \left\lfloor \frac{153 * m + 2}{5} \right\rfloor + 365.0 * y + \left\lfloor \frac{y}{4} \right\rfloor - \left\lfloor \frac{y}{100} \right\rfloor + \left\lfloor \frac{y}{400} \right\rfloor - 32045.0 \tag{2.5}$$

where,

$$a = \frac{14 - month}{12}$$
$$y = year + 4800 - a$$
$$m = month + 12 * a - 3.$$

2.3.3 Greenwich Mean Sidereal Time

Sidereal time (Fig. 2.2) is useful for locating celestial objects. It is the measure of the Earth's rotation against distant stars. A sidereal day is approximately 23h 56m 4s long. Given a Julian date $jdtt$, Julian millennia is equal to $t = (jdtt - 2451545)/365250$. Sidereal time $GMST(jdtt)$ is calculated as follows:

$$GMST(jdtt) = 7.27220521664e^{-5}(86400.0(jdtt\%1.0) + 24110.54841 - 43200 + (8640184.812866 + (0.093104 - 0.0000062 * t)t)t)t). \tag{2.6}$$

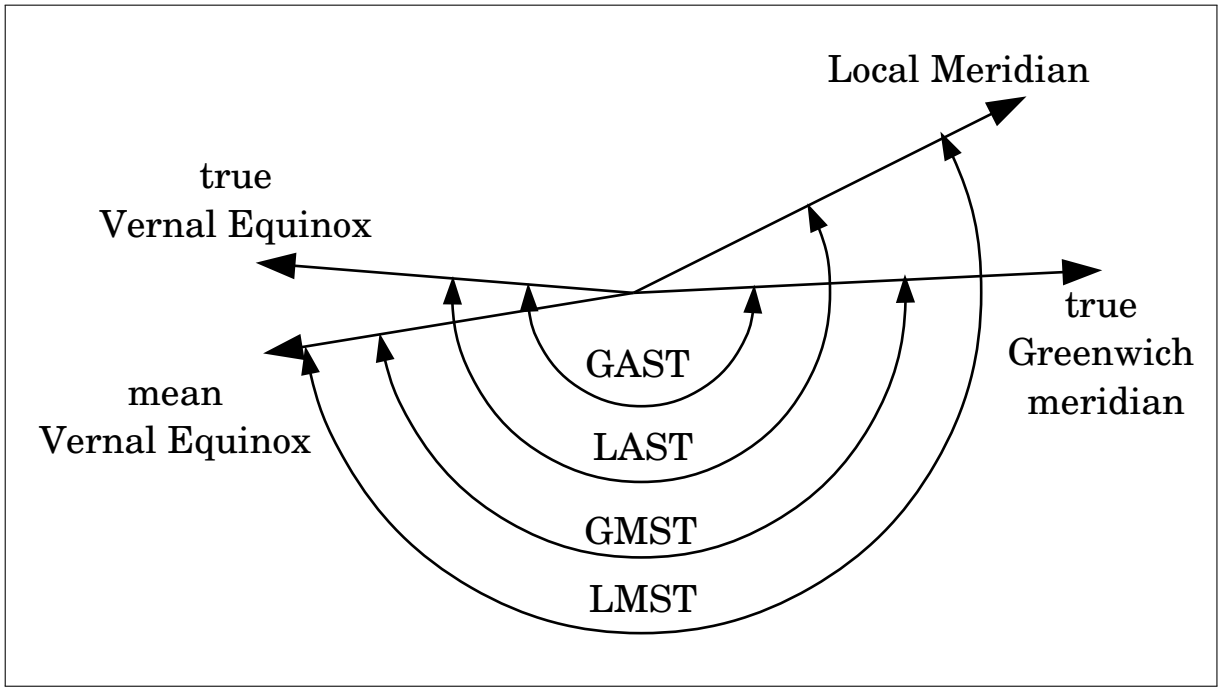


Figure 2.2: Different Sidereal times: GAST, LAST, GMST and LMST.

2.3.4 Greenwich Apparent Sidereal Time

Greenwich Apparent Sidereal Time (GAST) is Greenwich Mean Sidereal Time that is corrected for nutation in right ascension. A nutation is a change in the direction of the Earth's axis of rotation. The Equation of equinoxes (EE) [19] accounts for this difference and is factored in the calculation as follows:

$$GAST(jdtt) = GMST(jdtt) + EE. \tag{2.7}$$

2.3.5 Local Mean Sidereal Time

Local Mean Sidereal Time (LAST) is equal to GMST plus observer's longitude (lon_{obs}) measured positive to the east of Greenwich

$$LMST(jdtt) = GMST(jdtt) + lon_{obs}. \tag{2.8}$$

2.3.6 Local Apparent Sidereal Time

Local Apparent Sidereal Time is equal to GAST plus observer's longitude (lon_{obs}) measured positive to the east of Greenwich

$$LAST(jdtt) = GAST(jdtt) + lon_{obs}. \tag{2.9}$$

2.3.7 Hour Angle

The hour angle $HA(jdtt, ra)$ refers to one of the coordinates used in the equatorial coordinate system used to give the direction of a celestial body and is calculated as follows:

$$HA(jdtt, ra) = GAST(jdtt) - ra. \tag{2.10}$$

2.4 Positions of the Major Astronomical Bodies

Given time t in Julian millennia and observer coordinates lon_{obs}, lat_{obs} on the Earth, the ecliptic longitude and latitude of a planet can be computed with $P_{lon}(t), P_{lat}(t)$, where P are functions specific to a chosen planet. VSOP87 supports computing coordinates for eight planets in the Solar system. The equatorial right ascension (ra) and declination of the planet (dec) is computed using the function $C_{EclipEquat}(1, E_{lon}(t), E_{lat}(t), e)$. The hour angle is equal to $ha = HA(jdtt, ra)$. Finally, the horizontal coordinates are defined by azimuth and altitude and are computed with function $O(ra, dec, ha)$:

$$O(ra, dec, ha) = \begin{bmatrix} azim & alt \end{bmatrix} \tag{2.11}$$

where:

$$
\begin{aligned}
x &= cos(ha)cos(dec) \\
y &= sin(ha)cos(dec) \\
z &= sin(dec) \\
x_h &= xsin(lat_{obs}) - zcos(lat_{obs}) \\
y_h &= y \\
z_h &= xcos(lat_{obs}) + zsin(lat_{obs}) \\
azim &= atan2(y_h, x_h) + \pi \\
alt &= asin(z_h).
\end{aligned}
$$

2.4.1 Position of the Sun

Given time t in Julian millennia, observer coordinates lon_{obs}, lat_{obs} on the Earth, the ecliptic longitude and latitude of the Earth can be computed with $E_{lon}(t), E_{lat}(t)$. The equatorial right ascension and declination of the Sun ra, dec is computed with $C_{EclipEquat}(1, E_{lon}(t) + \pi, E_{lat}(t), e)$. The hour angle is equal to $ha = HA(jdtt, ra)$. Finally, the horizontal coordinates of the Sun defined by azimuth and altitude is computed with function $O(ra, dec, ha)$:

2.4.2 Position of the Moon

Similarly to calculating the position of the planets, the position of the Moon is calculated with the functions $M_{lon}(t), M_{lat}(t), M_{rad}(t)$. These function calculate the geocentric ecliptic coordinates of the Moon. Afterwards, the equatorial right ascension(ra) and declination(dec) of the Moon is computed with $C_{EclipEquat}(1, M_{lon}(t), M_{lat}(t), e)$ function. Finally the azimuth and altitude is obtained with $O(ra, dec, HA(jdtt, ra))$.

2.5 Radiometric Quantities

In the previous section, the calculation required to determine the position of astronomical bodies was described. Since the bodies and stars differ in brightness in order to objectively describe the amount of light an object emits, radiometric quantities are used. Radiance $L_{e,\Omega}$ measures radiant energy emitted or received by a surface per unit

solid angle (Fig. 2.3) per unit projected area. It is measured in $Wsr^{-1}m^{-2}$. Spectral radiance refers to the measured radiant energy of a given wavelength and its units are $Wsr^{-1}m^{-2}nm^{-1}$ or $Wsr^{-1}m^{-2}Hz^{-1}$

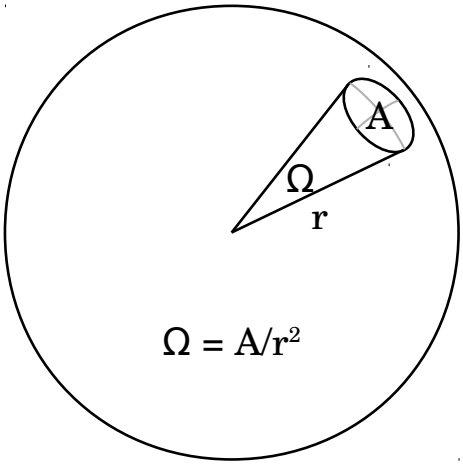


Figure 2.3: Solid angle Ω, A is the spherical surface area, r is sphere radius.

Irradiance E_e measures the radiant energy received by a surface per unit area (Fig. 2.4). Its units are Wm^{-2}. Spectral irradiance measures the radiant energy received by a surface per unit area per wavelength. Its units are $Wm^{-2}nm^{-1}$ or $Wm^{-2}Hz^{-1}$

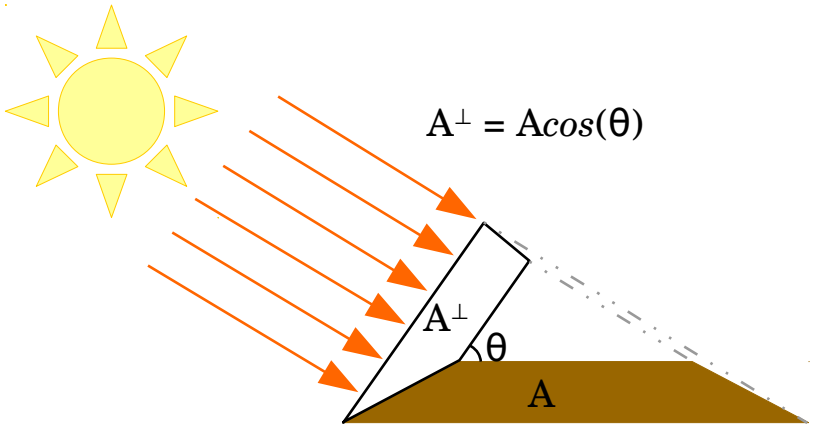


Figure 2.4: Irradiance E_e for an area A is calculated by first calculating the area $A^{\perp}$, which is perpendicular to the light direction.

2.6 Black Body Radiation

A black body is an ideal material which absorbs all electromagnetic radiation. At a constant temperature a black body emits black body radiation, emitted according to Planck's law. The spectrum of such radiation is determined only by its temperature. The radiation emitted from stars can be modelled as black body radiation. Figure 2.5 shows the spectrum of the Sun with and without atmospheric absorption and the spectrum of a black body with a temperature of 5778K.

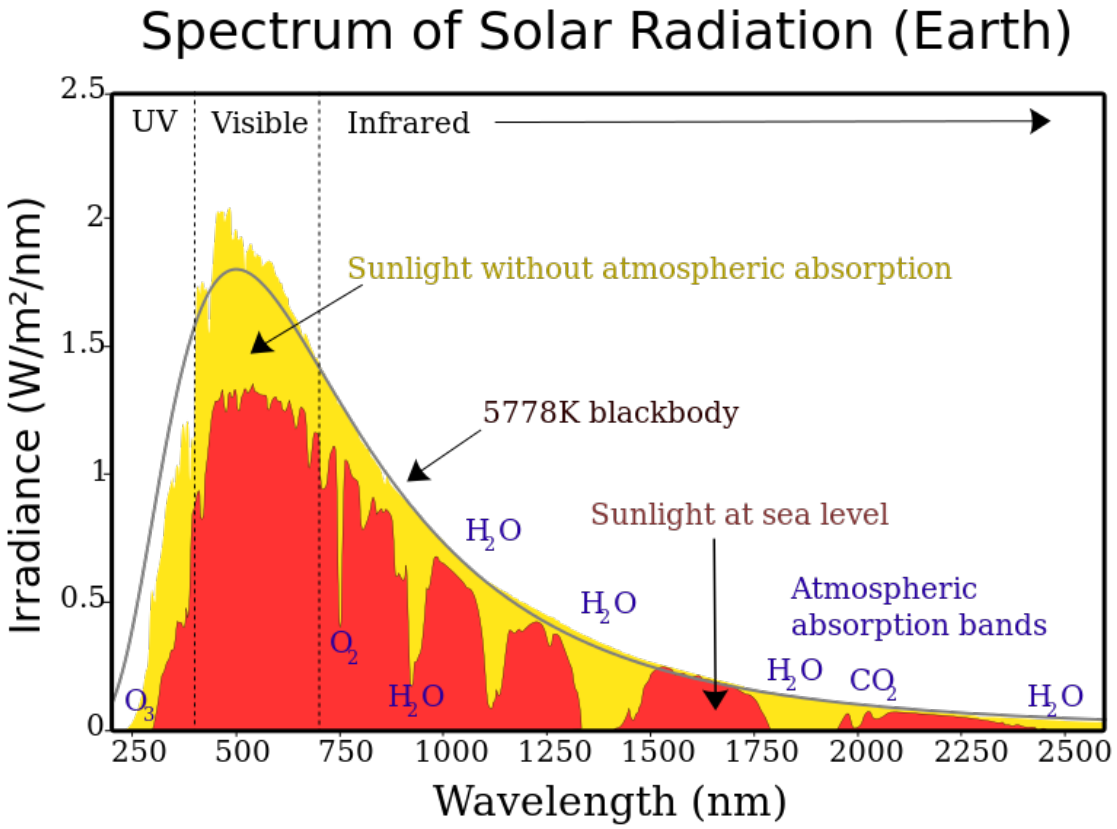


Figure 2.5: Solar spectrum and blackbody radiation emitted by a black body with temperature of 5778K (Credit: Nick84 [CC BY-SA 3.0], via Wikimedia Commons).

2.7 Participating Media

Th term participating media is used to describe volumes of space filled with particles which can interact with light. Such particles may be molecules, dust, water droplets

or other. As light travels through such medium, it may interact with these particles (Fig. 2.6). Some light may be absorbed, scattered into different directions or emitted by a particle. Absorption coefficient β_a and scattering coefficient β_s describe how much light is absorbed and scattered. Examples of physical phenomena which can be modelled as participating media include the sky, clouds and fog.

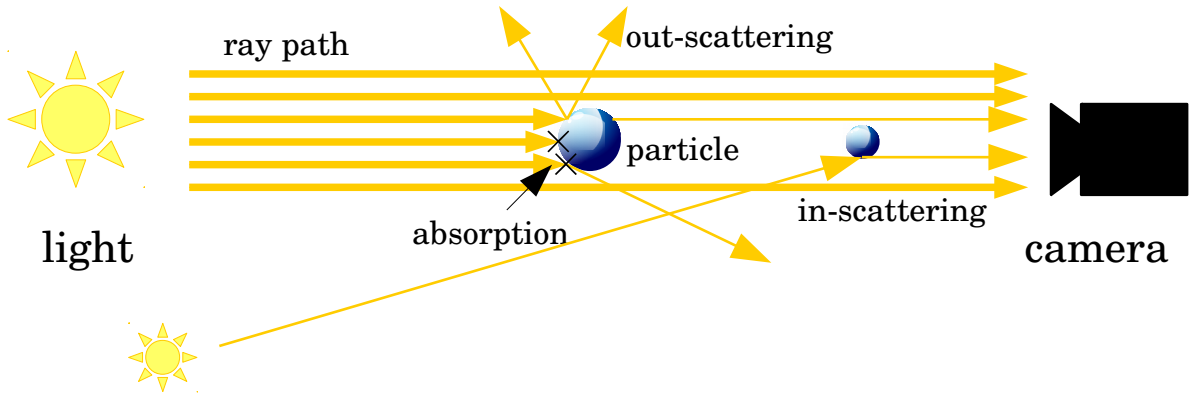


Figure 2.6: Absorption and scattering.

2.7.1 Absorption, Transmittance and Optical Depth

As light travels though a medium a percentage of it may be absorbed. This percentage is called transmittance. Given a volume with uniform distribution of particles, transmittance T over a distance d is calculated as:

$$T = e^{-d\beta_{ex}}. \tag{2.12}$$

where β_{ex} is the extinction coefficient of the media and is equal to the sum of the absorption and scattering coefficients:

$$\beta_{ex} = \beta_a + \beta_s. \tag{2.13}$$

Transmittance through a volume with non-uniform particle distribution can be calculated by integrating transmittance over a ray path (see Section 2.8.5).

$$T = \int_a^b e^{-d\beta_{ex}(x)} dx. \tag{2.14}$$

Optical depth τ, absorbance A and transmittance T are related by the equation:

$$T = e^{-\tau} = 10^{-A}. \tag{2.15}$$

2.7.2 Scattering

To simulate light scattering, the amount of light that scatters at each direction has to be known. A function that computes this quantity is called a phase function. It calculates the angular distribution of light scattered at a given direction (Fig. 2.7). Given a scattering angle, a phase function computes the percentage of light scattered at the scattering angle. The phase functions presented below are normalized so that integrating over 4π steradians equals 1. The simplest of these is an isotropic scattering function which scatters light uniformly in all directions:

$$p(\theta) = \frac{1}{4\pi}. \tag{2.16}$$

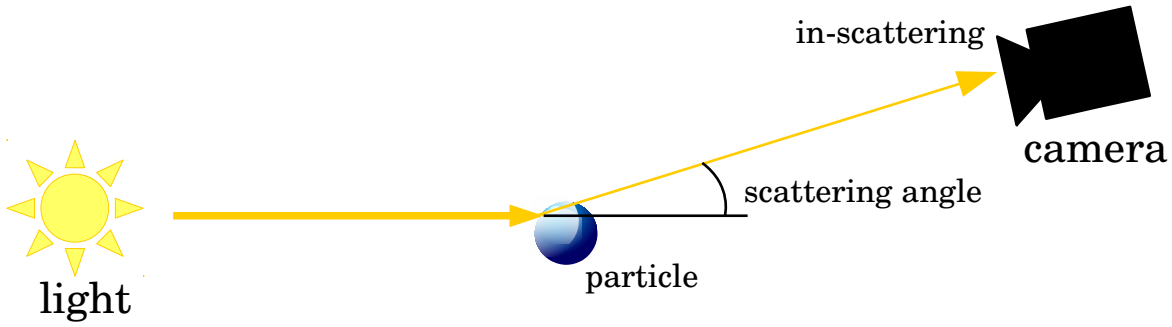


Figure 2.7: Light scattered in a given direction.

Rayleigh Scattering

Rayleigh Scattering is scattering of electromagnetic radiation by particles smaller than its wavelength. Figure 2.8 shows a plot of the Rayleigh phase function.

$$p(\theta) = \frac{3}{16\pi}(1 + \cos^2(\theta)). \tag{2.17}$$

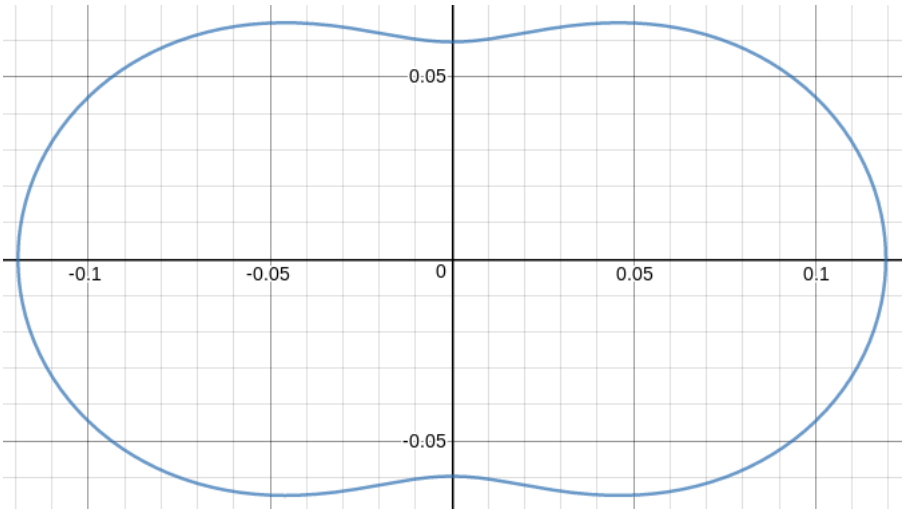


Figure 2.8: Plot of normalized Rayleigh phase function.

Mie Scattering

Mie scattering models scattering of light through particles in the lower portion of the atmosphere with size similar to the wavelength of scattered light (e.g. dust, smoke and water droplets).

