Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Miguel Machado Fernandes

Rendering the Sky

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Pedro Miguel Machado Fernandes

Rendering the Sky

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
António Ramires Fernandes

ACKNOWLEDGEMENTS

Finishing this thesis represents the end of the most unique phase of my academic journey, which I could not have completed without the help of multiple people to whom I will always be thankful for.

First, I'd like to thank professor António Ramires Fernandes, for tirelessly and constantly helping me with organizing my time, guiding the my work in the correct direction and reviewing this document.

To Bruna, for constantly being an unwavering compass and confidant, motivating me to always do my best while unconditionally supporting all my decisions. To André, Fábio, Tiago, Tiga and Vítor, for being a group of constant friendship that could never fail to make me laugh.

And finally to my parents and family, for supporting me for 23 years with everything I needed to live a fulfilling life and achieve this goal.

STATEMENT OF INTEGRITY

ABSTRACT

The appearance of the sky is defined by the position of the elements we can see in it. With that in mind, it's interesting to comprehensively understand the procedures needed to accurately compute the position of said elements in the sky, given the latitude, longitude of the observer and the date. Alongside the accurate position of these elements, we can take into consideration the atmospheric scattering and attenuation, as well as taking advantage of our virtual environment to implement non-realistic features that enhance the visualization experience.

Two applications, one for desktop and for Android, were developed using multiple free publicly available tools and resources. The 3 most relevant resources are a star catalogue (Nash), giving us the positions of over 140.000 stars, the set of equations presented in Paul Schlyter's website (Schlyter) which compute the positions of the planets of the Solar System, and the SAMPA algorithm developed by NREL (2012). The latter both computes the position of the Sun and Moon and allows us to translate the position of celestial elements from a geocentric coordinate system to a topocentric one. This thesis mostly focuses on describing how to use these resources to accurately compute the position of and display the stars, planets, Moon and Sun.

Besides the accurate positioning of these elements, atmospheric scattering is also taken into account, during the day, which gives our sky its familiar color, as well as atmospheric attenuation, during the night, which color shifts the planets, Moon and stars depending on their position on the sky. For ease of visualization, non-realistic features were added that increase the visibility of usually imperceptible elements, such as increasing the scale of the planets, increasing the brightness of the dimmest stars and overlaying the artwork of the 88 constellations over the night sky.

KEYWORDS Atmospheric scattering, night sky, stars, real time simulation

RESUMO

A aparência do céu é definida pela posição dos elementos que conseguimos ver nele. Com isto em mente, é interessante perceber compreensivamente os processos para calcular a posição desses elementos no céu, dada a latitude e longitude do observador e data. Para além da posição exata destes elementos, podemos ter em consideração a dispersão atmosférica e atenuação, assim como tirar vantagem do nosso ambiente virtual para implementar funcionalidades não realistas que melhoram a experiência de visualização. Duas aplicações, uma para *desktop* e uma para Android, foram desenvolvidas com o uso de múltiplas ferramentas e recursos disponíveis gratuitamente *online*. Os 3 recursos mais relevantes são um catálogo de estrelas (Nash), que contém as posições de mais de 140.000 estrelas, as equações apresentadas no *website* do Paul Schlyter (Schlyter) que calculam a posição dos planetas do Sistema Solar, e o algorimto SAMPA desenvolvido pelo NREL (2012). Este último calcula a posição do Sol e da lua, assim como nos permite fazer a translação da posição de elementos celestiais de um sistema de coordenadas geocêntrico para um sistema topocênctrico. Esta dissertação foca-se principalmente em descrever como usar estes recursos para calcular com precisão a localização das estrelas, planetas, Lua e Sol. Para além do posicionamente preciso destes elementos, dispersão atmosférica também é tida em conta, durante o dia, fenómeno que dá ao nosso céu as suas cores, assim como atenuação atmosférica, durante a noite, que altera as cores dos planetas, Lua e estrelas dependendo da sua posição no céu. Para facilitar a vizualização, funcionalidades não realistas foram adicionadas para aumentar a visibilidade de elementos normalmente impercetíveis, tais como aumentar a escala dos planetas, aumentar a luminosidade das estrelas menos brilhantes e desenhar a arte das 88 constelações sobre o céu.

PALAVRAS-CHAVE Dispersão atmosférica, céu noturno, estrelas, simulação em tempo real

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

Part I

INTRODUCTORY MATERIAL

1 INTRODUCTION

The night sky has fascinated humans ever since we've been able to look up to it to admire its natural beauty. Tremendous efforts have been put into research of what we can see in the night sky, among starts, other planets and every other body spotted on the cosmos.

To correctly place these celestial elements on screen several resources and tools are required. These resources include star catalogs, such as the "Astronomical Almanac" U.S. Naval Observatory and H.M. Nautical Almanac Office, the GAIA project (European Space Agency, a) or the Hipparchus catalogue(European Space Agency, b). These catalogs provide star coordinates in geocentric coordinates.

On the other hand, some celestial elements can not have their position cataloged. These include the solar system planets and our moon. As opposed to stars, these elements suffer major positional shifts relative to earth.

1.1 MOTIVATION AND OBJECTIVES

Visualizing the sky on a computer screen allows for a degree of virtualization that enhances the experience.

An objective of this work is to gather the information regarding the required steps for accurately displaying, position wise, celestial elements in a computer screen.

The main goal is to provide the reader with the full process of computing the position of these elements for a particular latitude, longitude and date, while at the same time adding some rendering features that are relevant for the rendering of the sky, such as atmospheric scattering and attenuation.