Henyey-Greenstein Scattering

The Henyey-Greenstein phase function can be used to model media with strong forward or backward scattering. The parameter $-1 \leq g \leq 1$ enables the modelling of backscattering, isotropic scattering and forward scattering. A value close to zero is similar to isotropic scattering. As the value approaches one forwards scattering increases as illustrated in Figure 2.9

$$p(\theta) = \frac{1}{4\pi}\Big[\frac{1-g^2}{(1+g^2-2g\cos(\theta))^{3/2}}\Big]. \tag{2.18}$$

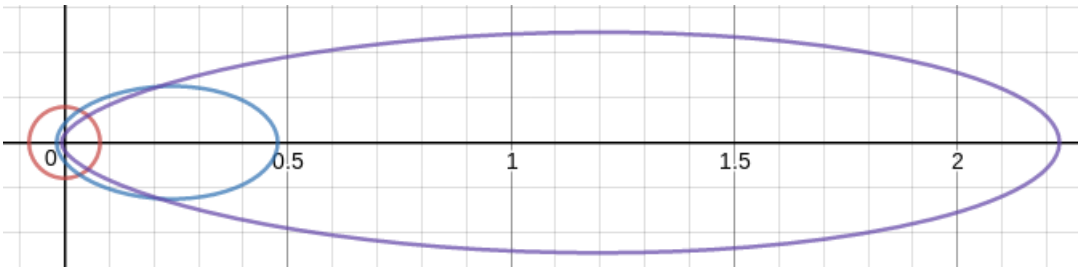


Figure 2.9: Plot of normalized Henyey-Greenstein phase function for values of g=0, 0.5, 0.75 (red, blue, violet).

In-scattering Equation

The in-scattering equation calculates the percentage of light that scatters at a given point at an angle θ. Given a scattering coefficient β_s and phase function $p(\theta)$ of the media the in-scattering equation is calculated as:

$$I_s(\theta) = \beta_s p(\theta). \tag{2.19}$$

2.8 The Rendering Pipeline

The rendering pipeline describes the steps taken to render 3D models to a 2D screen. For real-time purposes the rasterisation pipeline is generally used which includes the three main stages: application, geometry and rasterization (Fig. 2.10) [20].

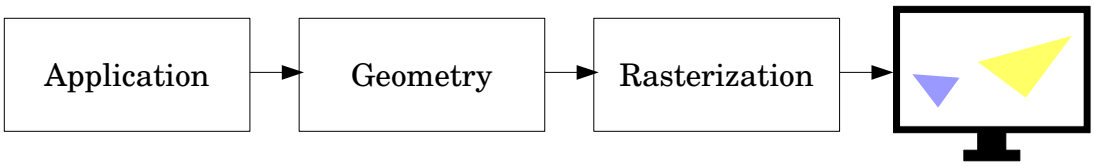


Figure 2.10: Three main rasterization pipeline stages.

2.8.1 Application Stage

In the application stage, 3D models to be rendered in the next frame are passed to the next stage are created or updated based on user input, collision detection, physics

engine or game logic. A virtual camera is used to describe the location of an observer and a look at direction in 3D space.

2.8.2 Geometry Stage

Models obtained from the previous stage contain geometric primitives which are usually triangles, lines or points. A point is also referred to as a vertex. The goal of this stage is to project these primitives defined in 3D model space coordinate system to a screen space coordinate system. A frustum is a volume visible by the camera, its shape is of a pyramid with a trimmed top. Primitives not located inside the frustum are not visible and are thus clipped (Fig. 2.11, 2.12).

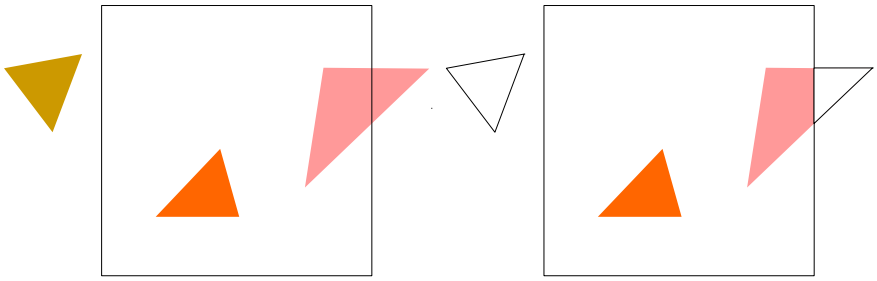


Figure 2.11: Clipping of primitives outside of camera frustum. The yellow triangle is discarded. The pink triangle is clipped and two new vertexes are created.

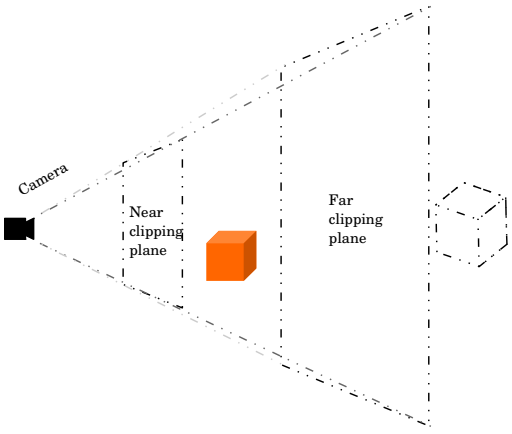


Figure 2.12: Clipping of primitives outside of camera frustum. The orange cube is located in between the two clipping planes and inside the frustum. The other cube is behind the far clipping plane and is thus clipped.

2.8.3 Rasterization Stage

A screen is composed of a grid of pixels which will be referred to as fragments. In this stage, primitives in screen space coordinate system are converted to a set of fragments. (Fig. 2.14). The colour of each fragment is computed by a program (refered to as a fragment shader in OpenGL terminology) specifically written to be executed on the GPU and is output to a buffer which can then be displayed on screen.

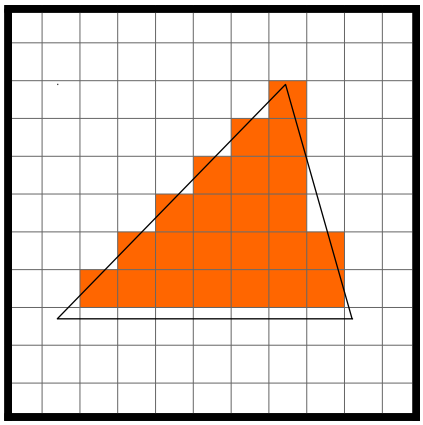


Figure 2.13: Rasterization of a triangle.

2.8.4 Tone Mapping

Tone mapping is a technique where one set of colours is mapped into another. It is generally combined with high-dynamic-range (HDR) rendering or photography, where the range of colours used is outside of the range displayable by a standard LCD monitor. Tone mapping can be used to map the high-dynamic-range into a low dynamic range which is suitable for displaying on a monitor.

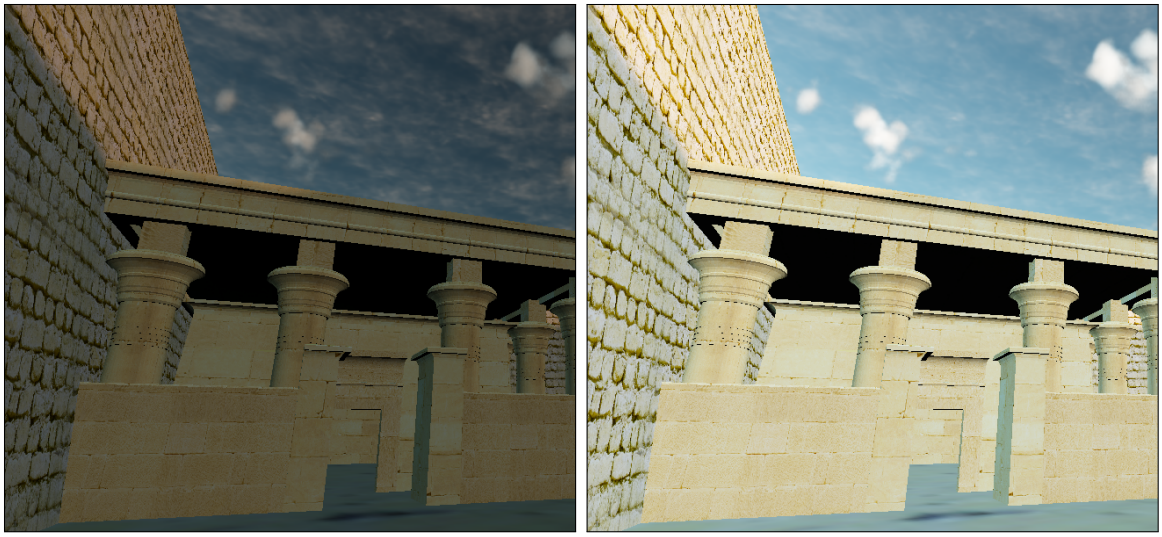


Figure 2.14: Left: image without tone mapping. Right: tone mapped image.

2.8.5 Ray Marching

Ray tracing [21] is a technique in 3D rendering where given a ray with a position and direction an intersection with the ray and the scene is calculated. The colour is then calculated based on the material of the intersected object. Ray marching [22] is a rendering technique similar to ray tracing. Starting at a given position samples are taken in the direction of a ray. If rendering surfaces, at each sample an intersection check is performed and the sampling process usually stops. In volume rendering at each sample, the light scattered and absorbed is computed and the sum is rendered as colour.

2.9 Summary

This chapter introduced the concepts and theory involved in this work. This included the correct calculation of time Julian date, conversions between celestial coordinate systems, introduction to participating media and finally a brief overview of the rasterisation pipeline for real-time rendering.

3 Literature Review

Rendering outdoor scenes requires the rendering of sky, sun and possibly clouds. The light from sky is mostly scattered from the sun. Nishita et al. [3] used a physically based atmospheric model, which models air particles and aerosols and its Rayleigh and Mie scattering. Among the many methods for rendering sky, they can be split into two categories based on their sky model: analytic and iterative. Analytic sky model is evaluated by a function which depends on several coefficients that are used to model the atmosphere. Iterative sky models generally use model such as Nishita's and evaluate it with numerical integration. Similarly, there are rendering methods suitable for real-time rendering and others for offline rendering.

3.1 Rendering of Atmosphere

There is a large amount of literature focused on rendering a realistic looking sky. Some of these are focused on real-time rendering of single or multiple scattering. Real-time methods in most cases involve a 2D, 3D or more dimensional lookup table. Usually computing the sky colour for a ray can by parametrized into four values, namely, angle between zenith and sun direction, ray direction and sun direction, zenith and ray direction and altitude of the observer. Methods that assume the observer is on the ground, provide a simplified model.

3.1.1 Analytic Models

Pretham [1] uses an analytical sky model which can be used in real-time without pre-computation (Fig. 3.1). The model uses a value of turbidity which is a ratio of optical thickness of haze and molecules to optical thickness of molecules. A turbidity value of 1 represents pure air, value of 64 a thin fog.

Figure 3.1: The sky rendered with Preetham model with turbidity of 3 (left morning and right evening, images taken from Preetham [1]).

Hosek et al. [2] use an analytical model which depends on Pretham's. Hosek's model use a separate fitting process for each wavelength to obtain a more realistic colour distribution at a slight increase in computation cost (Fig. 3.2).

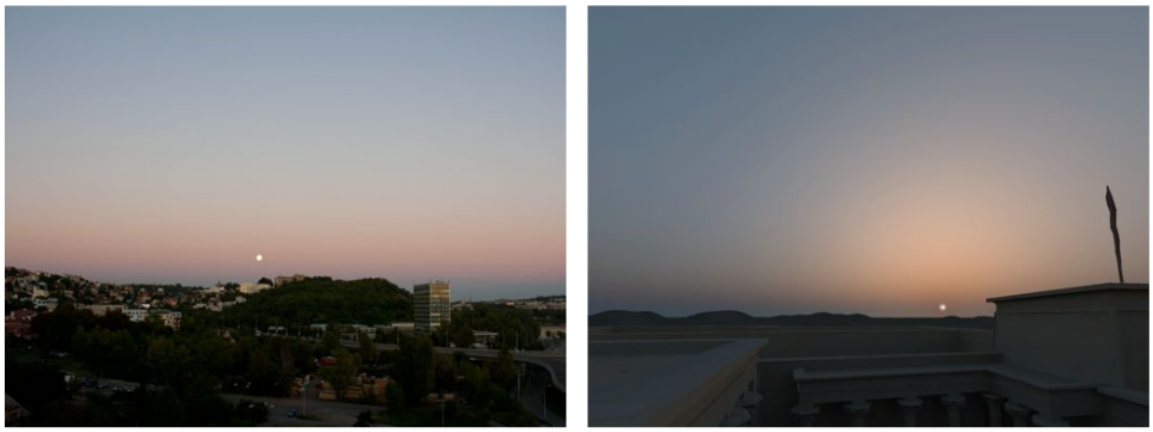


Figure 3.2: Right: Hosek model with turbidity of 9, Left: reference photograph (images taken from Hosek [2]).

3.1.2 Iterative Models

Nishita et al. [3] introduce a model to render Earth as viewed from outer space (Fig. 3.3). The atmospheric model depends on light scattering of air molecules and aerosols. The model uses single scattering, absorption in ozone layer is not accounted for as it is negligible compared to absorption by air molecules and aerosols, the density distribution of air molecules and aerosols are modelled to vary exponentially with

altitude and light is assumed to travel in straight path thought the atmosphere, thus ignoring the varying index of refraction.

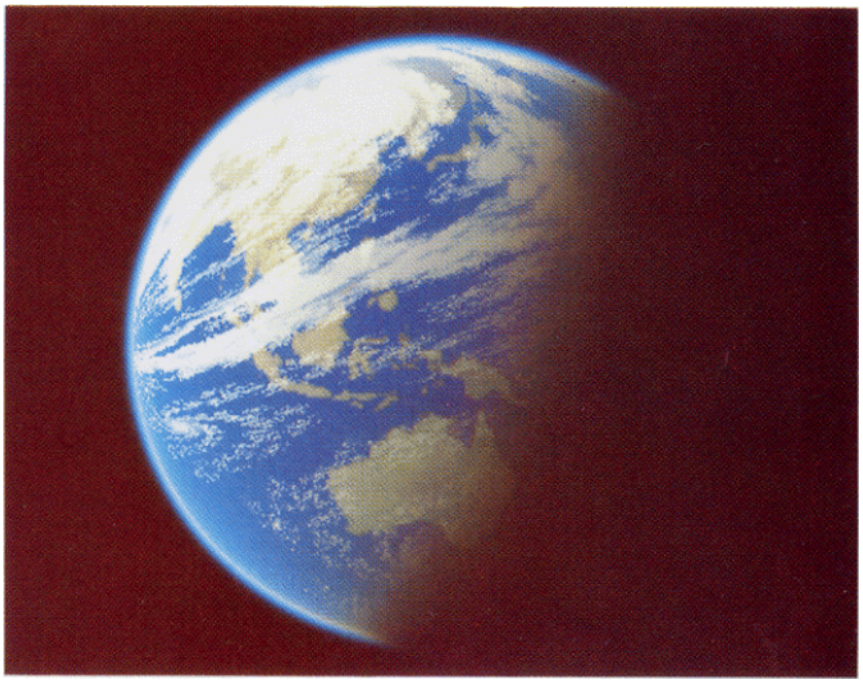


Figure 3.3: A rendering of the Earth as viewed from the space (image taken from Nishita [3]).

Jensen et. al [4] describe an approach to rendering a night sky (Fig. 3.4). Other than sunlight and moonlight other source of light present during night are described, such as: bright planets, zodiacal light, integrated starlight, air glow, diffuse galactic light and cosmic light.

Figure 3.4: A rendering of a night sky (image taken from Jensen [4]).

O'Neil [5] uses Nishita's model and implements it for GPU. Instead of computing transmittance or using a lookup table, a function that fits the values is used instead. The sky is calculated by numerical integration with five samples to achieve real-time performance (Fig. 3.5).

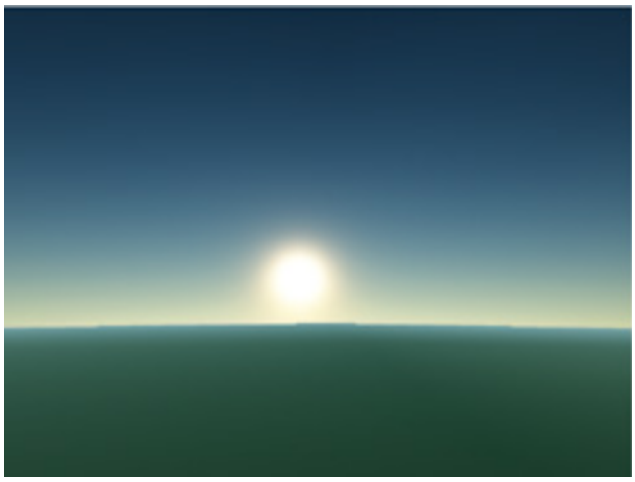


Figure 3.5: A rendering of a sky (image taken from O'Neil [5]).

Bruneton et. al [6] use a model of atmosphere which depends on air molecules and aerosols as in Nishita's model. Multiple scattering is taken into account and sky can be rendered from ground, custom altitude or from outer space (Fig. 3.6). Additionally, ground shadows and light shafts are rendered as well. The sky data is pre-computed into a two 2D lookup tables and one 4D lookup table stored as 8 3D tables within a single 3D table taking up 8 MB of data. Each of these 3D tables stores data for observer at a different altitude and the values are manually interpolated in a shader.

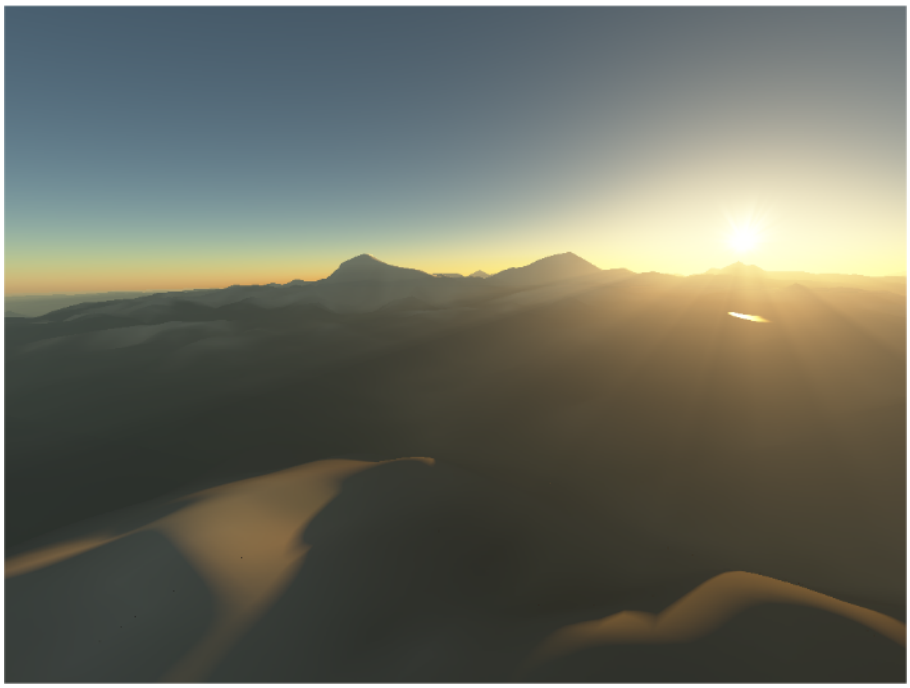


Figure 3.6: A rendering of a sky (image taken from Bruneton [6]).

3.2 Cloud Rendering

Rendering realistic clouds involves rendering cloud shape, cloud colour and possibly animation. There are many distinct methods and approaches used in the literature to render clouds.

Dobashi et al. [7] models clouds and their animation by employing cellular automata which are stored in a 3D array. Rendering is based on a splatting method by using billboards (Fig. 3.7).

Figure 3.7: A rendering of clouds with light shafts. (image taken from Dobashi [7]).

Bouthors et al. [8] method of rendering clouds accounts for all kinds of light paths through the cloud medium and preserves anisotropic scattering (Fig. 3.8). Cloud boundary is represented as a mesh. A procedural volumetric texture is used to add detail under the cloud boundary.