For the positioning of the celestial elements in the sky, we mainly make use of 3 publicly available resources. The first is the SAMPA algorithm, developed and maintained by NREL (2012). This tool computes the position of the sun and moon, as well as the conversion from a geocentric to a topocentric coordinate system, an operation we need to take advantage of our second resource, the HYG star catalogue (Nash), which contains, in a geocentric reference frame, the location of over 140.000 stars. The 3rd resource comes from Paul Schlyter's website (Schlyter), which contains a set of equations to calculate the position of the planets of the our Solar system, also in a geocentric coordinate system.

Alongside the accurate positioning of these elements, during the day atmospheric scattering from the sun is considered, which gives our sky its color, as well as atmospheric attenuation for the moon, planets and stars, which gives them a slight color shift depending on their position in the sky.

To take advantage of the fact that we are working in the virtual world, non realistic features can be implement to enhance the experience, such as altering the scale of the planets so they're visible, increase the brightness of the dimmest stars and visualizing both the stars that belong to a specific constellation, as well the associated artwork.

1.2 THESIS STRUCTURE

In chapter 2 we discuss the current state of the art of the involved fields, as well as the evolution of these fields in recent history. We go over the history of using computer to simulate atmospheric scattering, from it's minutes per frame earlier implementations to the current real time counterparts, as well as the main theoretical concepts of the theme. The different coordinate systems at use along the implementation are also explored, as well as the conversion between them. Finally we explore the current existing, publicly available star catalogs, and why we might choose one over the other.

In chapter 3, we delve into rendering the sky. Specific data flows are specified, along with all the necessary equations and, when applicable, how to use the API's of publicly available code.

2

STATE OF THE ART

Atmospheric scattering is a natural phenomenon resulting of the interaction between light emitted or reflected by celestial bodies and the atmosphere of our planet. This interaction results in the familiar blue sky we can see during the day, and the reddish color, especially near the horizon, we can observe at sunset.

Lord Rayleigh (Strutt, 1871) and Gustav Mie (Mie, 1908) have described the behavior of the light-particles interaction that produces these colors. The core of the computational models around this topic seek to solve the equations presented in these papers, as accurately and as quickly as possible, explored in sections 2.1 and 2.2.

A rendering of the sky also requires the accurate positioning of celestial elements, especially the sun, moon, and stars, which are by far the brightest objects we can see. Over the years astronomers developed increasingly more accurate, and in the case of stars, more widely encompassing, ways of calculating where they would appear on the sky, for a given date, latitude and longitude. The main coordinate systems involved are presented in section 2.3.

2.1 ATMOSPHERIC SCATTERING

In most models, the Sun is considered the only light source that contributes towards atmospheric scattering, as other light sources are too weak to meaningfully contribute towards it. Also, due to the distance between the Earth and the Sun the light rays are practically parallel to each other upon hitting the atmosphere. It is also assumed the light reaches earth's atmosphere unaffected, as it travels trough the void of space.

Scattering occurs when these light rays interact with the particles in the atmosphere. As light of different wave lengths interacts differently with the same particles, scattering has an effect on the color of light that reaches an observer. In 1871, Strutt (Strutt, 1871) described scattering as a photon being deflected into a different direction due to its electromagnetic field hitting an atmospheric particle's electromagnetic field.

Generally, models consider particles of two different sizes: air molecules and aerosols, the latter being much larger. While the air molecules can be found everywhere in the atmosphere, exponentially decreasing

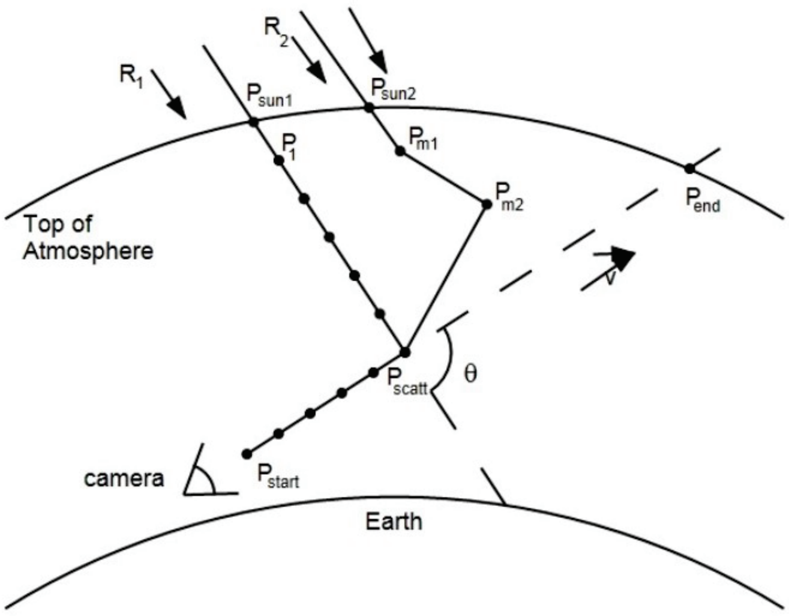


Figure 1: Evaluation of the intensity of a ray along its path. Source: (Lopes and Ramires Fernandes, 2014)

in density with the altitude, aerosols are mostly only found closer to the surface. The result of an interaction between a particle and a photon is mainly dependent on the size of the particle and the wavelength of the photon.

To better clarify the light scattering phenomenon, figure 1 shows a possible path of a ray from the moment it enters the atmosphere until it reaches the camera.