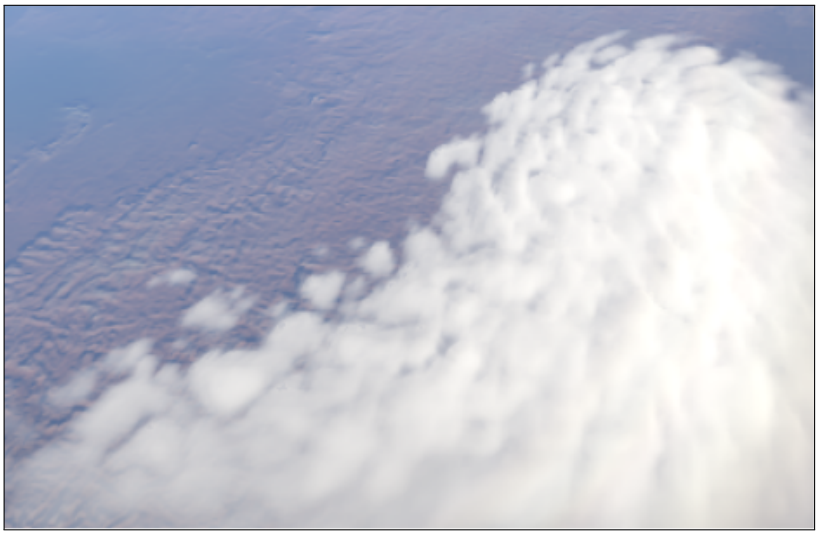


Figure 3.8: A rendering of clouds (image taken from Bouthors [8]).

Harris et al. [9] uses partial differential equations of fluid motion among other methods to model a cloud. This methods uses voxels to store cloud data (Fig. 3.9). Data is stored as a 3D table packed into a 2D texture

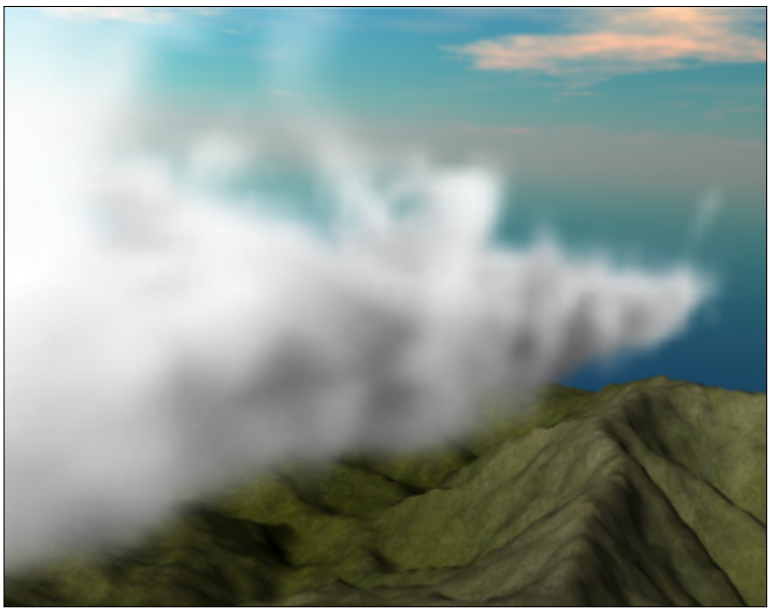


Figure 3.9: A rendering of clouds (image taken from Harris [9]).

Schneider et al. [10] use volumetric modelled with 3D noise textures (Fig. 3.10). 128^3 texture of Perlin-Worley noise is used for the base shape of a cloud. Additional 32^3 detail texture, composed of layered Worley noise of different frequencies per channel is used to add detail. A weather map image is used to control the presence of clouds. The weather map is a top down 2D texture, red channel contains information where clouds are present, green channel contains clouds' height and blue the clouds' starting altitude. The clouds are rendered by ray marching the density field created with the noise textures. At most 64 samples are taken for rays looking at the zenith and at most 128 samples for rays looking at the horizon. This is due to the fact that the ray path is much longer at the horizon. At each step where density is not zero 6 additional samples are used to determine whether the current sample is in shadow. The algorithm is optimized so that it does cheap work if its not in cloud. 1 out of 16 pixels of each 4×4 block is updated per frame and re-projection from previous frame is used to further optimize this approach.

Figure 3.10: A rendering of clouds (image taken from Schneider [10]).

3.3 Summary

This chapter looked at the existing literature involved with the rendering of physically based atmosphere and clouds. There were two main approaches to render atmosphere: iterative and analytical. The methods discussed on rendering clouds were more diverse and included rendering clouds with the use of participating media, voxels, sprites, splatting and others. The following chapters will introduce the method used in this work to render the atmosphere and clouds.

4 Sky Model

This chapter presents a physical sky model used to model the sky and a method to evaluate it to obtain the colour of the sky. Among sky models, analytical models use a set of parameters to calculate the colour of the sky, for instance, Preetham model [1]. Iterative models rely on ray marching in order to integrate the in-scattered light. For real-time algorithms this may involve pre-computing data and storing it in lookup tables. Analytical models can be based on actual sky radiance. Iterative model introduced by Nishita et al. [3] is a physical model that models atmospheric scattering due to air molecules and aerosols in the atmosphere. The Nishita's model of the atmosphere was chosen for this work because it is physically based and is more flexible compared to an analytical models. With the right scattering coefficients it may be used to render skies located on other planets. The model is described in the following section.

4.1 Atmospheric Model

As light travels through Earth's atmosphere, a portion of it gets absorbed or scattered in different direction. At small distances this is not visible, however when looking at distant mountains they fade into a blue colour of the sky. Similarly, the sky has its colour because of scattering. What the light is scattering off are small particles such as air molecules and aerosol particles. In order to simulate this scattering a model is constructed which specifies the location and density of these particles. Earth is modelled as a sphere with radius $R_g = 6360$km and atmosphere with radius $R_a = 6420$km. The density of the particles decreases exponentially with altitude.

The atmosphere is treated as participating media with scattering coefficients β_s^R, β_s^M and absorption coefficients β_a^R, β_a^M. Two sets of coefficients are used, for Rayleigh and Mie scattering, one to represent particles with size smaller than the wavelength of light and the other larger. $p(\theta)^R$, $p(\theta)^M$ are their respective phase functions. Since the atmosphere is not an uniform medium, as the density of particles depends on altitude, the scattering coefficients are represented by a function of altitude:

$$\beta_s^R(h) = \beta_s^{R_0} e^{-h/H_R} \tag{4.1}$$
$$\beta_s^M(h) = \beta_s^{M_0} e^{-h/H_M} \tag{4.2}$$

.

$\beta_s^{R_0}$, $\beta_s^{M_0}$ are scattering coefficients at sea level, $H_R = 8\text{km}$,$H_M = 1.2\text{km}$ are scale heights. Coefficients used are:

$$\beta_s^{R_0} = [5.5e^{-6} \quad 13e^{-6} \quad 22.4e^{-6}] \tag{4.3}$$
$$\beta_s^{M_0} = [21e^{-6} \quad 21e^{-6} \quad 21e^{-6}] \tag{4.4}$$
$$\beta_a^R = \beta_a^M = [0 \quad 0 \quad 0] \tag{4.5}$$

4.2 Sky Colour Function

The colour of the sky illuminated by single source of light can be reduced to a function $L(\alpha, \mu, \gamma, h)$ with four arguments: view zenith angle α, view sun angle μ, zenith sun angle γ and observer altitude h (Fig. 4.1). In the work presented by Bruneton, this function is pre-computed and stored in 4D lookup table, while in this work this function is computed at runtime.

The transmittance of a ray though atmosphere can be reduced to function $T(\alpha, h)$ with two arguments: view zenith angle α and observer altitude h:

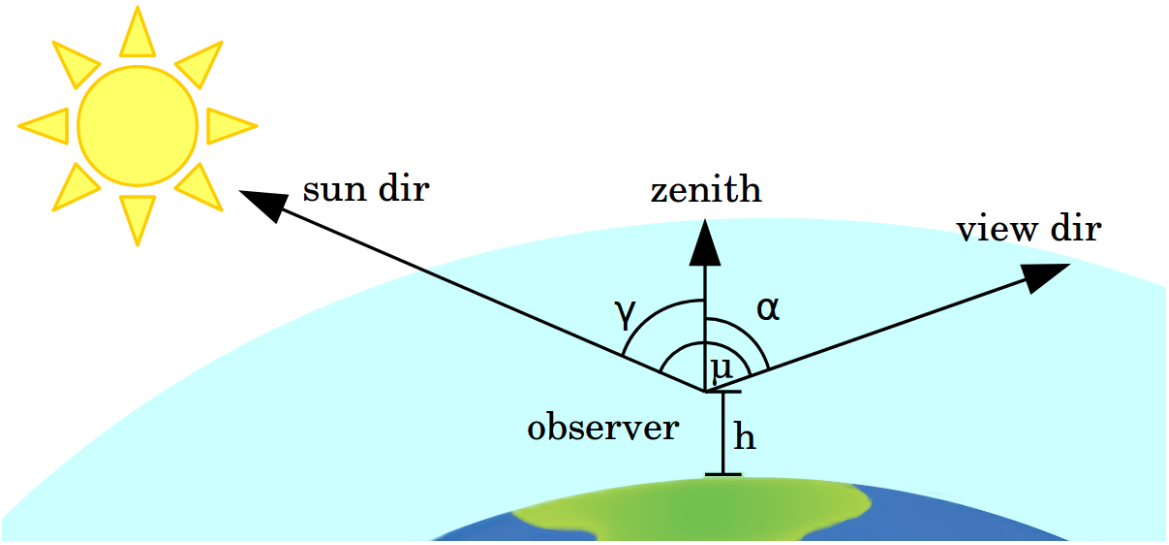


Figure 4.1: Parametrizing the sky colour function based on observer's altitude, view direction and sun direction.

To evaluate L for a single ray for single scattering, the points where the ray enters and exits atmosphere are computed and the path is split into several segments to begin the integration process. At each segment the amount of sunlight that reaches the segment is calculated as a product of $T(\alpha, h)$ and sunlight irradiance (Fig. 4.2).

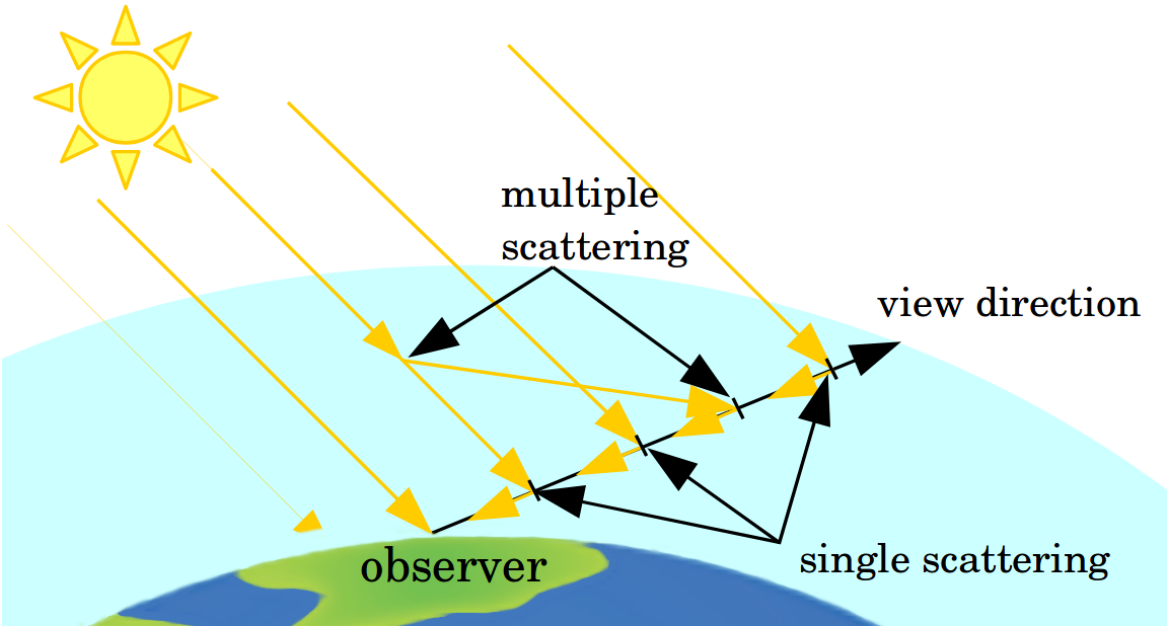


Figure 4.2: Scattering of light in the atmosphere.

4.3 Calculating Path Length

A ray can be defined by its starting point s and direction d. Thus a point on ray is equal to $p(t) = s + dt$. Entry and exit points to the atmosphere $a = p(t_a), b = p(t_b)$ are calculated by using mathematical definitions of circle and line, or sphere and line. Due to the symmetry of sphere, t_a, t_b can be calculated by knowing α, h. The function $P(\alpha, h)$ which returns t_a, t_b is pre-computed into a lookup table.

4.4 Calculating Transmittance

Calculating $T(\alpha, h)$ represents calculating the percentage of light that passes through the atmosphere.

$$T(\alpha, h) = e^{-(\beta_{ex}^{R}\tau(H_R)+\beta_{ex}^{M}\tau(H_M))} \tag{4.6}$$

Transmittance T is calculated by first calculating the optical depth τ. Due to symmetry the ray origin and direction can be represented by a 2D vector. Given α, r the start and end points of ray path a, b are calculated. Let d be the distance between a, b. S is the number of integration samples. Points $p_0, p_1, ...p_{S-1}$ lie on the path ab, and $h_0, h_1, ...h_{S-1}$ represent the altitude for each point respectively. Then the optical depth is calculated as:

$$\tau(H) = \frac{d}{S} \sum_{i=0}^{S-1} e^{-h_i/H} \tag{4.7}$$

Transmittance function $T(\alpha, h)$ is pre-computed into a 2D lookup table.

4.5 Calculating In-scattered Light

To calculate the in-scattered light, points $a = p(P(\alpha, h)_a), b = p(P(\alpha, h))_b$ are calculated which represent the points where the ray enters and exits the atmosphere. a and b are represented by 3D vectors. The distance $d = |a - b|$, S is the number of samples, 4 to 12 samples are used depending on the distance d. Points $p_0, p_1, ...p_{S-1}$ lie on the path ab, and $h_0, h_1, ...h_{S-1}$ represent the altitude for each point respectively. Points $\hat{p_0}, \hat{p_1}, ...\hat{p}_{S-1}$ represent normalized vectors.

$$L(\alpha, \mu, \gamma, h) = p(\mu)^{ray} \beta_{ex}^{ray} R(H^{ray}) + p(\mu)^{HG} \beta_{ex}^{mie} R(H^{mie}) \tag{4.8}$$

$$O_I(H) = \frac{d}{S} \sum_{i=0}^{I} e^{-h_i/H} \tag{4.9}$$

$$R(H) = \sum_{i=0}^{S-1} e^{-(\beta_{ex}^{ray} O_i(H^{ray}) + \beta_{ex}^{mie} O_i(H^{mie}))} T(\hat{p_i} \cdot L, h^i) \tag{4.10}$$

4.6 Rendering

The sky rendering system was implemented in Java with the use of JMonkeyEngine [23], which is a 3D game engine written in Java which uses OpenGL[24].

4.6.1 Lookup Tables

Lookup tables refer to the use of pre-computing data and storing it in a memory for later access. The advantage of lookup tables is in retrieving a pre-computed result from a memory instead of computing it every time. The trade-off is in an increased memory usage. $P(\alpha, h)$ and $T(\alpha, h)$ are stored in two 512×512 textures. A texture is a 2D array stored in the video memory. A 2D texture $T^{LUT}(x, y)$ (Fig. 4.3) with bilinear interpolation is used to map values within the range $0 \leq x \leq 1$, $0 \leq y \leq 1$.

$$T(\alpha, h) = T^{LUT}(0.5cos(\alpha) + 0.5, \frac{h}{R_a - R_g}) \tag{4.11}$$

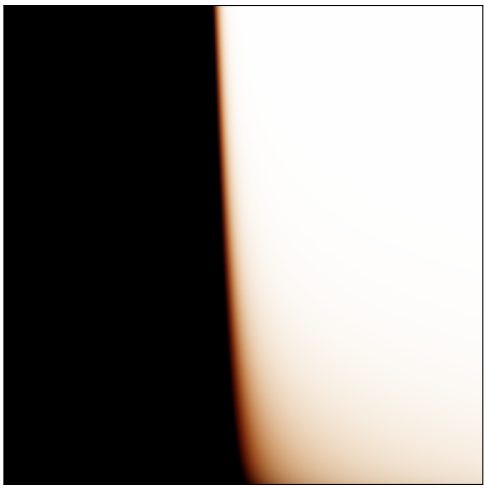


Figure 4.3: The transmittance lookup table.

The cosine of angle α is used instead as it is equal to a dot product between two vectors. While the above mapping works, the precision is limited to the lookup table. The density of particles increases exponentially with altitude, the above mapping is however, linear. Similarly, more texture samples could be located near the horizon. The following mapping (Fig. 4.4, 4.5) better utilizes the texture space and improves the accuracy near the horizon:

$$P(\alpha, h) = P^{LUT}\left(\frac{1}{2}|sgn(\alpha)cos(\alpha)|^{1/3} + \frac{1}{2}, \frac{h}{R_a - R_g}^{1/4}\right) \tag{4.12}$$

$$T(\alpha, h) = T^{LUT}\left((0.6 + |sgn(\alpha)cos(\alpha)|^{1/3})0.625, \frac{h}{R_a - R_g}^{1/4}\right) \tag{4.13}$$

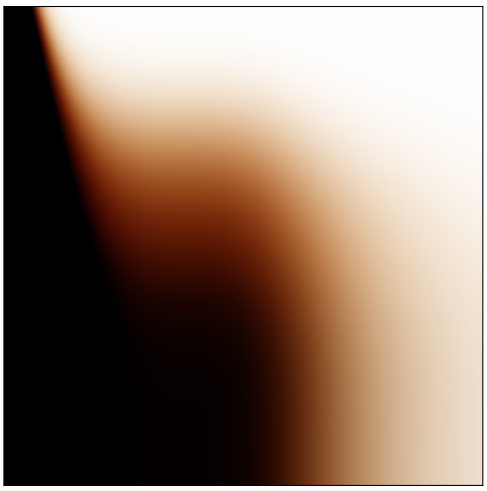


Figure 4.4: The transmittance lookup table with with non-linear mapping.

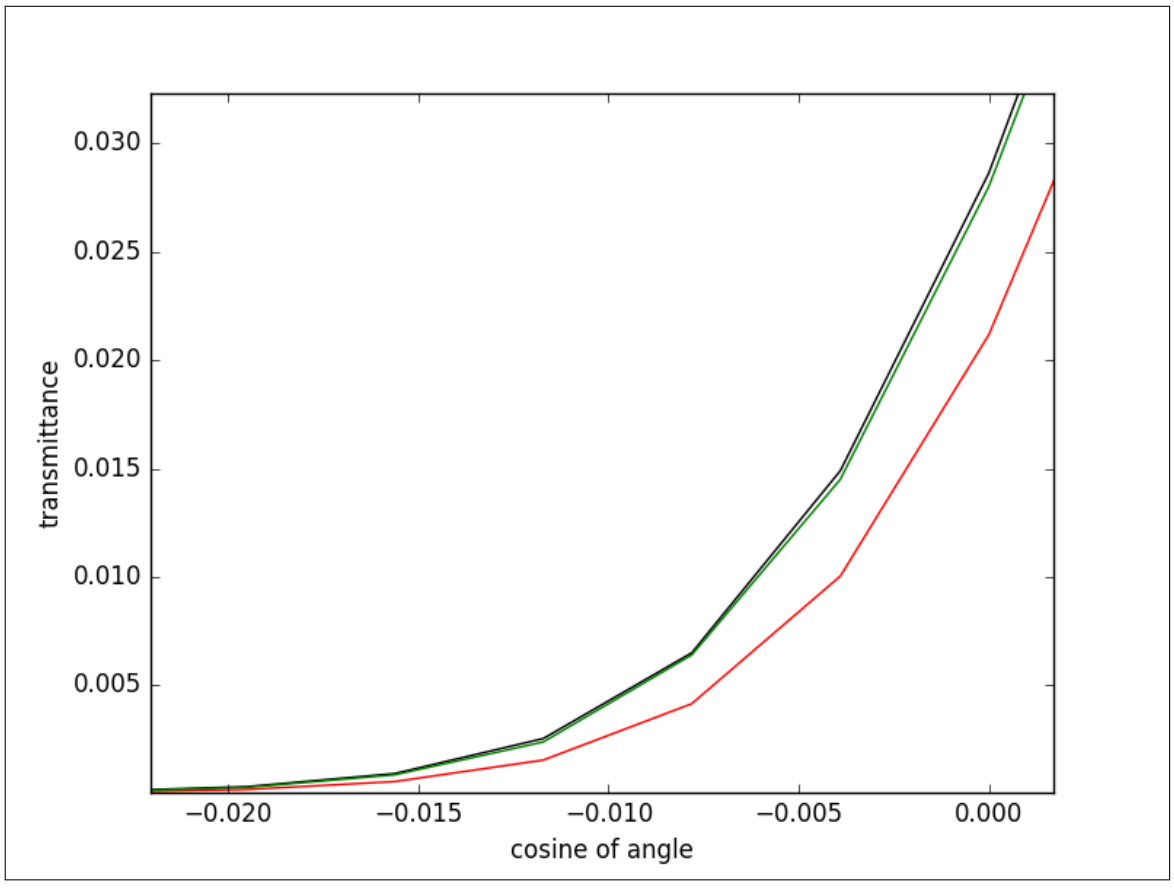


Figure 4.5: Transmittance values black (reference), red (linear mapping), green (cubic mapping).

4.6.2 Rendered Sky

In order to evaluate the model, ray marching is used. Since the rasterization pipeline is used a full screen quad is rendered instead and ray marching is performed in the fragment shader (Fig. 4.6). The sky can be rendered as a 360° map (Fig. 4.7) or as a part visible by a camera only. The advantage of first is that if the outside conditions are not changing from frame to the next, the sky can be pre-computed into a 360° texture. Figure 4.8 shows a 360° view of the sky rendered at different resolutions, the sky texture with resolution of 128 is hardly distinguishable from the sky texture with resolution of 512. For performance reasons the sky is rendered to a texture with half the resolution of the screen.

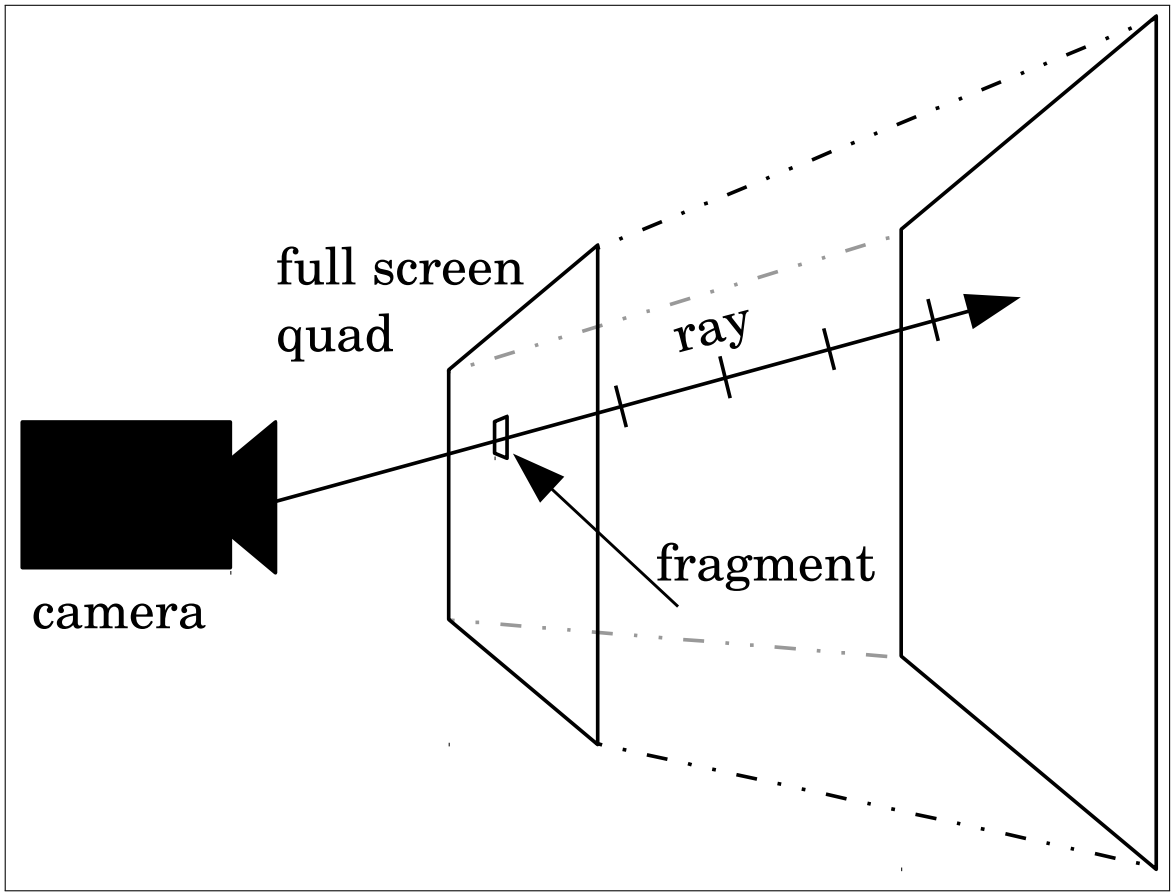


Figure 4.6: Rendering full screen quad to perform ray marching in a fragment shader.

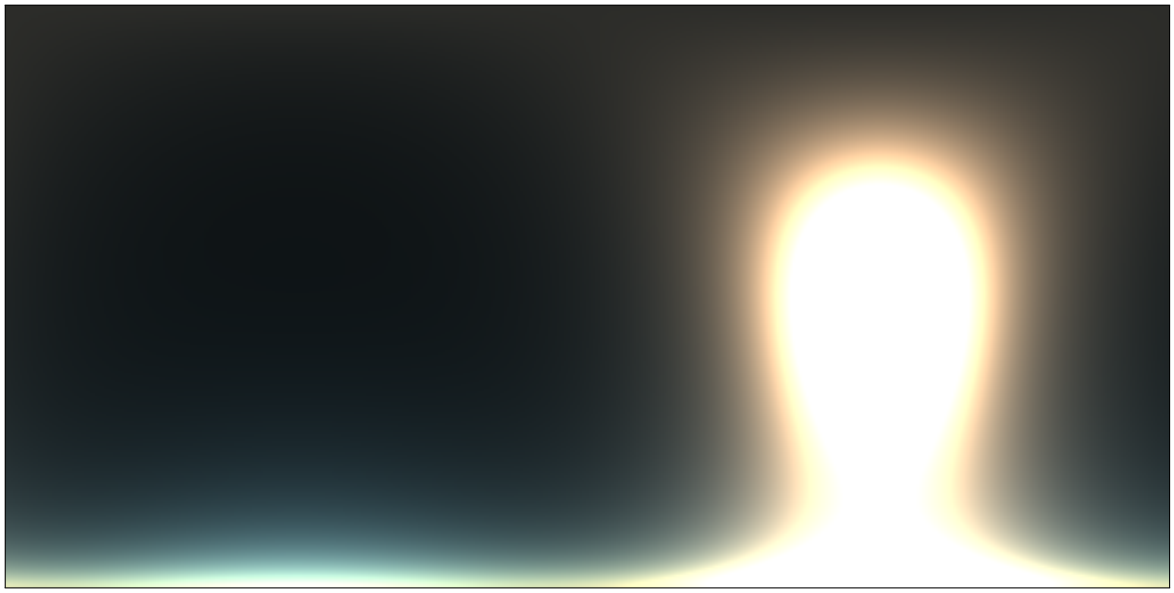


Figure 4.7: 360° view of sky rendered into a texture of size 512×512 (upper portion is shown only).

Figure 4.8: 360° views of the sky rendered into textures of sizes 128, 64, 32 and 16 pixels (from left to right).

4.6.3 Tone Mapping

The above image shows a bright Sun and a dark sky. This is because irradiance is a radiometric quantity. A linear increase in irradiance does not provide a linear visual

increase of brightness. Tone mapping function $T(c)$ is used to transform colours with a scale value $T_s = 4$ (Fig. 4.9).

$$T(c) = 1.0 - e^{-T_s c} \tag{4.14}$$

Figure 4.9: 360° view of the sky with tone mapping.

4.6.4 Aerial Perspective

Distant mountains and objects fade into the colour of the sky. Given a colour c of an object at distance d, transformed colour is given by $cT + L$, where $T = e^{-d\beta_{ex}}$ is the transmittance between the observer and the given point and L is in-scattered light calculated with a single sample. When rendering the sky, β_{ex} is computed by numerical integration. In this case an average value is used, given observer altitude h_o, and observed point altitude h_p, $\beta_{ex} = \beta_s^R(\frac{h_o+h_p}{2}) + \beta_s^M(\frac{h_o+h_p}{2}) + \beta_a^R(\frac{h_o+h_p}{2}) + \beta_a^M(\frac{h_o+h_p}{2})$.

4.6.5 Sky Light and Image-based Lighting

Outdoor scenes are lit by light from sky. To simulate this light a 360° view of the sky is rendered into 16×16 texture (Fig. 4.10). The diffuse lighting contribution of the sky map is decomposed into 6 directional lights. The resolution of 16×16 is sufficient for this purpose.

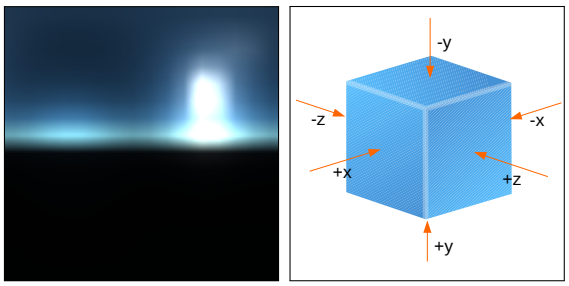


Figure 4.10: Left: 16×16 sky light map, right: one light for each axis.

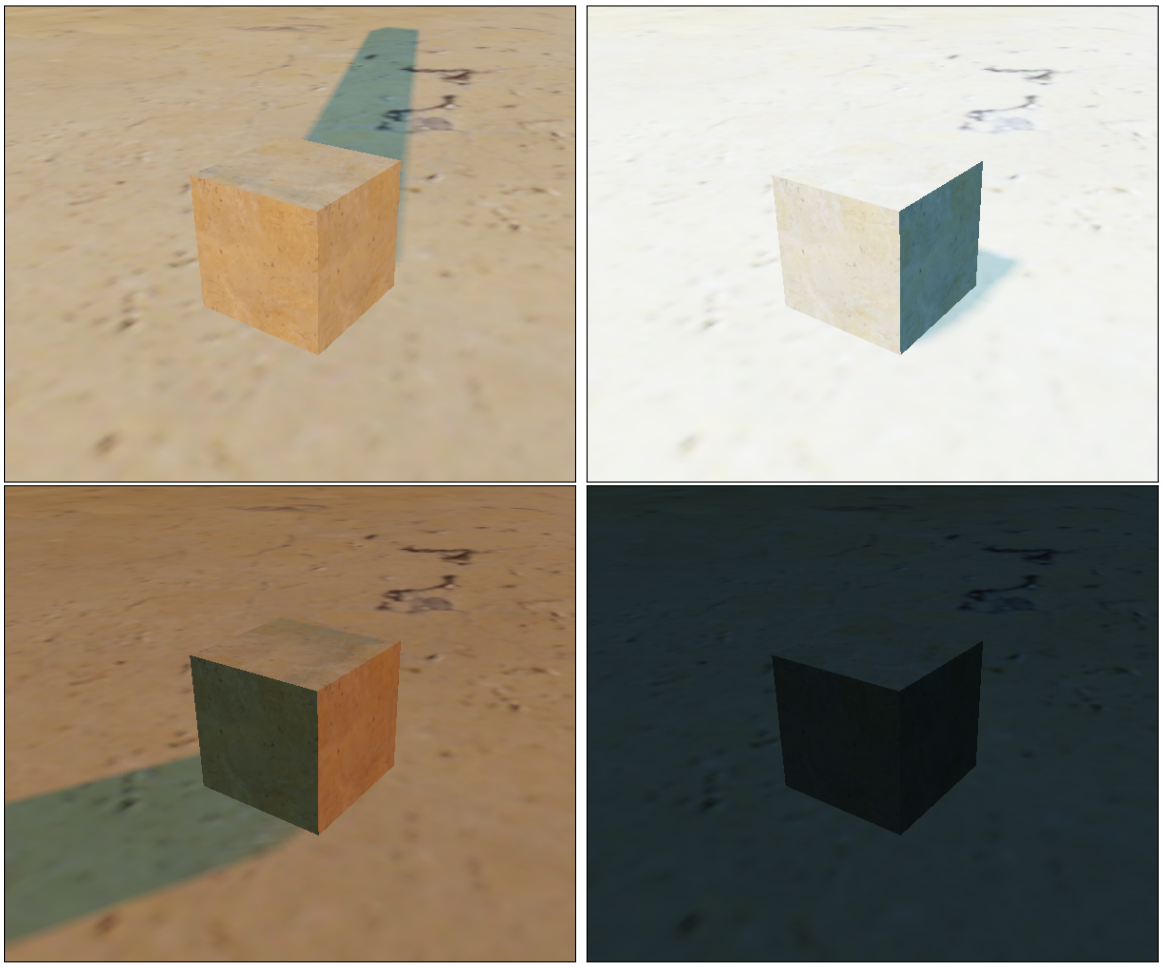


Figure 4.11: Cube rendered with one directional light for the sun and six directional lights for the sky at sunrise, noon, sunset and night.

A directional light is created for each axis (D_{+x},D_{-x},D_{+y},D_{-y},D_{+z},D_{-z}) (Fig. 4.10, 4.11). The contribution of lighting of each pixel of sky light map is added to the directional lights. This is accomplished by summing the contribution of each texel for each directional light. For each texel c in the sky light map and its direction d. If

$d_x > 0$ then $D_{+x} = D_{+x} + c|d_x|$. Similarly, if $d_x < 0$ then $D_{-x} = D_{-x} + c|d_x|$. This is repeated for the y and z axes accordingly.

4.7 Results

Figure 4.14, 4.15 and 4.16 show the result of rendering the sky at sunrise and during day. The rendering was done on Intel HD 4400 with resolution 1600×900, 8 GB of RAM and Intel i7-4510U CPU. The sky was rendered to a texture with half the resolution of the screen. The result result was then copied to the screen. Figure 4.13 shows the time taken for each of these actions.

Time	Time[ms]
Sky texture render (800×450)	0.29
Copy to screen	2.9

Figure 4.12: Average rendering time of sky for Figure 4.14 at resolution 1600×900.

Time	Time[ms]
Sky texture render (800×450)	0.29
Copy to screen	1.89

Figure 4.13: Average rendering time of sky for Figure 4.14 at resolution 1280×720.

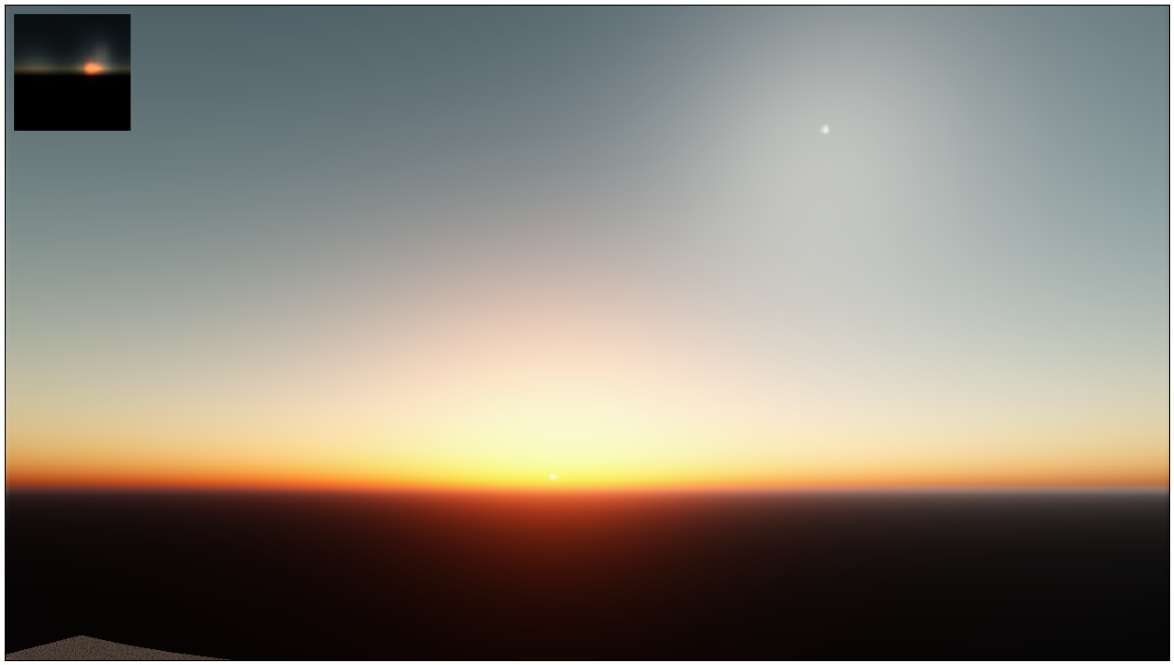


Figure 4.14: Sunrise with visible moon. 16×16 Sky light map is shown in the upper left corner.

Figure 4.15: Sun high in the sky next to moon. 16×16 Sky light map is shown in the upper left corner.

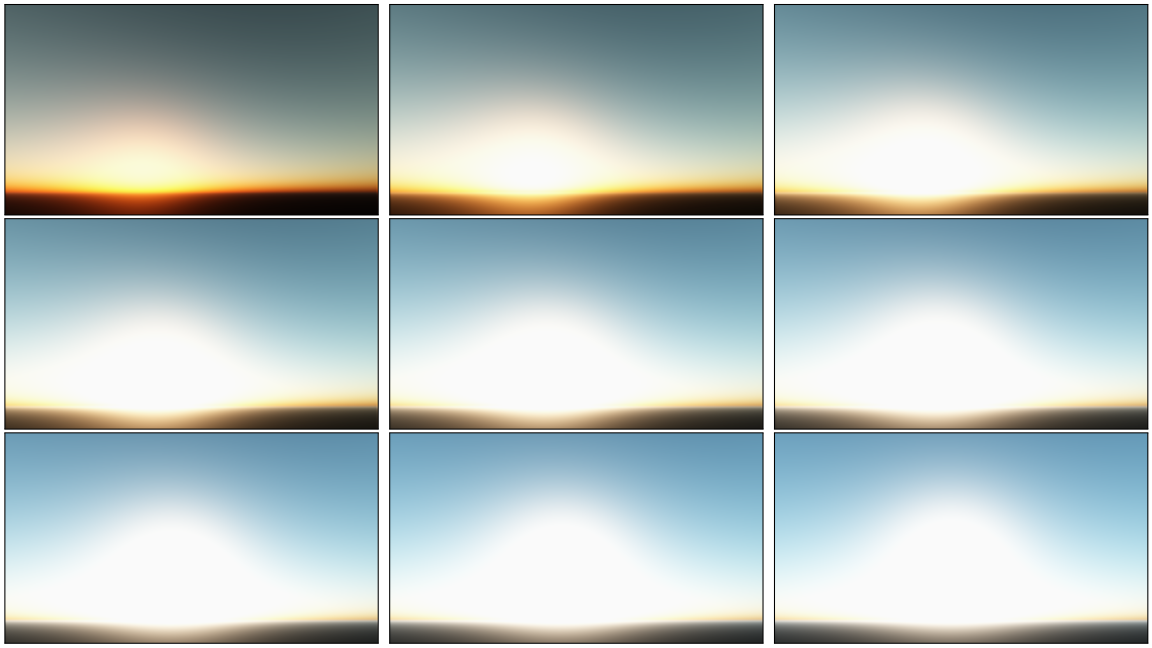


Figure 4.16: Sunrise rendered from 4:20 am in 10 minute intervals.

4.7.1 Night Sky

The atmospheric scattering due to light reflected off the Moon is calculated in the same way as light scattered from the Sun. The difference is the amount of light that the Moon reflects from the Sun. Since the Sun is considerably far away an optimization can be made to consider the rays of light emitted from sun to be parallel, the same cannot be applied for the Moon. The coordinates of stars are obtained from HYG database [25]. The star colour is approximated based on the temperature stored in the database. The location of the star is calculated based on coordinates of the star and its proper motion taking into consideration the coordinates of the observer. Figures 4.18 and 4.19 show the rendered sky as viewed from Malta on the 1st May 2018 at 9pm GMT. Figures 4.20, 4.21 show two additional rendered images as viewed from Malta and their comparison with the planetarium software [11].