A ray R, a set of photons with a common trajectory, parallel to R_1 and R_2, hits the top of the atmosphere at point P_{sun1} with an intensity I_0. If we follow this same trajectory, we can see that this ray first interacts with a particle at point P_1. This interaction has the potential of deflecting the photons in any direction, meaning only a subset can maintain its initial direction. In this way, the ray gets attenuated. With each interaction the ray gets attenuated further. Eventually, in the given example, after multiple interactions, the ray hits point P_{scatt}. The intensity of the light that reaches point P_{scatt} can be calculated as

$$I_1 = I_0 att(p_A) \tag{1}$$

where p_A is the path between P_{sun1} and P_{scatt}. The value of $att()$ represents the attenuation of light, or optical depth, along the given path. This can also be thought of as the percentage of photons that will follow in that same direction along the given path.

We assume now that at P_{scatt} some of the photons are scattered towards the camera. The intensity of the light leaving this point will be equal to I_2, given by

$$I_2 = I_1 phase(\theta) \tag{2}$$

where phase determines what percentage of the photons will travel in a particular angle θ, in relation to the original direction. The process by which light is scattered towards the camera is called in-scattering.

Between P_{scatt} and the camera the light will suffer further attenuation, meaning that the intensity of the light reaching will be

$$I_3 = I_2 att(p_B) \tag{3}$$

where p_B is the path between P_{scatt} and the camera. The process of attenuation of the intensity of the light along a path is referred to as out-scattering.

The total amount of light reaching the camera in a direction v, as seen in figure 1, can be computed as the sum of all in-scattering contributions caused by rays parallel to R_1, that interact with particles along $\vec{v}$.

2.1.1 *Attenuation function*

Computing the attenuation first requires knowing the particle density. In general, according to (Nishita et al., 1993) the density at a particular height h can be approximated as:

$$\rho(h) = exp(-h/H_{den}) \tag{4}$$

where H_{den} is a reference height, representing the maximum height where we can expect to find particles. For example, in (Nishita et al., 1993), for air molecules $H_{den} = 7994m$ and for aerosol particles $H_{den} = 1200m$. The use of this function reflects the exponential falloff in density as the height increases. The values used for H_{den} reflect the prevalence of aerosols much lower down in the atmosphere.

The attenuation is also affected by the wavelength of light. This makes attenuation a function of the wavelength, λ, and the density, ρ. Equation 5, also from (Nishita et al., 1993), shows how to compute attenuation along a path s, for type T, that can be either air particles (R) or aerosols (M).

$$att_T(s,\lambda) = exp(-\frac{4\pi K}{\lambda^4}\int^s \rho_T(p)dp) \tag{5}$$

where K is a constant for the standard atmosphere, which is the molecular density at sea level, approximately $1.204 kg/m^3$.

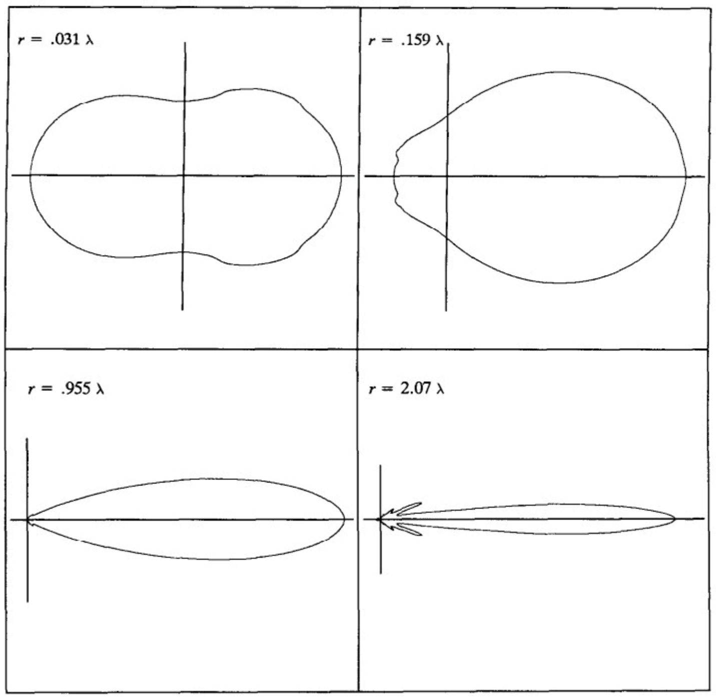


Figure 2: Polar plots of angular scattering functions for spheres of water, for different radii described as a fraction of the wavelength of the incident light. r is the radius of the particle, λ is the wavelength of the incident light. Source - (Klassen, 1987)

From this equation we can see that light with longer wavelengths, like red and yellow, are less attenuated. In turn, light with shorter wavelengths, like blue and purple, are more strongly attenuated.

2.1.2 *Phase function*

The phase function describes how light interacts with particles, when it comes to a possible change in direction. In simple terms, it describes the probability of light being scattered in a particular direction after interacting with a particle. It is usually visualized as seen in figure 2.

As we can see in said figure, the scattering of light greatly depends on the relation between the size of particle and the wavelengths of the incident light. When the radius of particle is much smaller than the wave length of light, as seen in the top left of figure 2 it is scattered mostly forwards and backwards. As the size of the particle grows, more and more of the light is scattered directly forwards.

There are multiple formulations of the phase function. Gustav Mie (Mie, 1908) presented a function that works for all particle sizes, but is computationally expensive. Rayleigh presented an approximation (equation 6) that works for small particles, about an order of magnitude smaller than the wavelength of

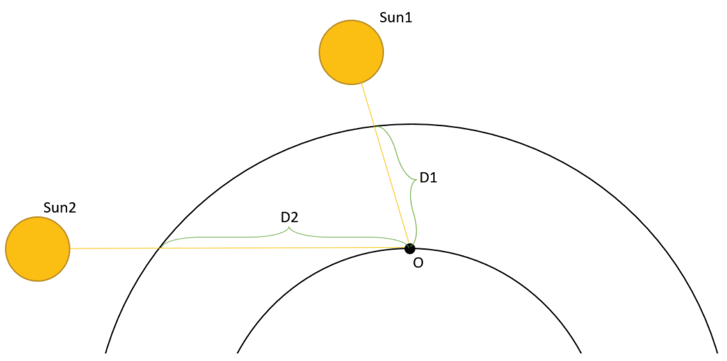


Figure 3: Representation of the difference in angles that creates the different colors of the sky

visible light, like that of air molecules, which gives a similar result to the top left image of figure 2, confirming the equation matches real life results (Nishita et al., 1993).

$$phase_R(\theta) = \frac{3}{4}(1 + cos^2(\theta)) \tag{6}$$

For larger particles, a common approach, used also by (Nishita et al., 1993), is the Henyey-Greenstein phase functions as an approximation to Mie's phase function for larger particles (equation 7).

$$phase_M(\theta) = \frac{1}{4\pi}\frac{1 - g^2}{(1 + g^2 - 2gcos(\theta))^{\frac{3}{2}}} \tag{7}$$

This equation favors scattering most of the light forward, which also aligns with real life results.

2.1.3 *The colors of the sky*

The two previous concepts are responsible for the colors of the sky, as we see them everyday.

Consider the sun in 2 different positions in relation to the Earth, as we can see in figure 3. *Sun1* is in a position the would closely match noon for the observer sitting at point *O*. And *Sun2*, where the sun's rays are practically parallel to the observer's horizon, a sunset.

It is easy to intuitively see that D1 is considerably shorter than D2. As discussed in the two previous sections, shorter wavelengths are scattered more heavily. It follows then that blue light is scattered the most heavily, as the sun doesn't emit a significant amount of violet light which would be scattered even more heavily. The sky gains a blue coloration as light of that wavelength ends up coming from every direction.

However, the distance D2 is big enough that that same blue light is almost completely scattered by the time it reaches the observer. This makes it so only the light with larger wavelengths, closer to red in color, can reach the observer, coloring the sky with the red hues of a sunset.

2.2 ATMOSPHERIC SCATTERING ALGORITHMS

This section is mostly based in (Lopes and Ramires Fernandes, 2014).

The earliest models, before the turn of the millenium, suffered from the lack of computing power. Especially before the birth of programmable GPU's, which excel at parallel computations, the prospect of real time rendering was not even in the horizon. Even then, some simplifications were often made to speed up the process to a reasonable amount of time.

More recently, the basic algorithm used for the calculation has become essentially unchanged, any improvements coming from implementations in new technologies or small changes to the small building blocks of the algorithm, like, for example, a more efficient way to compute a certain calculation.

2.2.1 *Klassen Two Layer Model*

Victor Klassen (Klassen, 1987) proposed a model composed of 2 layers, each of constant density. Furthermore, only the bottom most layer, closest to the Earth, contains particles of the same size or bigger than the wavelength of light, subject to Mie scattering. Klassen refers to these bigger particles as "haze". The top layer is assumed to have only air molecules. Finally, it is assumed that the majority of the light is scattered only once and that an in-scattering event is very unlikely to happen in the top layer of the atmosphere.

It's of note that both distances d_2 and d_3, as seen in figure 4, could be 0, which would effectively make it a single layer model, with no haze, where the light would only be attenuated along distance d_1.

With the assumption the in-scattering will only happen in the haze-filled hair, the attenuation in the top-layer is proportional to the distance d_1, as shown in figure 4. Klassen describes this attenuation in equation 8, where I_0 is the intensity of the light before entering the atmosphere and $\beta^{\lambda}_{ex_m}$ is the extinction coefficient for light of wavelength λ for Rayleigh scattering, indicated by the *m* in the subscript, for molecular scattering.

$$I_1 = I_0 exp(-\beta^{\lambda}_{ex_m} d_1) \tag{8}$$

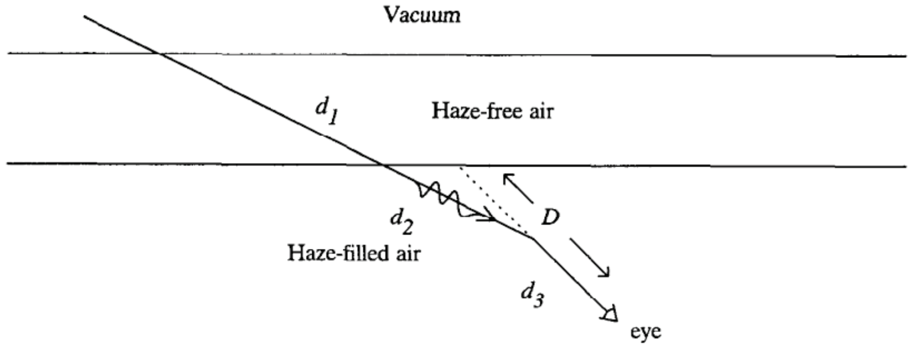


Figure 4: Model proposed by Klassen. Source Klassen (1987)

After reaching the haze-filled air, the light will travel the distance d_2, this time being subject to both Rayleigh and Mie scattering, before reaching the scattering point, as described in equation 9.

$$I_2 = I_1 exp(-(\beta^{\lambda}_{ex_m} + \beta^{\lambda}_{ex_p})d_2) \tag{9}$$

The subscript *p* indicates scattering by particles.