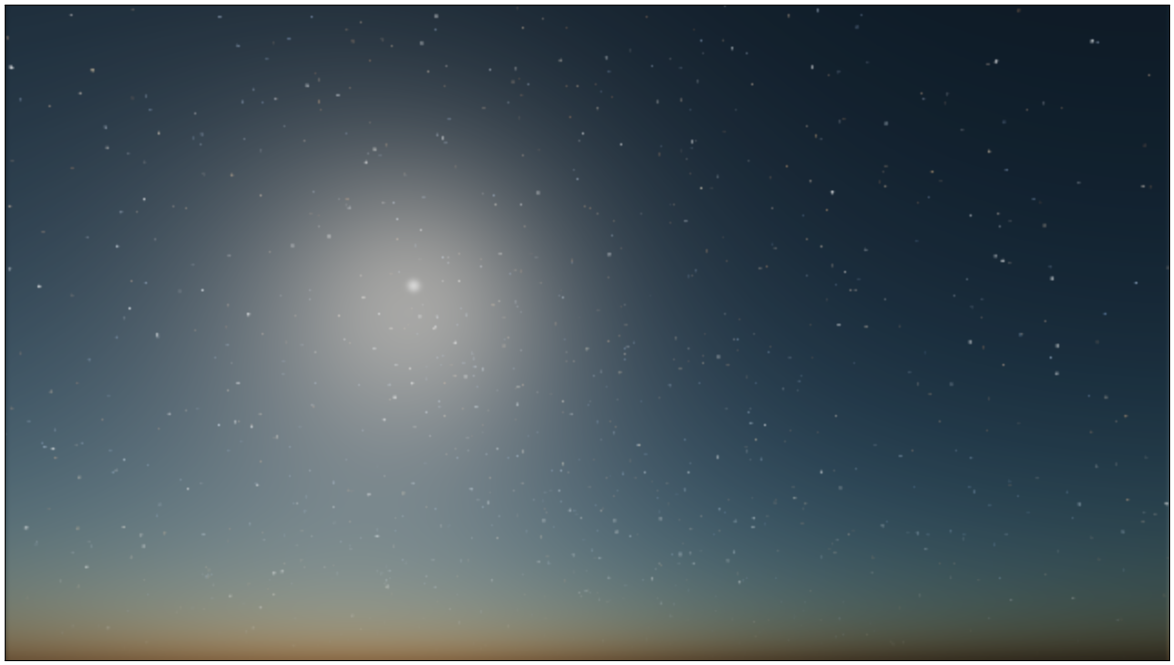


Figure 4.17: Night sky with moon and stars.

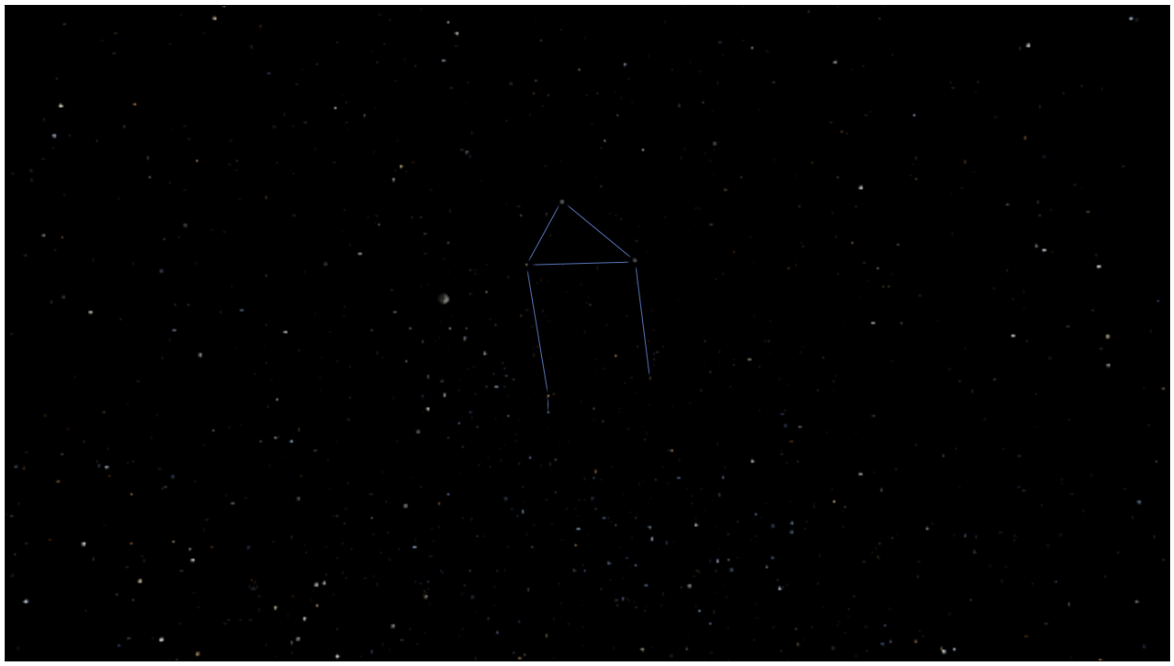


Figure 4.18: Night sky with moon and stars rendered without an atmosphere. The constellation of Libra is highlighted.

Figure 4.19: Night sky rendered by a planetarium software [11].

Figure 4.20: Night sky rendered with atmosphere (left), without atmosphere with the Leo constellation highligted (center) and with a planetarium software (right) [11] on the 1st March 2018 at 19:00 GMT.

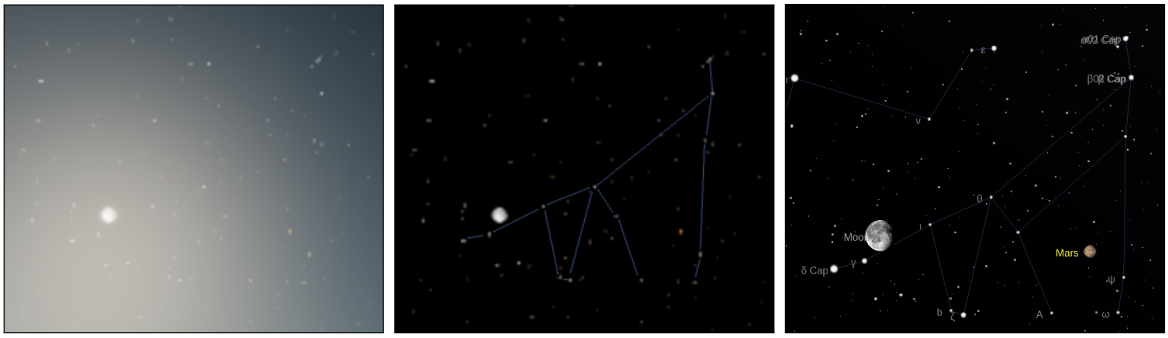


Figure 4.21: Night sky rendered with atmosphere (left), without atmosphere with the Capricorn constellation highligted (center) and with a planetarium software (right) [11] on the 1st July 2018 at 23:00 GMT.

4.8 Summary

This chapter presented the physical model for rendering atmospheric scattering. The model was evaluated with the use of numerical integration while real time performance was achieved with the use of lookup tables. To obtain a realistically looking sky the reason of using tone mapping was explained. Afterwards, a cheap method to render colouring due to aerial perspective and sky lighting was provided.

5 Cloud Model

Among the many ways of rendering volumetric clouds some are suitable to be used in real time. The method described here is similar to work of Schneider [10]. The similarity is in the use of a weather map, Worley noise and ray marching to render the clouds. His work uses from 64 to 128 samples per ray and as an optimization temporal re-projection and the render of clouds is spread across 16 frames. This work uses 16 samples per ray without the aforementioned optimisations. The visual quality is instead preserved by offsetting each ray by a 4×4 matrix, generating a mipmap and sampling it to obtain a smoothed image.

5.1 Cloud Variants

Clouds are named based on their shape and altitude in the sky. There are clouds in the low level of atmosphere ranging from ground to 2km, mid level from 2km to 5km and high level from 5km upwards. Three common clouds: cumulus, stratus, cirrus are showed in the next section. Besides these types, cloud such as cumulonimbus can range from 2km to 18km.

From these various types of clouds cumulus clouds were rendered with volumetric approach. Above the volumetric cloud layer cirrus clouds were modelled with 2D Curl noise.

5.1.1 Cumulus

Cumulus (Fig. 5.1) is a cloud located in the low level of atmosphere. In the same level stratocumulus is located. Altocumulus is located in mid level and cirocumulus in high level of the atmosphere.

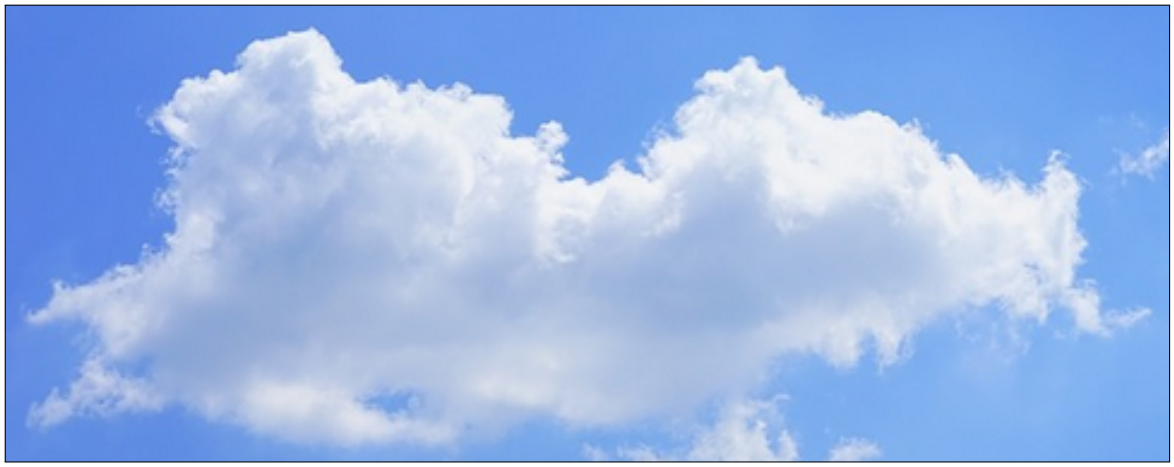


Figure 5.1: Cumulus (Photograph, public domain).

5.1.2 Stratus

Stratus (Fig. 5.2) cloud us located in the low level of atmosphere. Similarly, altostratus is located in mid and cirrostartus in high levels of the atmosphere.

Figure 5.2: Stratus (Photograph, public domain).

5.1.3 Cirrus

Cirrus (Fig. 5.3) is located in the high level of atmosphere.

Figure 5.3: Cirrus (Photograph, public domain).

5.1.4 Clouds as Participating Media

Volumetric clouds are modelled as participating media with non-uniform density, scattering coefficient β_s^M, absorption coefficient β_a^M, located between altitudes H_{from}, H_{to}. The density is given by density function $D(p)$ where p is 3D vector representing a position with origin at sea level.

5.1.5 Weather Map

Weather map is a top down image with information regarding cloud density and height. It consists of three channels, red channel stores cloud density $W_{density}(p)$, green and blue store cloud heights $W_H^{bottom}(p)$, $W_H^{top}(p)$ (Fig. 5.4).

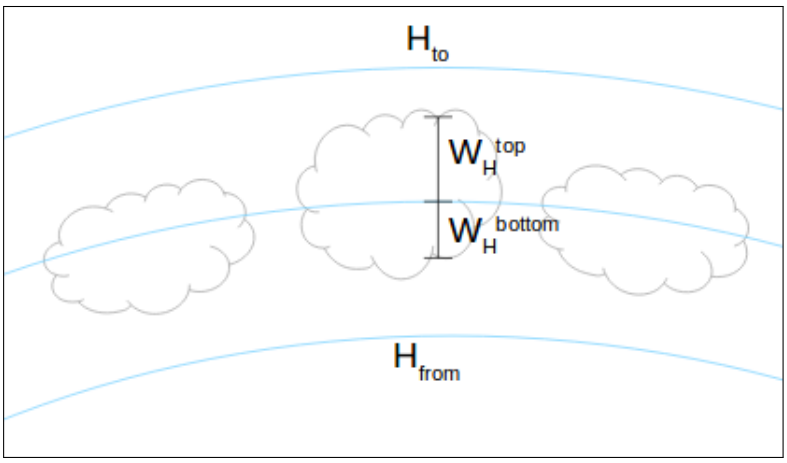


Figure 5.4: Cloud height $W_H^{bottom}(p)$, $W_H^{top}(p)$.

5.1.6 Density Function

Density function $D(p)$ depends on the top down weather map containing cloud density $W_{density}(p)$ and its height $W_H^{bottom}(p)$, $W_H^{top}(p)$; and a 3D Worley noise(Fig. 5.5) $N(p)$. Given planet radius R_{planet}, a position vector with origin at sea level can be converted to an altitude with function $A(p)$.

$$A(p) = \left| \begin{bmatrix} p_x & p_y + R_{planet} & p_z \end{bmatrix} \right| - R_{planet} \tag{5.1}$$

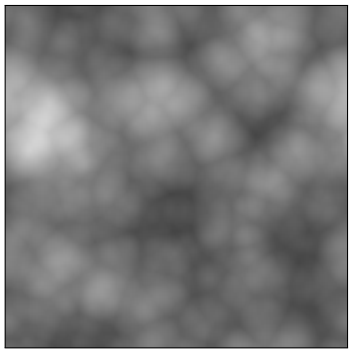


Figure 5.5: 2D Slice of Worey Noise function $N(p)$.

Two functions are defined to further define the shape of a cloud: $H_{scale}(p)$ increases cloud density at their bottom and $H_{gradient}(p)$ increases cloud density in lower sections of the atmosphere. The volumetric cloud layer is located between altitude H_{from}, H_{to}. 2000m and 3000m above ground is used respectively.

$$H_{scale}(p) = (A(p) - a)(A(p) - a - h)(-4/h^2) \tag{5.2}$$

where:
$a = 0.5(H_{to} - H_{from})(1 - W_H^{bottom}) + H_{from}W_H^{bottom}$,
$h = 0.5(H_{to} + H_{from})(W_H^{bottom} + W_H^{top})$

$$H_{gradient}(p) = (A(p) - H_{from})/(H_{to} - H_{from}) \tag{5.3}$$

$W_{coverage}$ parameter is used to change the amount of clouds present, with 0 representing a clear sky and 1 representing the density as expressed in the weather map.

$$D^{coverage}(d) = max(d + W_{coverage} - 1, 0)/W_{coverage} \tag{5.4}$$

$$D(p) = clamp([D^{coverage}(W_{density}(p))H_{scale}(p) - N(p)]H_{gradient}(p), 0, 1) \tag{5.5}$$

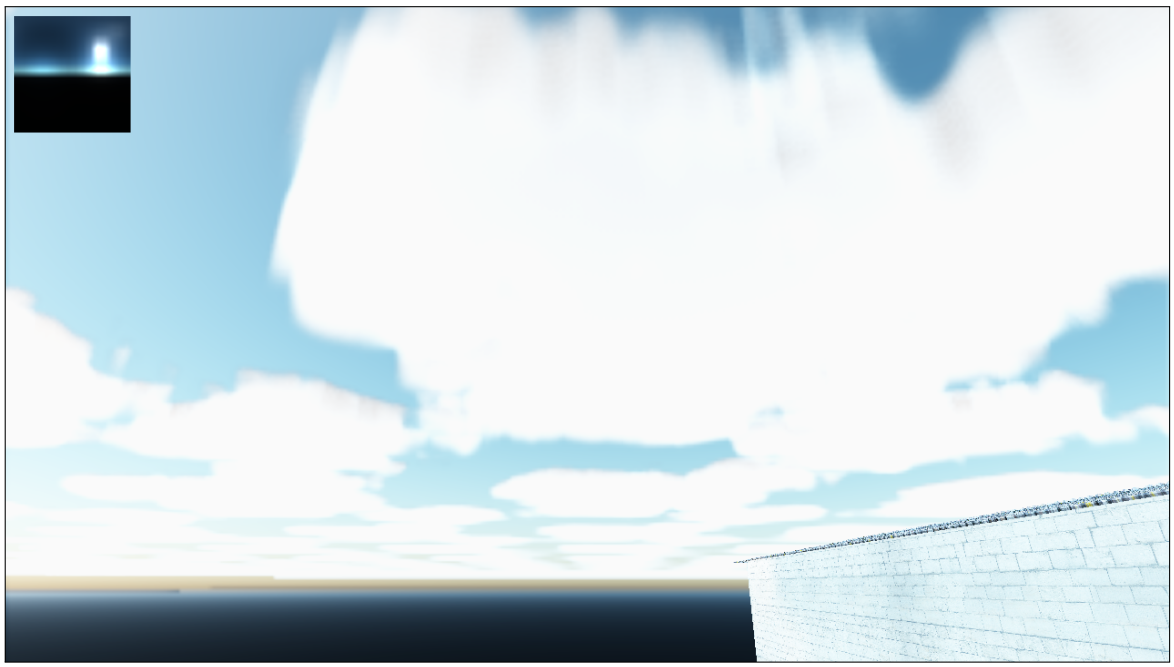


Figure 5.6: Render of clouds without Worey Noise.

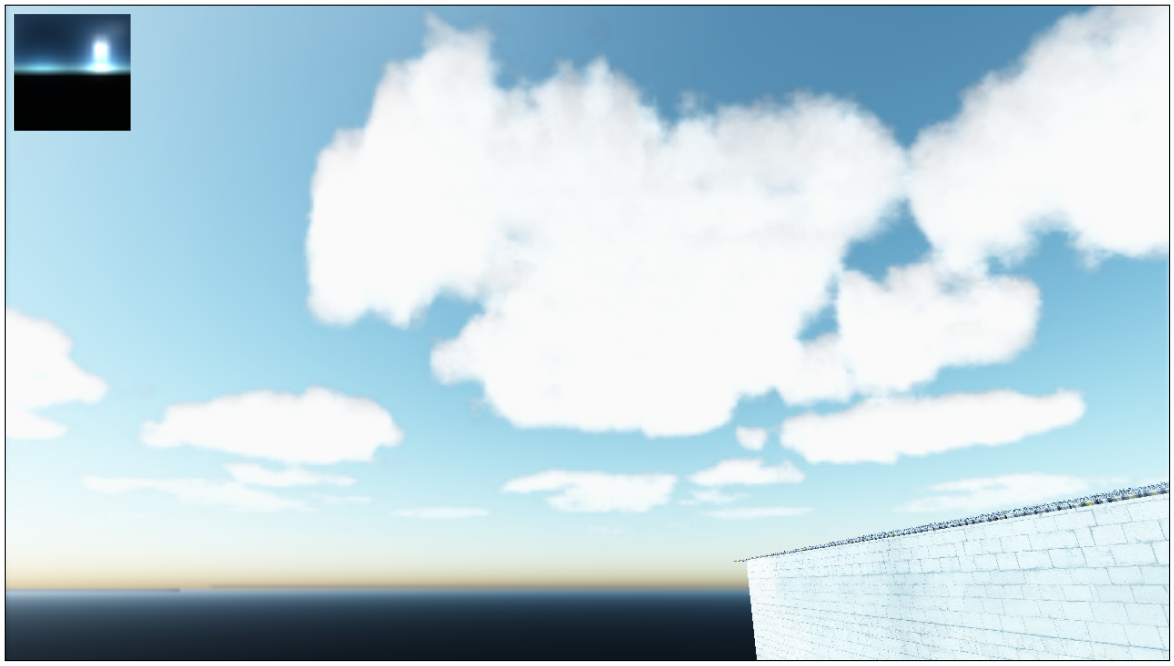


Figure 5.7: Render of clouds with Worey Noise.

5.2 Cloud Rendering

Given the camera position and direction p_{cam}, p_{dir}, point on line segment $p(t) = p_{cam} + p_{dir}t$, t_0, t_1 is calculated so that it intersects the volumetric cloud layer (Fig. 5.8). Due to symmetry, t_0, t_1 can be obtained from function $C(\alpha, r)$, where α is the view zenith angle. $C(\alpha, r)$ is pre-computed and stored in in 2D LUT. S represents the number of ray march samples per ray and s is the distance between ray march samples. Let I be vector containing colour and alpha initialized to $I = (0.0, 0.0, 0.0, 1.0)$. For each sampled point p if the density $d = D(p)$ is greater than zero, the lighting contribution is computed in the following way:

$$E = \beta_{ex}^{M} d \tag{5.6}$$

$$T = e^{-Es} \tag{5.7}$$

$$L = C_{sunlight}(p)p(\theta)(1 - e^{-2Es}) + C_{ambient} \tag{5.8}$$

$$I_{rgb} = I_{rgb} + I_a(L - LT)/max(E, 0.0000001) \tag{5.9}$$

$$I_a = I_a T \tag{5.10}$$

Function $C_{sunlight}(p)$ returns the colour of sun at specific point. The colour of sun

depends at a specific point depends on the colour of sun C_{sun} and attenuation due to the atmosphere. Attenuation is obtained by sampling the transmittance lookup table $T(p)$. The colour of sun is calculated by treating it as a black body with an effective temperature of photosphere of 5772 K. Spectral irradiance is calculated at wavelengths in the interval [390, 830] nm. By using a XYZ colour conversion table, a colour in XYZ format is obtained. A sRGB working space with D65 reference white matrix is used to convert XYZ to RGB. The result is equal to $C_{sun} = (1, 0.8748, 0.8151)$.

$$C_{sunlight}(p) = T(p)C_{sun} \tag{5.11}$$

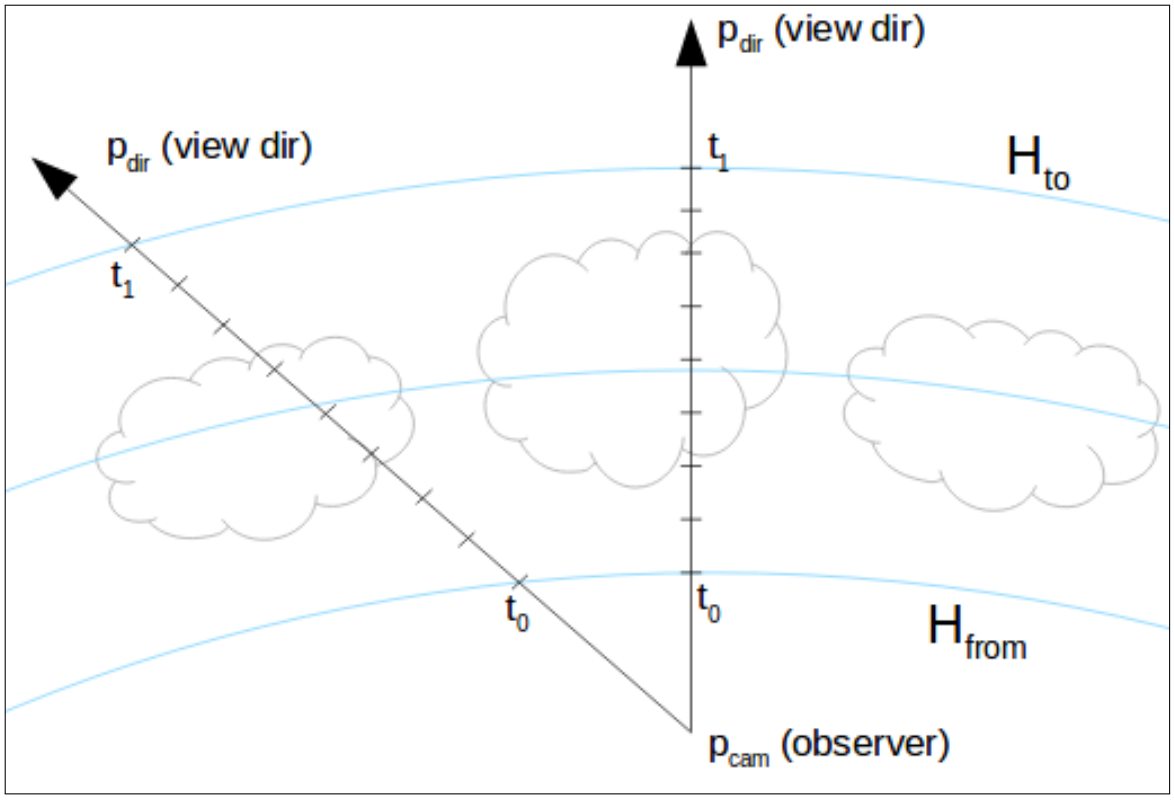


Figure 5.8: Ray marching clouds. Each sample is taken along the viewing direction between inside the volumetric cloud layer.

5.2.1 High Altitude Clouds

Above the volumetric cloud layer cirrus clouds are modelled with Curl noise. A texture with three Curl noises $N_c(x, y)$ is used. Point p is the intersection of ray with high altitude cloud layer, s_s is a scale value and o is an offset which varies with time to simulate cloud movement.

$$
\begin{aligned}
C_a(p) = (C_{sunlight}(p) + C_{ambient}) \\
N_R^c(p_x s_s + o_x, p_z s_s + o_y) \\
N_G^c(p_x s_s + o_y, p_z s_s + o_x) \\
N_B^c(p_x s_s + o_x, p_z s_s + o_x)
\end{aligned}
\tag{5.12}
$$

5.3 Weather Map Generation

The weather map can be supplied an artist or it can be automatically generated (Fig. 5.9). It consists of three channels. Red channel represents the cloud density, 0 being no cloud is present. The green and blue channels represents the cloud height from the middle of the volumetric cloud layer. A random weather map is generated by creating a mesh that represents cloud shape, an ellipse is used. Values such as the number of shapes, their minimum and maximum sizes can be adjusted. Afterwards a noise texture is rendered for the red channel for each cloud mesh. The noise gradually fades at the edge of a cloud mesh. To make the generated weather map seamless, the cloud geometry is rendered multiple times with an offset. This is done with instancing so that one draw call is required to render the weather map.

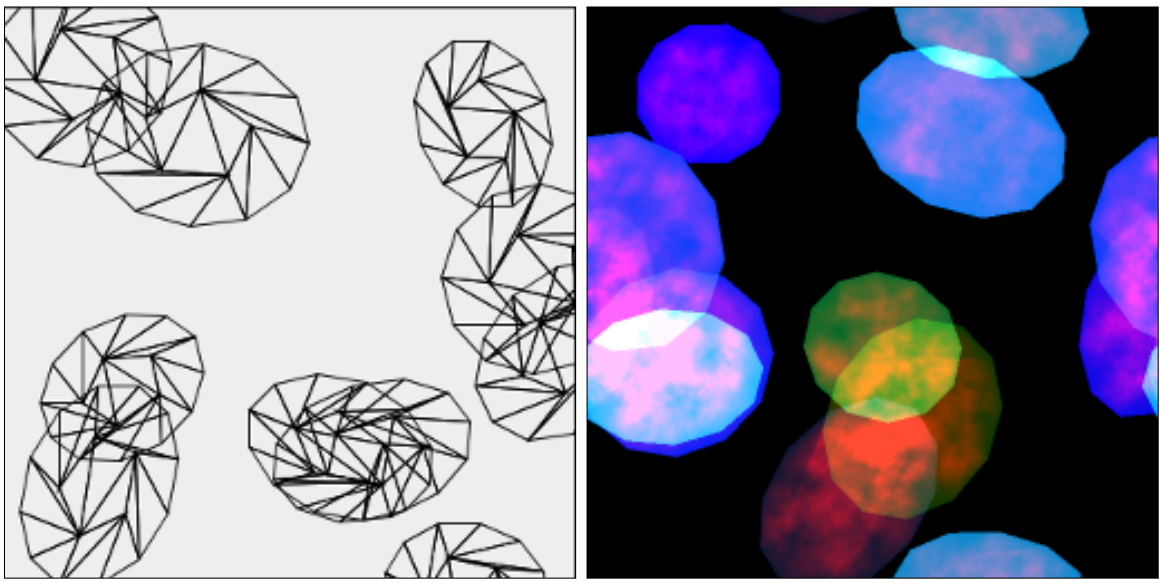


Figure 5.9: Generated weather map (left wireframe).

The data for each separate cloud is preserved for the purpose of cloud animation. The user can change the cloud parameters during frames or move them in a any

direction. The movement of clouds can also be determined by a vector field which can be represented as a 2D texture. Figure 5.10 illustrates the movement of clouds in the weather map based on a vector field.

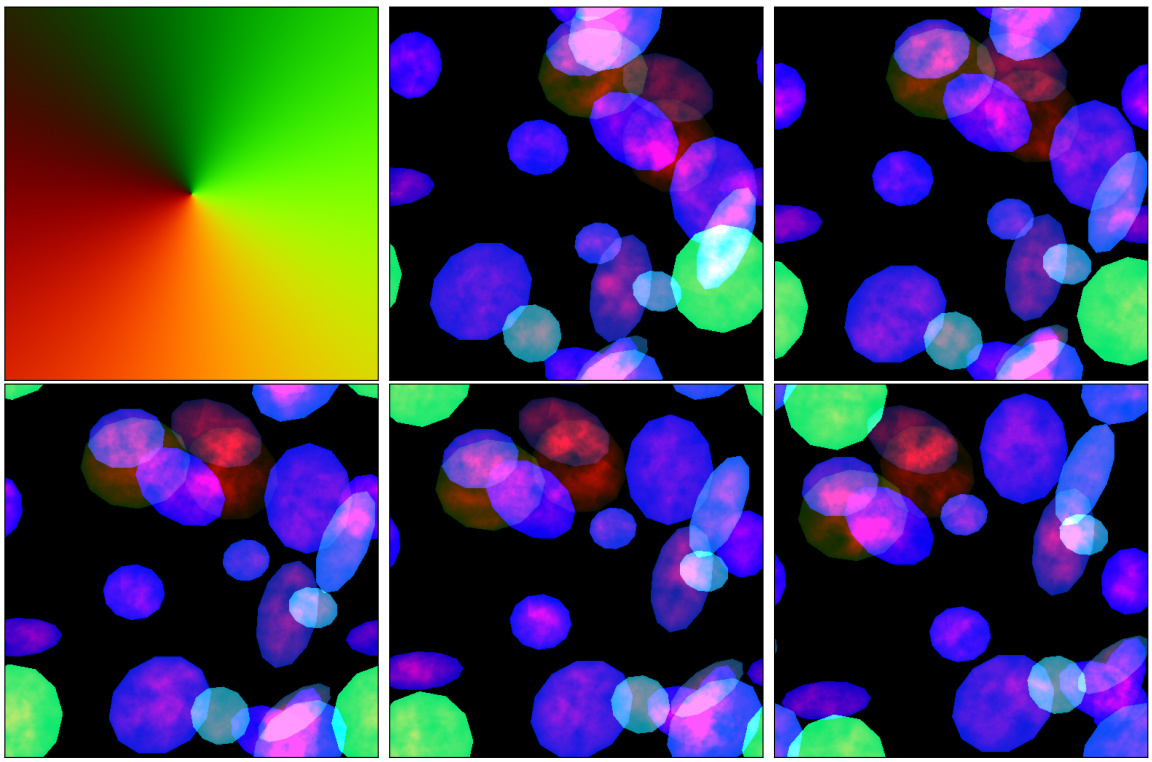


Figure 5.10: Vector field image representing anti-clockwise rotation around center (first image), weather map animation based on the vector field.

5.4 Implementation

The clouds are rendered into texture which contains sky by rendering a quad. By default the sky texture has half the size of desired resolution. S represents the number of ray marching samples. The smaller the value the faster the ray marching will be. Figures 5.11, 5.12, 5.13 and 5.14 show renders with 4, 8, 16, 32 samples respectively.

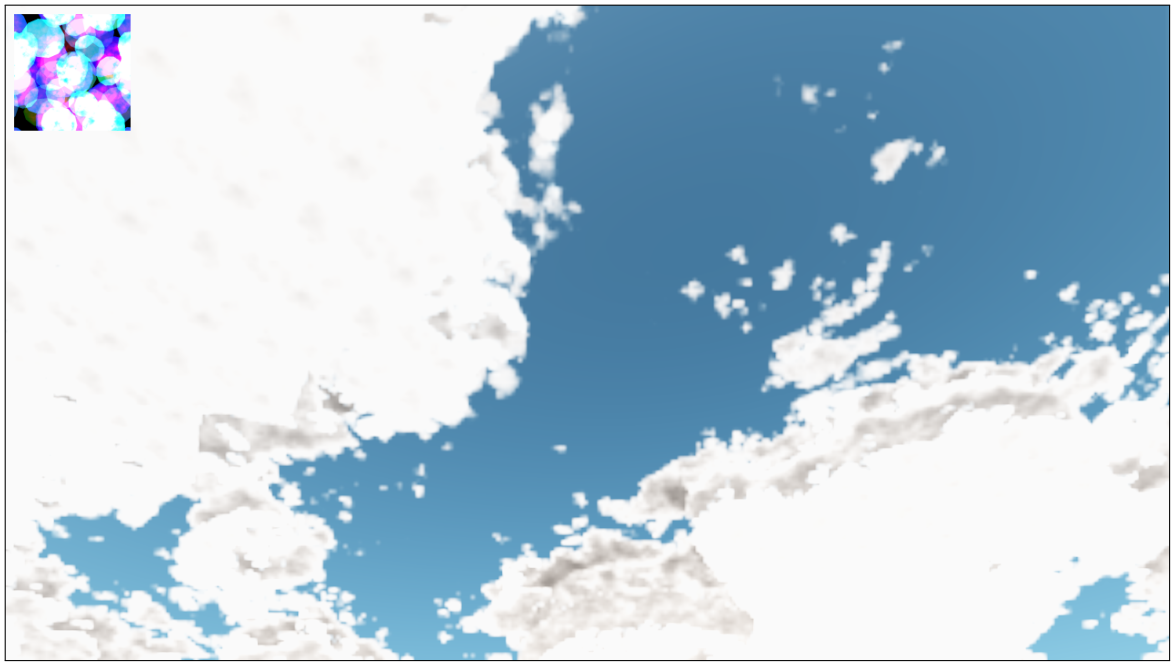


Figure 5.11: Cloud rendered with 4 samples.

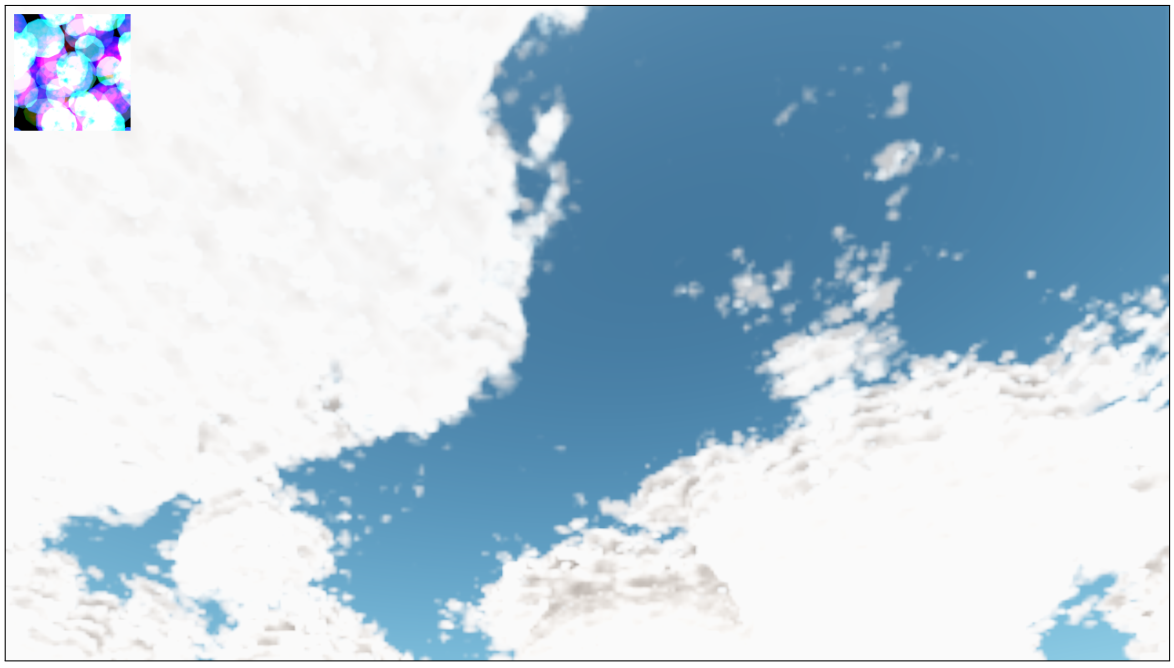


Figure 5.12: Cloud rendered with 8 samples.

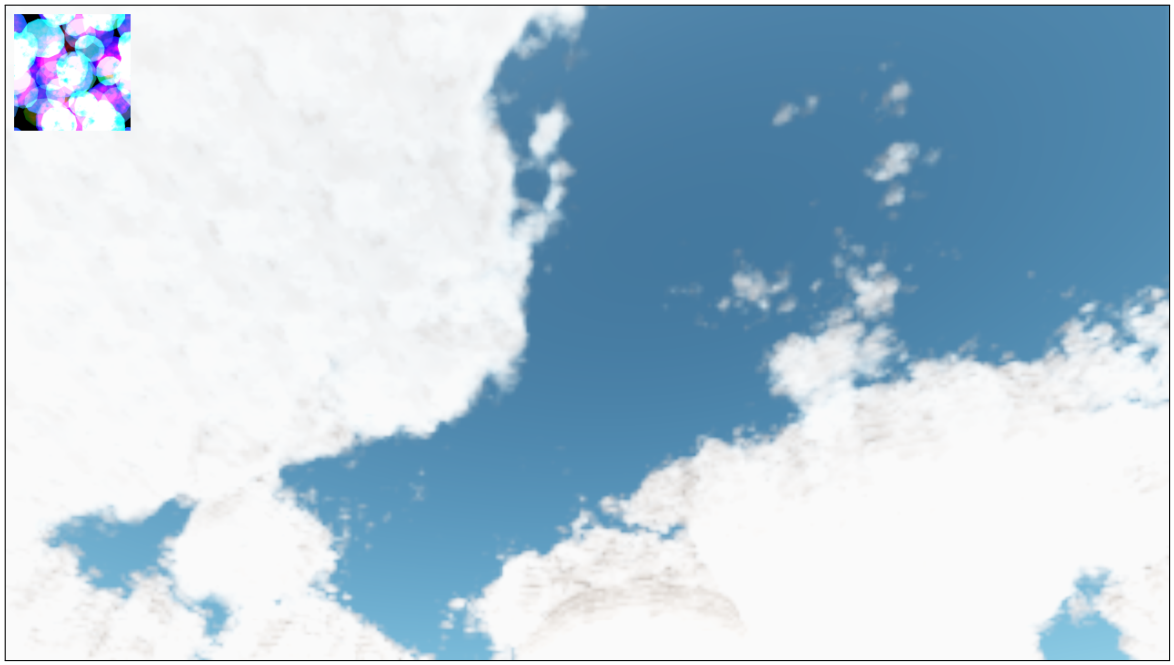


Figure 5.13: Cloud rendered with 16 samples.

Figure 5.14: Cloud rendered with 32 samples.

A closer look at a cloud reveals banding at low number of samples. Banding refers to a visible formation of stripes of contrasting colours. At 16 samples the bands are clearly visible (Fig 5.15). At 64 samples the bands are still visible but not that apparent

(Fig 5.16).

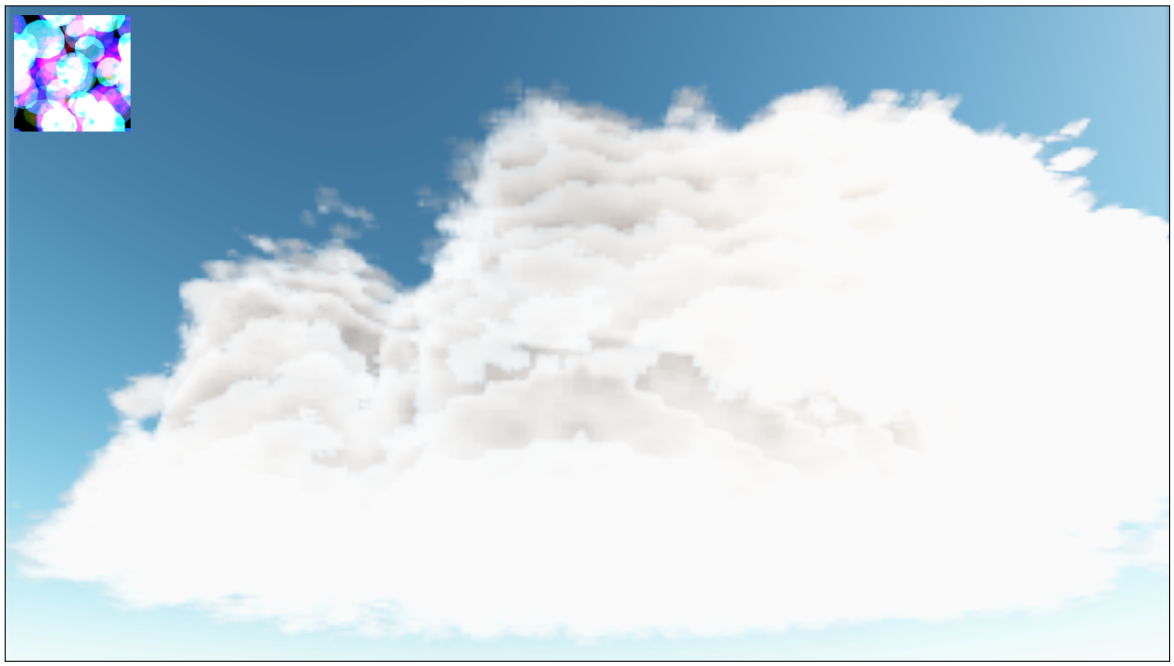


Figure 5.15: Cloud rendered with 16 samples - visible bands.

Figure 5.16: Cloud rendered with 64 samples - visible bands.

5.4.1 Ray Origin Offset

To improve visual quality while using low number of samples, each ray is offset by a fraction of distance between sampled points in the direction of ray (Fig. 5.17). Let f_x, f_y be the current rendered pixel coordinate, then the sampling position is computed as:

$$p(t) = p_{cam} + p_{dir}t + M(f_x\%4, f_y\%4)s \tag{5.13}$$

$$M(x,y) = \begin{bmatrix} 0.0 & 0.5 & 0.125 & 0.625 \\ 0.75 & 0.22 & 0.875 & 0.375 \\ 0.1875 & 0.6875 & 0.0625 & 0.5625 \\ 0.9375 & 0.4375 & 0.8125 & 0.3125 \end{bmatrix} \tag{5.14}$$

Figure 5.17: Cloud rendered with 16 samples per ray with ray offset, bands are no longer visible.