At this point, a percentage of the light will be scattered towards the eye at angle ψ. The intensity of the light leaving the scattering point is described in equation 10.

$$I_3 = I_2 \beta^{\lambda}_{sc}(\psi) \tag{10}$$

β^{λ}_{sc} being the phase function.

Finally, the light will have to travel distance the distance d_3 to reach the observer, which scatters the light in the same way as we have seen in equation 9.

$$I_4 = I_3 exp(-(\beta^{\lambda}_{ex_m} + \beta^{\lambda}_{ex_p})d_3) \tag{11}$$

Substituting back and grouping terms yields

$$I_4 = I_0 \beta^{\lambda}_{sc}(\psi) exp(-\beta^{\lambda}_{ex_m} d_1 - (\beta^{\lambda}_{ex_m} + \beta^{\lambda}_{ex_p})(d_2 + d_3)) \tag{12}$$

which gives us the amount of light coming from one possible path. To get the total amount of light reaching the eye, we have to integrate over D as seen in figure 4, which in turn will mean integrating over all different possible values of d_2 and d_3. If we assume a simplified view of how light works, we can

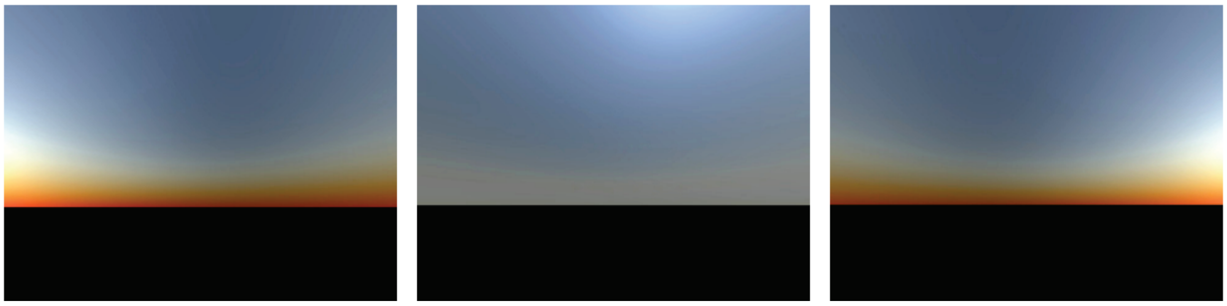


Figure 5: Images using Klassen model of the atmosphere with 2 layers. Sunrise, Midday and Sunset, respectively. Source: Lopes and Ramires Fernandes (2014)

imagine parallel beams of light that each intersect with the segment of length D at each of its points, each one scattering light towards the viewer. Adding all these will give us the total amount of light.

$$I = I_0\beta^{\lambda}_{sc}(\psi)\int_0^D exp(-\beta^{\lambda}_{ex_m}d_1 - (\beta^{\lambda}_{ex_m} + \beta^{\lambda}_{ex_p})(d_2 + d_3))\, dd_3 \tag{13}$$

Klassen's model can reproduce a cloudy or foggy day, by adjusting the particle density and the height of the haze layer.

An example of the sky rendered with this method can be seen in figure 5. It's not too hard to see that this method is not particularly accurate. The difference in the colors of the sky along the day is almost non existent, and the color shift as someone looks further away from the horizon is very unpronounced. The orange tint of the sunrise and sunset also don't look very natural.

2.2.2 *Continuous Atmosphere models*

Kaneda et. al. (Kaneda et al., 1991) implemented single scattering with a continuous atmosphere. Having a non-uniform atmosphere makes it so at each point of the approximation along the view ray, the atmosphere density needs to be calculated, making the approximation, in turn, closer to being physically accurate.

Besides, where as in Klassen's model the attenuation over d_1 and d_2 was calculated with equation 8, with a continuous atmosphere they can also be calculated with an integral for more accurate results. Note that with an atmosphere of constant density, the integral simplifies into the equation Klassen used.

It is also worth noting that this introduces a huge spike in the computing effort required for each ray.

(Nishita et al., 1993) took it upon himself to introduce several performance improvements. The density and the attenuation are, performance wise, the two major computation of these algorithms. In this paper, Nishita introduces a way of speeding up the density calculations by using a pre-calculated look-up table. The atmosphere is assumed as multiple spherical shells. The radius of each sphere is set so that the

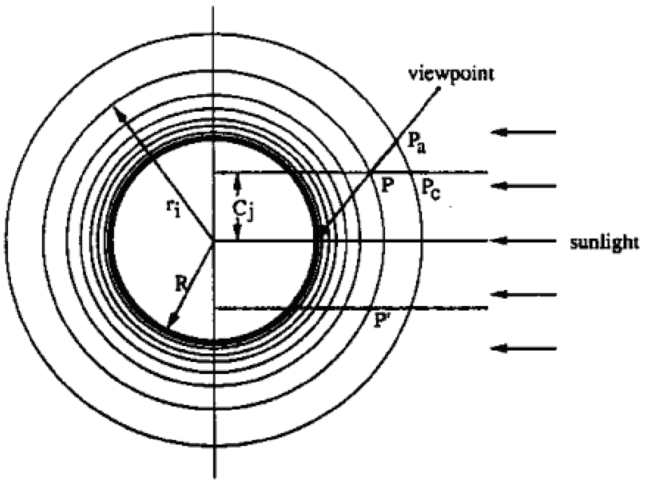


Figure 6: The atmosphere divided into a number of concentric spheres - Source (Nishita et al., 1993)

difference in density between each adjacent sphere is a given value. As a result, for lower altitudes the difference in radii between adjacent spheres is smaller than the difference for high altitudes, as seen in figure 6. This is important as the density of the atmosphere falls off exponentially with height, and therefore numerical integration errors become large with a constant interval.

The sampling points used in the integration are the intersection points between the view ray and these spheres, which are easily computed as the intersection of a line and sphere. In turn, it is easy to look up the density at these points which matches the lookup table for the density of that particular sphere, indexed by altitude.

The optical depth between the sun and an arbitrary point can be calculated in the following fashion. Consider a cylinder, with radius C_j that passes trough the center of the earth and is perpendicular to the light direction, as seen in figure 6. The optical density at the intersection of the cylinder with each sphere is equal (eg., P and P' in figure).

The optical depth at the intersections between multiple cylinders of radius C_j and the spheres with radius r_i is stored in a lookup table, a 2D array $[C_j, r_i]$. The optical density for an arbitrary point P can be calculated by first calculating the cylinder *j* and sphere *i* that include P. The optical depth can be calculate by linear interpolation from $[C_j, r_i]$, $[C_{j+1}, r_i]$, $[C_j, r_{i+1}]$ and $[C_{j+1}, r_{i+1}]$.

This optimization provides a significant reduction in computing when compared to (Kaneda et al., 1991), albeit still slower than (Klassen, 1987). That said, it is more physically accurate than this older model.

As we can see in figure 8, this model is considerably more accurate than Klassen's. The difference in the color of the skies as the time of day changes is more obvious, and the transition to the orange tinted

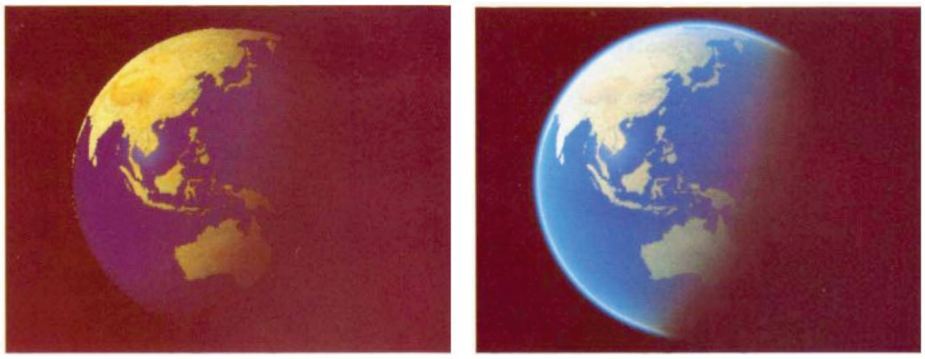


Figure 7: Two pictures of the Earth rendered using (Nishita et al., 1993) model, without (left) and with (right) atmospheric scattering enabled. Source - (Nishita et al., 1993)

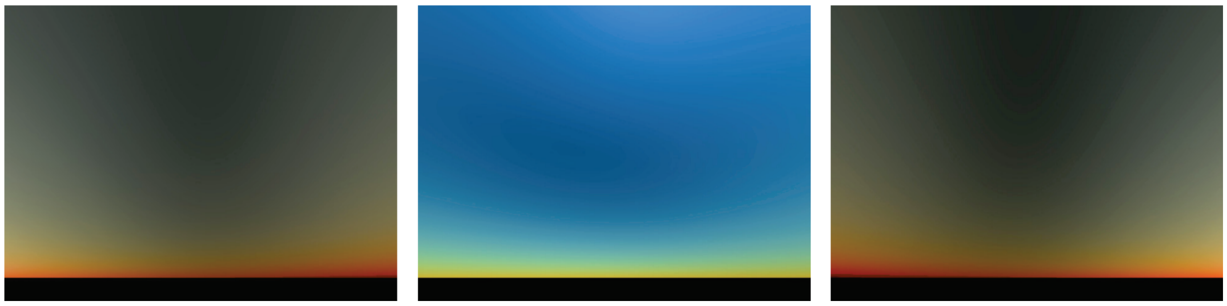


Figure 8: Images using Nishita model of the atmosphere in clear sky, Sunrise, Midday and Sunset respectively. Source: Lopes and Ramires Fernandes (2014)

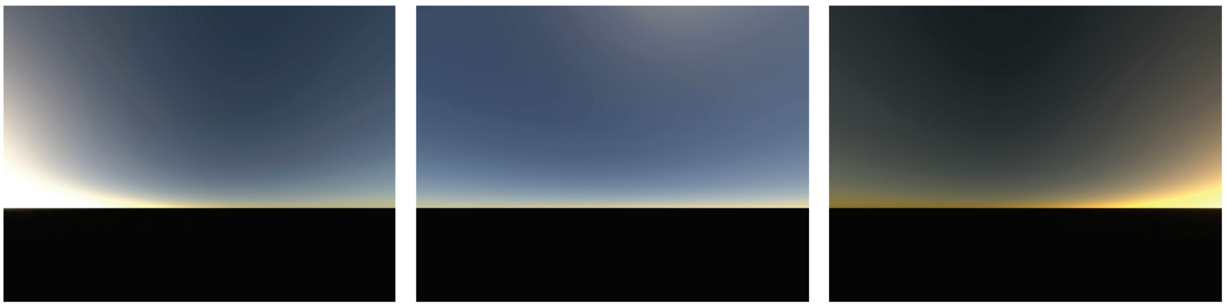


Figure 9: Images using Nishita et al. model of the atmosphere in clear sky with multiple scattering. Source: Lopes and Ramires Fernandes (2014)

horizon at sunrise and sunset looks a lot more natural. The difference in hues along the sky at a given time, for example, mid day, is noticeable more realistic.