5.4.2 Rainbow

A rainbow (Fig. 5.19) is rendered by using a Mie phase function, which is stored in a 1D texture (Fig. 5.18). The light coming from rainbow is reflected and refracted of

water droplets in the atmosphere. Given a wavelength of light λ and a water droplet of size r, the phase function can be calculated by simulating light interaction with the water droplet. The process involves simulating reflections and refractions of light rays at the boundary of a water droplet. This process was repeated for wavelengths from 390nm to 780nm. Since the index of refraction depends on the wavelength a table of computed index of refractions was used, one for each wavelength. The spectrum was subsequently converted to XYZ colour space and then to RGB colour space with a sRGB working space with D65 reference white.

Figure 5.18: A Mie phase function stored in 1D texture.

Figure 5.19: A render of clouds and a rainbow.

5.4.3 Cloud Soft Shadows

To render cloud soft shadows (Fig. 5.21) for each rendered fragment a shadow value has to be computed. A shadow value of one represents absence of a shadow and one represents presence of a shadow. A value between zero and one represents a shadow formed by filtering some but not all of incoming light. Each fragment has a world position p, a ray is constructed in the direction d towards the light. Intersection point p_i with the volumetric clouds layer is calculated (Fig. 5.20). To obtain a shadow value s the density of clouds is sampled at $D(p_i)$. The shadow value is then calculated by interpolating the density value.

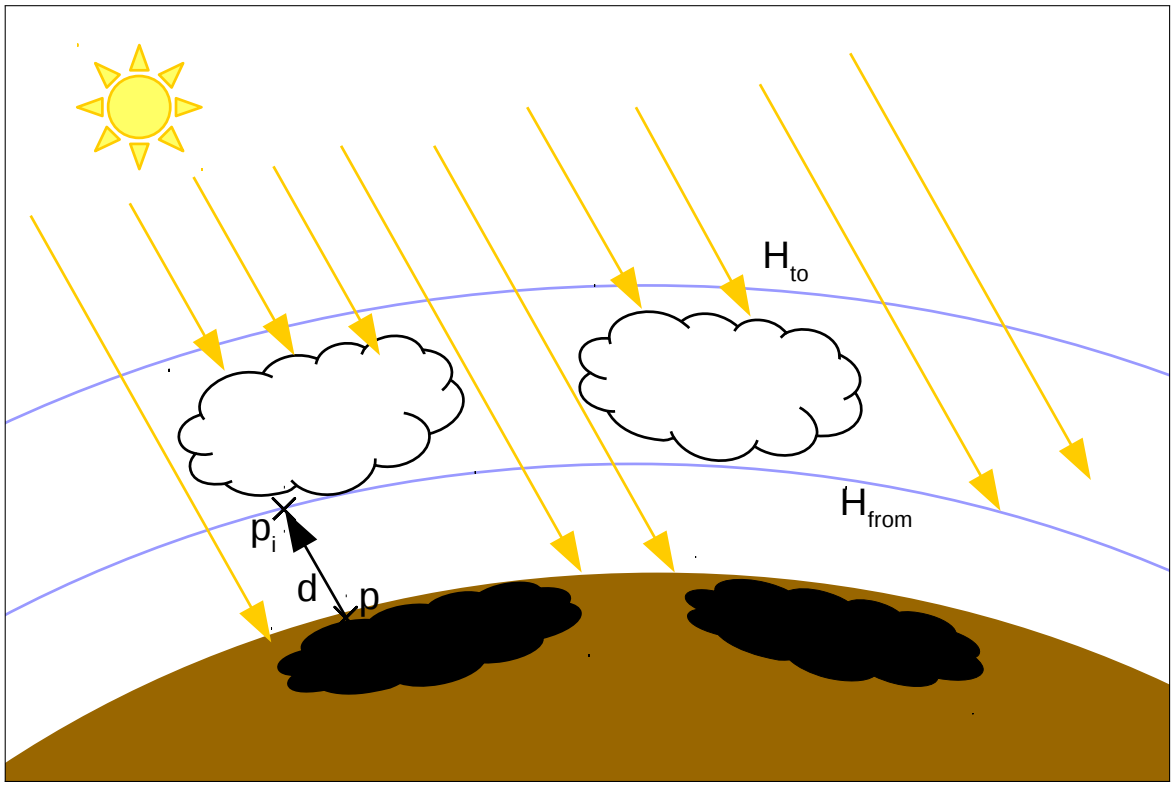


Figure 5.20: Point p is shadowed by the density value obtained at point p_i.

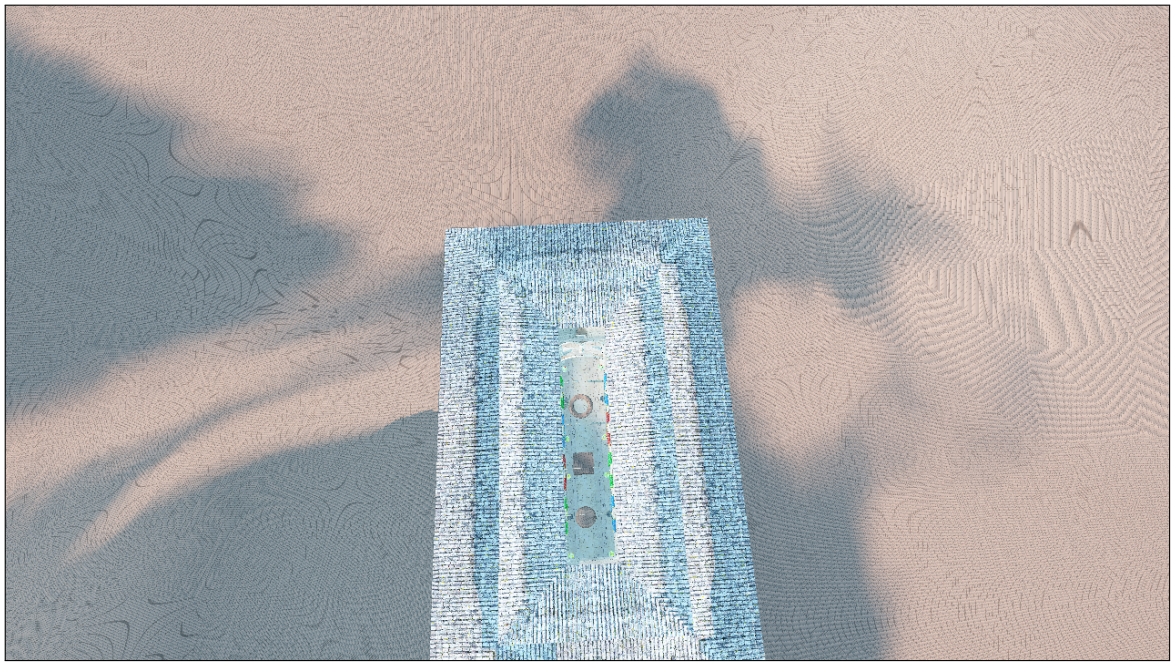


Figure 5.21: A render of the cloud soft shadows.

5.5 Results

Figures 5.23, 5.25, 5.26 and 5.27 show the result of rendering the sky with clouds at different locations and time. The rendering was done with resolution of 1600×900 on Intel HD 4400 which is nowadays considered to be a low-tier integrated graphics card. The sky and clouds were rendered to a texture with half the resolution of the screen. The mip maps for this texture were generated to smooth out the result, which was then copied over the screen. Finally the cloud shadows, sky lighting and atmospheric scattering was computed as a post process effect. Figure 5.22 shows the time taken for each of these actions. The time taken to render clouds depends on the portion of sky that is visible. Figure 5.24 shows the average time taken to render a sky with clouds at different resolutions.

Time	Time[ms]
Sky texture render (800×450)	0.3
Copy to texture (800×450)	0.75
Clouds render (800×450)	2.09
Mip map generation	0.37
Copy to screen	1.78
Cloud soft shadows, Aerial perspective, sky lighting	1.8

Figure 5.22: Average rendering time of sky and clouds in a deferred rendering set-up with fully visible sky for Figure 5.23 with resolution of 1600×900.

Figure 5.23: Rendering of volumetric clouds and high altitude 2D clouds at sunrise (weather map displayed in the upper left corner).

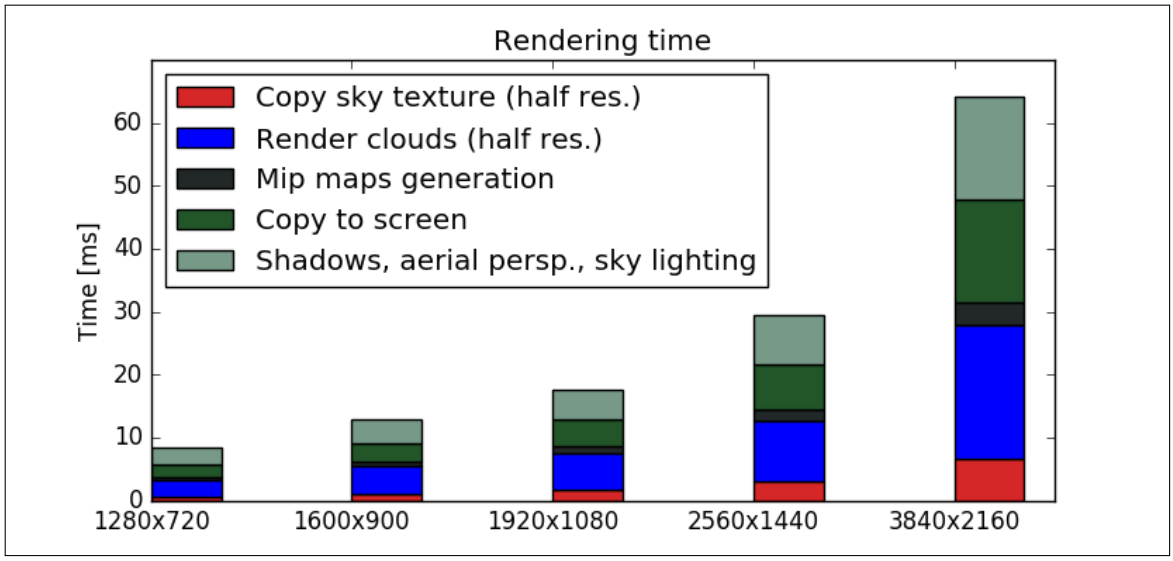


Figure 5.24: Performance of each action shown at different resolutions.

Figure 5.25: A render of clouds with the sun high in the sky.

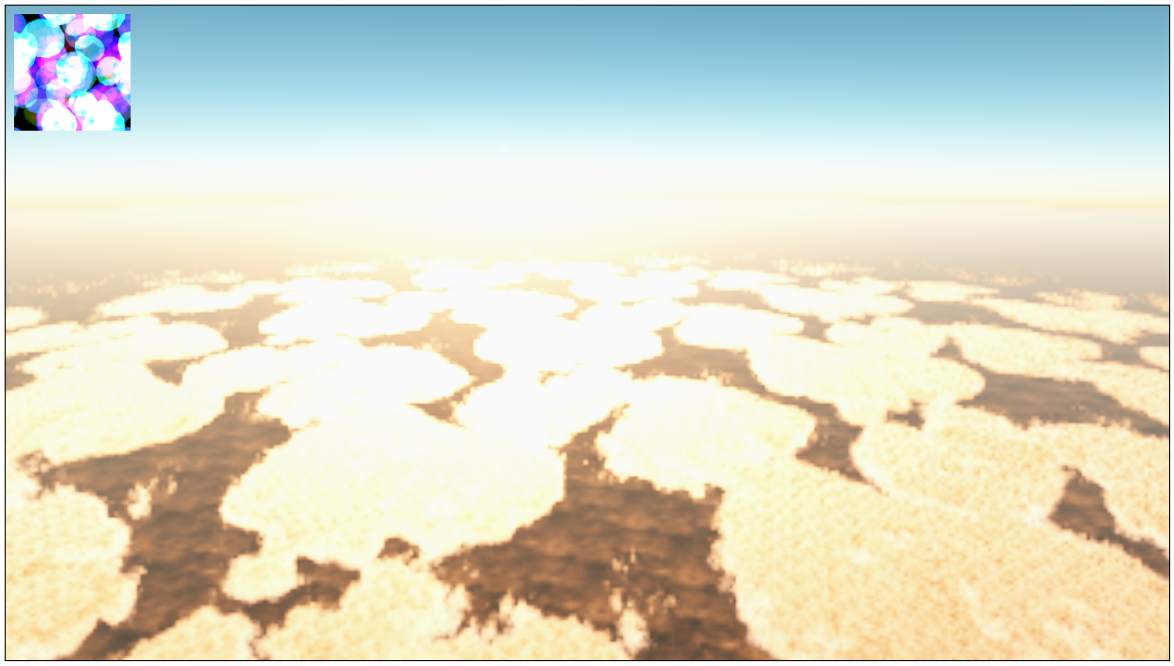


Figure 5.26: A render of clouds viewed from above.

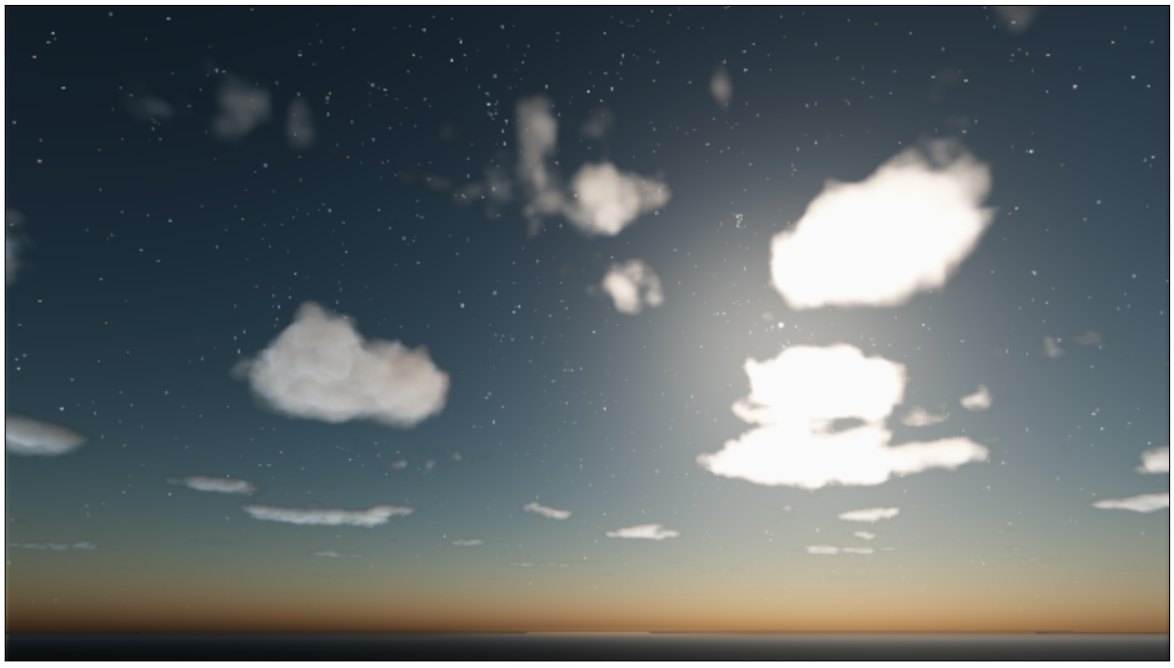


Figure 5.27: A render of a night sky with moon and stars.

5.6 Summary

This chapter presented the technique for rendering volumetric clouds and height altitude clouds. The clouds were rendered as participating media with a density function calculated based on weather map with detail added by a 3D Worley noise texture. High altitude clouds were rendered as with the use of 2D Curl noise. While artists can produce a custom weather map, a weather map generation was presented as an alternative. Other techniques such as rendering of rainbows and cloud soft shadows were presented.

6 SevenSky Library

The sky rendering system is implemented as an SevenSky API for JMonkeyEngine. In this chapter, the design principles and the structure of this library is presented along with the explanation of user configurable variables. Finally, case studies are provided to show the usability of the library for the creation of serious games.

6.1 Overview of the Library

The library is composed of four main packages: core, elements, generators and colour (Fig. 6.1). The package core contains the scene processor SevenSky and an utility class Space. Package elements contains elements which can be plugged into the scene processor. Package generators contains generators for look up tables, weather maps and noise textures. Finally the package colour contains a class useful for colour conversion.

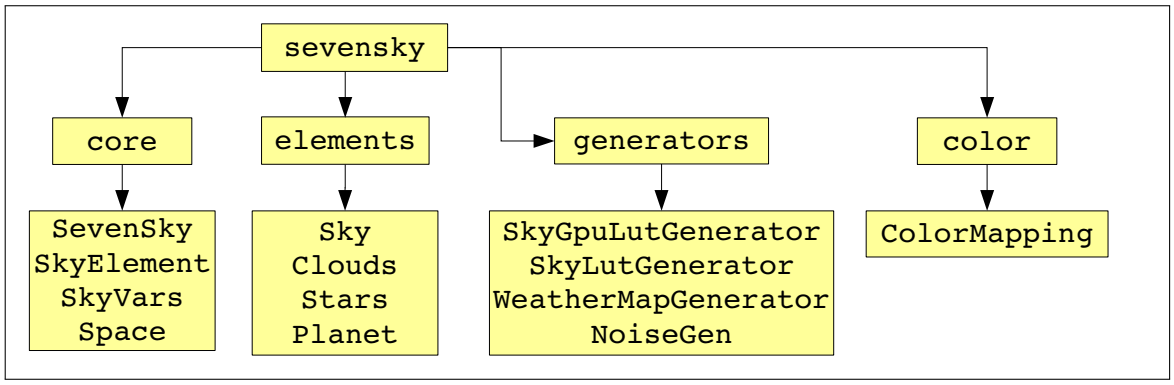


Figure 6.1: Overview of the library.

6.2 Design Principles

Coupling refers to the dependency between modules. It is desirable to minimize coupling so that a change in one module will not require changing the others. This

principle was employed to design a modular and extensible API. The user is able to render a standalone sky, clouds and stars or any combinations of these. Alternatively, the user can provide his own implementation of each of these or introduce a new sky element.

6.3 SevenSky Library Description

The main class of the library is a scene processor SevenSky located in the core package. It is responsible for rendering sky elements which can be the atmosphere, clouds, stars, planets or any other element which implements the SkyElement interface. User configurable values are stored in an instance of class SkyVars as key, value pairs. Each sky element can define keys to be used with SkyVars. If two elements define a key with the same name, then the value is shared between the two. This way multiple sky elements can use a single instance of a lookup table or other resource. The class SevenSky defines two keys (Fig. 6.2).

`PLANET_RAD`	R_g	Planet's radius
`ATMOSPHERE_RAD`	R_a	Atmosphere's radius

Figure 6.2: Variables defined in class SevenSky.

An instance of the scene processor SevenSky is created by providing its constructor an instance of SkyVars. Finally the created scene processor is registered within the JME system (Fig. 6.3).

```
1 SkyVars vars = SkyVars.Earth();
2 SevenSky sevenSky = new SevenSky(vars);
3 ...
4 app.getViewPort().addProcessor(sevenSky);
```

Figure 6.3: Creating an instance of SevenSky and registering it as a scene processor within JME3.2.

6.3.1 Atmospheric Scattering

Class Sky is is responsible for rendering atmospheric scattering. It defines several parameters shown in Figure 6.4 and inherits from SkyElement, thus it can be attached to the scene processor SevenSky in the way presented in Figure 6.5.

RAYLEIGH_SCATTERING	$\beta_s^{R_0}$	Rayleigh scattering at sea level
RAYLEIGH_ABSORBTION	$\beta_a^{R_0}$	Rayleigh absorption at sea level
RAYLEIGH_SCALE_HEIGHT	H_R	Rayleigh scale height
MIE_SCATTERING	$\beta_s^{M_0}$	Mie scattering at sea level
MIE_ABSORBTION	$\beta_a^{M_0}$	Mie absorption at sea level
MIE_SCALE_HEIGHT	H_M	Mie scale height
SUN_MIE_G	g_{sun}	Henyey-Greenstein phase function parameter
MOON_MIE_G	g_{moon}	Henyey-Greenstein phase function parameter
TRANSMITTANCE_LUT	T^{LUT}	Transmittance lookup texture
PATH_LENGTH_LUT	P^{LUT}	Path length lookup texture
SunDir	d_{sun}	Direction vector towards Sun
MoonPos	p_{moon}	Position of moon relative to observer

Figure 6.4: Variables defined in class Sky.

```
1 SkyVars vars = SkyVars.Earth();
2 SevenSky sevenSky = new SevenSky(vars);
3
4 Sky sky = new Sky();
5 sevenSky.add(sky);
6
7 app.getViewPort().addProcessor(sevenSky);
```

Figure 6.5: Adding Sky instance to the scene processor.