2.2.3 *Nishita's multiple scattering model*

Following from his previous work, Nishita et al (Nishita et al., 1996) proposed a model that integrated multiple scattering to compute the colors of the sky.

To do so, the atmosphere is divided into a number of volume elements, commonly referred to as voxels. First, using the same single scattering method from before adapted to work with voxels, the intensity of light in each voxel is calculated and stored.

While calculating the intensity of a sample point, in addition to to the intensity of the light in the corresponding voxel, second order scattering is added to it. From the sample point, sampling rays are fired in a number of directions directions. For each of these sampling rays, the single-scattering algorithm is applied, taking into account distance, angle and density as usual. In this manner, for each sampling point, light that would come from the location of other voxels in the atmosphere after being scattered contributes to the light intensity of that particular point.

The process can be done recursively for higher order scattering, but as the authors note in the paper, the significance of the contributions after the second order scattering is negligible.

This model results in an even more accurate result, close to the theoretical model. However, as expected, the computing power required goes up exponentially when compared to the single-scattering model.

We can see in figure 9 that, in addition to the improvements already mentioned in the previous section, the general hue of the sky is noticeably more realistic.

2.2.4 *Analytical models*

While physically intuitive, the previous models have the major drawback of requiring a lot of computational time to produce a result. Especially at a time where programmable GPUs didn't exist yet, these models were not practical. To try and fix this, Dobashi et al. (Dobashi et al., 1997) and Preetham et al. (Preetham et al., 1999) proposed a set of analytical function that could calculate the color of the sky, given a viewing direction and the position of the sun.

Dobashi et al. proposed to use a sum of basis functions to calculate the intensity of light in sky, particularly, cosine functions. The expression is composed of two terms, dependent on the wavelength being calculate and the azimuth and altitude of the sun. As the azimuth of the sun is assumed to be 0 for any calculation, this value becomes the view direction. In a broad sense, the first term ($\Psi_i(v)$) can be thought of as the intensity of the light for the entire sky for that particular view direction, which is a basis function, and the second term ($W_i^{(\lambda)}(\phi_{sun})$) a weighting factor for that basis function dependent on the altitude of the sun.

$$L_\lambda(v, \phi_{sun}) = \sum_{i=0}^{N} W_i^{(\lambda)}(\phi_{sun})\Psi_i(v) \tag{14}$$

The number of basis function required, N, and parameters of these two terms were approximated from the results of a high quality ray tracing model. After computing the values of the light intensity with the ray tracing model, the functions were modeled to fit these results.

The usage of this method, as expected, produces less realistic results. However, it speeds up the rendering process, as there is no need to solve the integrals by sampling the ray's path, and can instead use a simpler analytical expression.

(Preetham et al., 1999) followed up (Dobashi et al., 1997) idea, replacing the basis with a set of function proposed by Perez et al. (Perez et al., 1993). In this model, Preetham also introduced a concept called *turbidity* that represents the relationship between scattering done by molecules as opposed to haze, which can be thought of as the thickness of the atmosphere, which can be useful to model, for example, a foggy day.

As mentioned, to computed the distribution of sunlight, Preetham et al. use the equations suggested by Perez et al, with the angles depicted in figure 10.

$$F(\theta, \phi) = (1 + Ae^{B/cos\theta})(1 + Ce^{D\phi} + Ecos^2\phi) \tag{15}$$

This formula calculates the intensity of sunlight based on the direction of the sunlight and 5 distribution values, those being darkening and brightening of the horizon (A), luminance gradient near the horizon

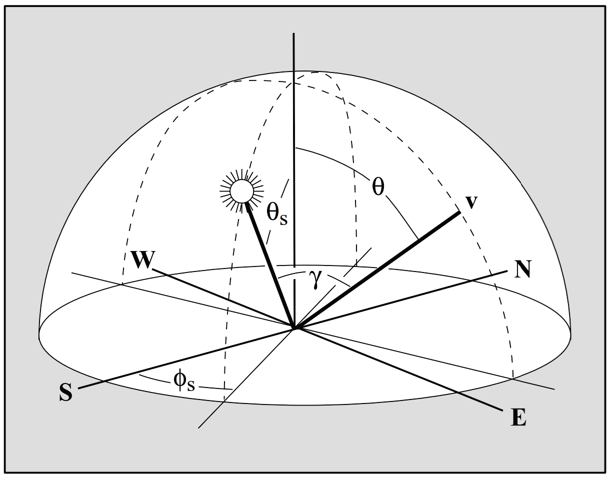


Figure 10: The coordinates for specifying the sun position and direction v on the sky dome. Source: (Preetham et al., 1999)

(B), relative intensity of the circumsolar region (C), width of the circumsolar region (D), and relative back scattered light (E). These distributions values depend on turbidity, and can be found on Appendix A of (Preetham et al., 1999).

Similarly to (Dobashi et al., 1997), (Preetham et al., 1999), used the results of a high quality ray tracing model, in this case, the model of (Nishita et al., 1996) with multiple scattering to model these parameters.

With the sunlight intensity calculations taken care of by these functions, it is necessary to calculate the attenuation in each viewing directions. For this effect, (Preetham et al., 1999) use a set of pre-computed values for the fraction of light that is scattered into the camera ($S(\theta, t)$) and light that is scattered out and "lost" ($B(\theta, \phi, s, t)$), where *t* is the turbidity and *s* is the distance from the camera and the object (or the top of the atmosphere if no object is intersected).

This model could obtain results similar to the ones with similar to the quality of (Nishita et al., 1996), while being orders of magnitude faster.