6.3.2 Volumetric Clouds and High Altitude Clouds

Class Clouds renders volumetric clouds and high altitude clouds. Its defined parameters are shown in Figure 6.6. Similarly to class Sky, class Clouds also implements SkyElement interface and thus can be added to the scene processor. The order of the

added elements matters, with the first one being rendered first. Figure 6.7 shows the right order of adding Sky and Clouds elements to the scene processor.

`VOLUMETRIC_CLOUDS_FROM`	H_{from}	Altitude of cloud layer lower boundary.
`VOLUMETRIC_CLOUDS_TO`	H_{to}	Altitude of cloud layer upper boundary.
`TRANSMITTANCE_LUT`	T^{LUT}	Transmittance lookup texture.
`CLOUD_PATH_LUT`	C^{LUT}	Cloud path lookup texture.
`NOISE_DETAIL_MAP`	N	Noise detail texture.
`MIE_PHASE_MAP`	M	Mie phase texture.
`HIGH_ALTITUDE_CLOUD_MAP`	N_c	Noise for high altitude clouds.
`WEATHER_MAP`	W	Weather map texture.
`SunDir`	d_{sun}	Direction vector towards the Sun.
`MoonPos`	p_{moon}	Position of the Moon relative to observer.
`SunIrradiance`	C_{sun}	Colour of the Sun.
`CloudOffset`	o	Cloud offset.
`Scattering`	β_M^{Sc}	Scattering coefficient.
`DetailScale`	s_{det}	Detail scale.
`Coverage`	$W_{coverage}$	Volumetric clouds coverage [0, 1].
`RainDensity`	s_{rain}	Rain density.
`AltoCoverage`	s_c	High altitude clouds coverage [0, 1].

Figure 6.6: Variables defined in class Clouds.

```
1  SkyVars vars = SkyVars.Earth();
2  SevenSky sevenSky = new SevenSky(vars);
3
4  Sky sky = new Sky();
5  sevenSky.add(sky);
6
7  Clouds clds = new Clouds();
8  sevenSky.add(clds);
9
10 app.getViewPort().addProcessor(sevenSky);
```

Figure 6.7: Adding Sky and Clouds instances to the scene processor. Sky will be rendered first.

6.3.3 Stars

Class Stars renders stars with star data obtained from a star catalogue. The parameters defined by class Stars are presented in Figure 6.8. Figure 6.9 shows the right order of adding Stars and Sky elements.

`observerLatitude`	obs_{lat}	Observer's latitude
`observerLongitude`	obs_{lon}	Observer's longitude
`julianDate`	$jdtt$	Julian date and time

Figure 6.8: Variables defined in class Stars.

```
1  SkyVars vars = SkyVars.Earth();
2  SevenSky sevenSky = new SevenSky(vars);
3
4  Stars stars = new Stars();
5  sevenSky.add(stars);
6
7  Sky sky = new Sky();
8  sevenSky.add(sky);
9
10 app.getViewPort().addProcessor(sevenSky);
```

Figure 6.9: Adding Stars and Sky instances to the scene processor. Stars will be rendered first.

6.3.4 Planet

The sky element Planet can be used to render planets or other distant objects. Figure 6.10 shows a listing with the creation of a Planet instance at a specific location with a user provided texture.

```
1  Texture2D marsTex = ...;
2
3  SkyVars vars = SkyVars.Earth();
4  SevenSky sevenSky = new SevenSky(vars);
5
6  Planet mars = new Planet();
7  mars.setDistance();
8  mars.setRadius();
9  mars.setDir(new Vector3(0,1,0));
10 mars.setTexture(marsTex);
11 sevenSky.add(mars);
12
13 Stars stars = new Stars();
14 sevenSky.add(stars);
15
16 Sky sky = new Sky();
17 sevenSky.add(sky);
18
19 app.getViewPort().addProcessor(sevenSky);
```

Figure 6.10: Adding Planet, Stars and Sky instances to the scene processor.

6.4 Use Case I - Cultural Heritage

The library have been used to render the sky with a Kalabsha temple model. The temple was appropriately rotated based on its geographical position. Figures 6.11, 6.12 and 6.13 show the results of rendering the temple at sunrise, noon and sunset. Finally, Figure 6.14 shows the same view at different times of the day. A sample code listing used to setup the scene is presented in Figure 6.15. Figures 6.17 and 6.16 show the rendering of Panagia Angeloktisti, a church located in Kiti village in Cyprus.

Figure 6.11: Kalabsha temple viewed in the early morning.

Figure 6.12: Inside of Kalabsha temple during day.

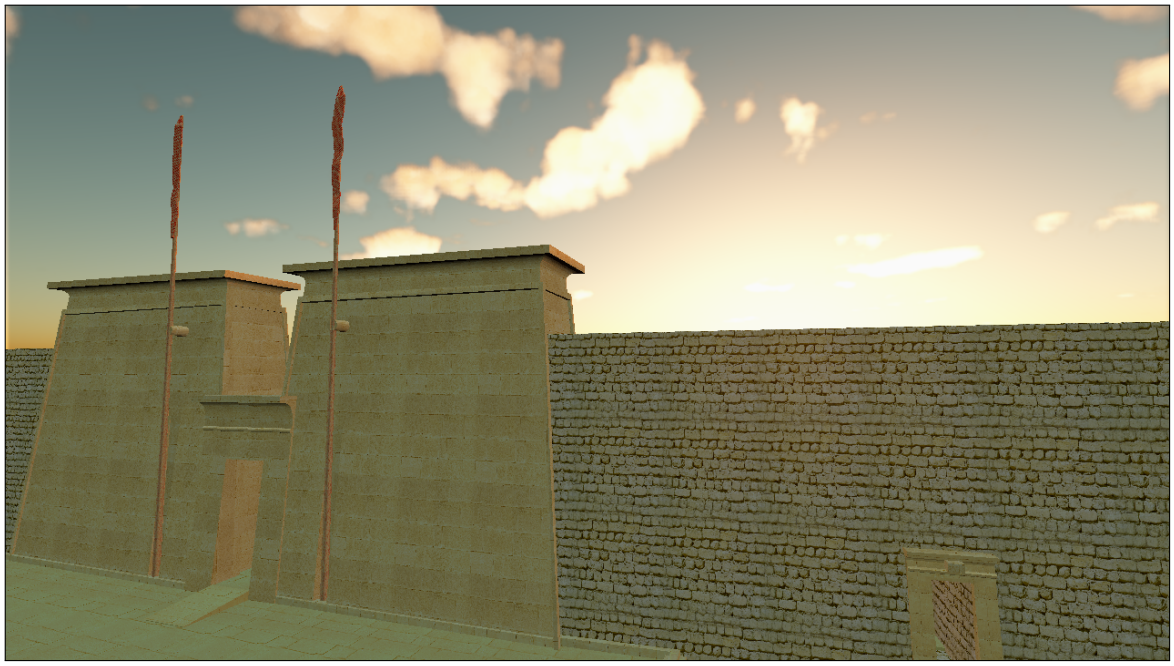


Figure 6.13: Sunset and Kalabsha temple entrance.

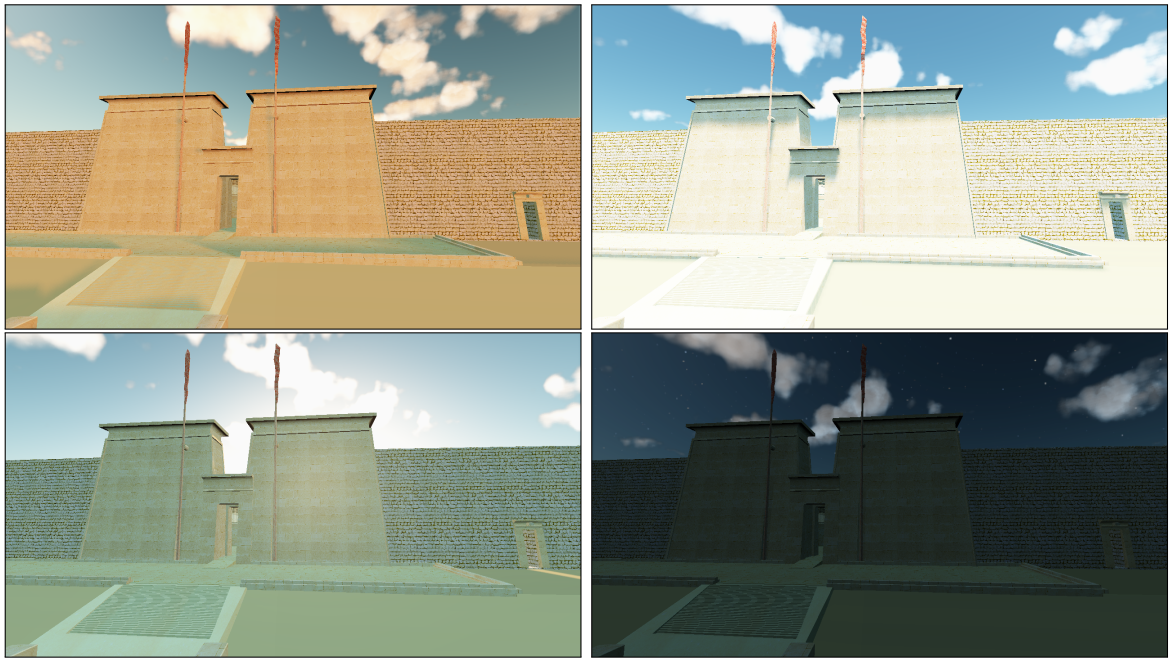


Figure 6.14: A view of Kalabsha temple at sunrise, day, evening and night.

```
AssetManager am = ...;
Spatial model = am.loadModel("Scenes/kalabsha.j3o");
model.rotate(0, -FastMath.DEG_TO_RAD*105, 0);
rootNode.attachChild(model);

SkyVars vars = SkyVars.Earth();
SevenSky s = new SevenSky(vars);

s.add(new Sky());
s.add(new Clouds());

app.getViewPort().addProcessor(s);
```

Figure 6.15: A code listing showing the code which loads and rotates the model of Kalabsha and adds the sky elements to the scene processor.

Figure 6.16: A view of Panagia Angeloktisti church located in Cyprus during day.

Figure 6.17: A view of Panagia Angeloktisti church located in Cyprus at sunrise, day, evening and night.

6.5 Use Case II - Educational Game

The library can be used to create an educational game about the solar system by rendering and appropriately labelling planets and stars of interest. Figures 6.18 and 6.20 shows a night sky of the Moon with the planet Mars and Figure 6.18 shows a night sky with the Jupiter. The code listing is presented in Figure 6.21.

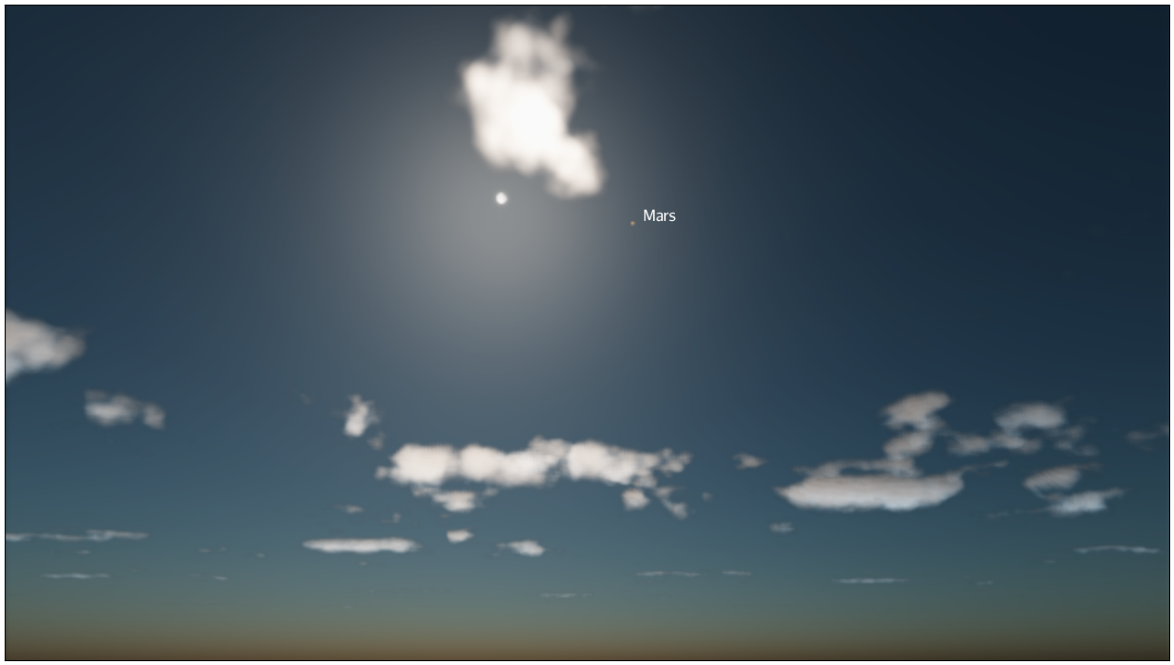


Figure 6.18: A render of a night sky with the Moon and the Mars.

Figure 6.19: A render of a night sky with the Jupiter.

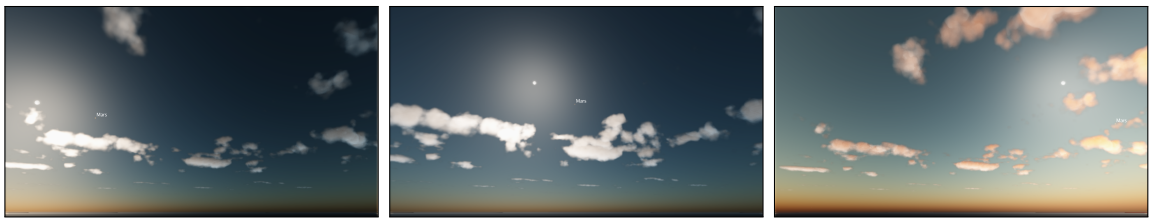


Figure 6.20: A render of a night sky with the Moon and the Mars on the 1nd July 2018 at 11pm and at 1am and 4am on the next day with the observer located in Malta looking in the south-eastern direction.

```
1  Texture2D marsTex = ..., jupTex = ...;
2
3  SkyVars vars = SkyVars.Earth();
4  SevenSky s = new SevenSky(vars);
5
6  s.add(new Planet(marsTex, Mars.instance, Mars.RADIUS_KM));
7  s.add(new SkyLabel("Mars", Mars.instance));
8  s.add(new Planet(jupTex, Jupiter.instance,
       Jupiter.RADIUS_KM));
9  s.add(new SkyLabel("Jupiter", Jupiter.instance));
10 s.add(new Sky());
11 s.add(new Clouds());
12
13 app.getViewPort().addProcessor(s);
```

Figure 6.21: A code listing showing how to add planets, planet labels, sky and clouds to the scene processor.

6.6 Summary

This chapter introduced the SevenSky library which implemented the theoretical concepts presented in this work. The library was designed in a modular and extensible manner with the user able to use or replace existing components. Finally, a use case for the visualisation of historical and cultural heritage sites was presented, with the Egyptian temple of Kalabsha and the Cypriot chapel of Kiti as subjects.

7 Conclusion

The proposed sky rendering technique simulates atmospheric scattering and generates skies capable of simulating daylight, moonlight, sunrise and sunset. The technique, which is aimed at interactive scenarios, has been further optimised through the use of lookup tables, making it a viable proposition for use in games and other applications that need to render physically correct or photorealistic skies. The technique also incorporates volumetric and high altitude clouds, rendered as participating media, with a focus on cumulus and high altitude cirrus clouds. Similarly, via optimisations such as lookup tables and a controlled number of samples per ray, volumetric clouds of adequate visual quality were rendered at interactive rates. Another phenomenon synthesised was the rainbow, which employed a precomputed Mie phase function stored inside a texture, to mitigate the cost of rendering.

Finally, the SevenSky library was designed and implemented; the library is capable of generating physically based skies, animated cloud formations and other phenomena according to the definitions provided above. The library was developed in Java and made use of JMonkeyEngine 3 as an OpenGL wrapper. A number of use cases have been provided to illustrate envisaged usage scenarios for SevenSky.

7.1 Contributions

The contributions of this work can be summarised as follows:

- a survey of the radiometric methods employed in estimating radiance from astronomical bodies depending on observer location, time and date;
- a literature review of offline and interactive rendering techniques for sky models, including participating media such as clouds;
- a novel technique for realistically animating cloud volumes;
- a Java library for interactive visualisation of physically correct skies.

7.2 Limitations

There are a number of limitations to the presented model, such as its use of single atmospheric scattering to reduce computational costs. Simulating multiple scattering, while more computationally expensive, might produce more accurate results, especially during sunrise and sunset periods. Although cloud rendering implements shadow casting, the clouds do not cast shadows on themselves. Self-shadowing clouds add to the realism of a synthesised image, especially when the observer is close to them. A further limitation of the system is that some of the lookup tables have been created with altitudes in the range between sea level and the top of the atmosphere, which is not adequate when rendering the sky from outer space. Thus, new lookup tables would have to be created.

Another limitation of this work is the lack of user studies to assess usability properties of the SevenSky API. Furthermore, the simple use cases presented for SevenSky can only provide a glimpse into the potential of the library.

7.3 Future Work

The current cloud rendering system supports only cumulus and cirrus clouds; since there exist many other variants, the system can be extended to model and render these forms. Furthermore, in terms of phenomena, the system only implements atmospheric attenuation, clouds and rainbows; our model could be extended to simulate different weather conditions at different degrees. For instance, rainy weather could be simulated at varying degrees of intensity, from drizzles to fully raging rainstorms. The system could also be extended to include other commonly observed phenomena such as lightning.

Another avenue for future work is that of extending the visualisation framework to support audio and other environmental noise, such that phenomena like thunder, rain drops and wind could be incorporated, adding to the realism and applicability of the library.

8 Bibliography

[1] Arcot J Preetham, Peter Shirley, and Brian Smits. A practical analytic model for daylight. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 91–100. ACM Press/Addison-Wesley Publishing Co., 1999.

[2] Lukas Hosek and Alexander Wilkie. An analytic model for full spectral sky-dome radiance. *ACM Transactions on Graphics (TOG)*, 31(4):95, 2012.

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

[4] Henrik Wann Jensen, Frédo Durand, Julie Dorsey, Michael M Stark, Peter Shirley, and Simon Premože. A physically-based night sky model. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 399–408. ACM, 2001.

[5] S ONEIL. Accurate atmospheric scattering. gpu gems 2, 2005.

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

[7] Yoshinori Dobashi, Kazufumi Kaneda, Hideo Yamashita, Tsuyoshi Okita, and Tomoyuki Nishita. A simple, efficient method for realistic animation of clouds. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 19–28. ACM Press/Addison-Wesley Publishing Co., 2000.

[8] Antoine Bouthors, Fabrice Neyret, Nelson Max, Eric Bruneton, and Cyril Crassin. Interactive multiple anisotropic scattering in clouds. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 173–182. ACM, 2008.

[9] Mark J Harris, William V Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM*

SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, pages 92–101. Eurographics Association, 2003.

[10] Andrew Schneider and Nathan Vos. The real-time volumetric cloudscapes of horizon: Zero dawn. In *Proc. SIGGRAPH*, 2015.

[11] Theskylive: The solar system at your fingertips.

[12] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[13] David S Immel, Michael F Cohen, and Donald P Greenberg. A radiosity method for non-diffuse environments. In *ACM SIGGRAPH Computer Graphics*, volume 20, pages 133–142. ACM, 1986.

[14] Unity Game Engine. Unity game engine-official site. *Online][Cited: October 9, 2008.] http://unity3d. com*, pages 1534–4320, 2008.

[15] Epic Games. Unreal engine. *Online: https://www. unrealengine. com*, 2007.

[16] J Andersson. Frostbite 2 engine, 2011.

[17] P Bretagnon and G Francou. Planetary solutions vsop87. *Astronomy and Astrophysics*, 202, 1988.

[18] Nachum Dershowitz and Edward M Reingold. *Calendrical calculations*. Cambridge University Press, 2008.

[19] Nicole Capitaine and A-M Gontier. Accurate procedure for deriving uti at a submilliarcsecond accuracy from greenwich sidereal time or from the stellar angle. *Astronomy and Astrophysics*, 275:645, 1993.

[20] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters/CRC Press, 2008.

[21] Andrew S Glassner. *An introduction to ray tracing*. Elsevier, 1989.

[22] Kun Zhou, Zhong Ren, Stephen Lin, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Real-time smoke rendering using compensated ray marching. In *ACM Transactions on Graphics (TOG)*, volume 27, page 36. ACM, 2008.

[23] Kirill Vainer. jmonkeyengine 3.2java opengl game engine, 2018.

[24] Randi J Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL shading language*. Pearson Education, 2009.

[25] TA Nexus. the hyg database, 2014.