This model was later analyzed in (Zotti et al., 2007). Comparing the results obtained by using (Preetham et al., 1999) model with actual measures and more physically accurate models. It was concluded that for values of turbidity below 2.0 and above 10.0 the results are not correct. This is probably due to the fact that the data fitting done by (Preetham et al., 1999) to (Nishita et al., 1996) model was only done for value between 2 and 6.

As a consequence of this analysis, a revised model was proposed in (Hosek and Wilkie, 2012), claiming to improve the rendition of sunsets and setups using high turbidity values. From a mathematical point of views these models are quite similar, with (Hosek and Wilkie, 2012) adding more coefficients to allow for a greater control of the scene and obtain more accurate results.

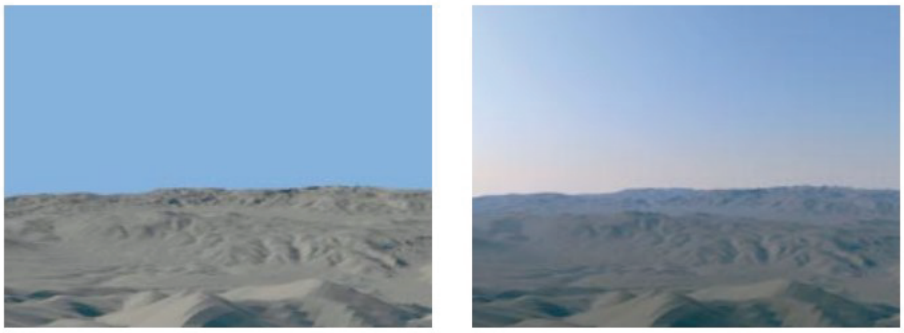


Figure 11: Left: rendered image of outdoor scene with constant sky. Right: The same image rendered with Preetham et. al. method. Source - (Preetham et al., 1999)

2.2.5 *GPU based models*

With the birth of the programmable GPU's, there was the possibility of a huge jump in performance, as atmospheric scattering is a perfect use case for parallelization, the strong suit of a GPU. This actually made it possible to run an algorithm with acceptable quality in real time, with interactive frame rates.

One of the first GPU models was presented by Dobashi et al. (Dobashi et al., 2002). The method is very similar to the one presented in (Nishita et al., 1993). The main difference is the introduction of the concept of "sampling planes", as opposed to sampling directions. This more easily allows for the algorithm to take advantage of the GPU to lower computation times. These planes are placed perpendicular to the view direction of camera, and divided into a grid. Each point on the grid is used as a sample point to apply the methods explored in (Nishita et al., 1993).

The next big jump in performance came at the hand of Sean O'Neal, in 2004 (O'Neil, 2004). He realized that the look-up table present in (Nishita et al., 1993) could be optimized to improve the per pixel performance, trading it off for some more pre-calculations. The look-up table proposed in (Nishita et al., 1993) stores values for the optical depth of the sun-light. O'Neal realized that an improved look-up table could also store values for the optical depth of the viewing ray. For that purpose, a 2D texture is used, indexed by *(h, θ)*. The h represents an altitude from the ground and θ represents a vertical angle. For each *(h, θ)* position in the texture, a ray is fired in the specific viewing direction, for which the optical-depth is stored. With the pre calculated values, the value at an arbitrary position can be calculated by interpolating the values in the look-up tables.

As we can see in figure 12, the result is very similar to Nishita's model, since it's based on that model, the main difference being O'Neal's model could run the simulation at interactive frame rates.

Not long after, O'Neal tried to optimize his algorithm even further by replacing these look up tables with analytical expressions (O'Neil, 2005). By plotting the results obtained for various h values with several

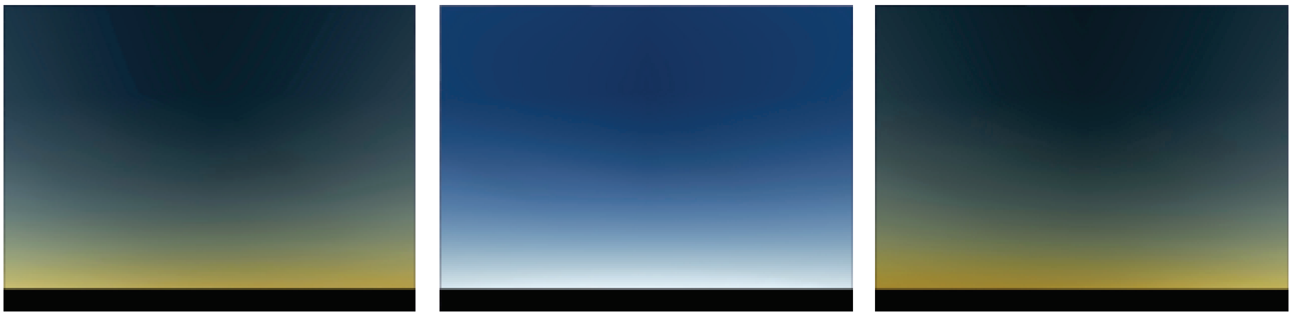


Figure 12: Images using O'Neal model of the atmosphere in clear sky. Source: Lopes and Ramires Fernandes (2014)

values of θ O'Neal realized that the results follow an exponential distribution for the values of *h*, where θ acts as a scaling value. With this knowledge, (O'Neil, 2005) proposes two analytical functions. One that describes the exponential distribution of *h*, and one that describes the scaling factor to which θ contributes.

In simple terms, the integral that is usually used to calculate the optical depth of a given path is replaced with $scale(\theta)D$, where D is the length of said path. For a view ray, θ is the angle of vision from the camera. For a light ray, θ is the angle between the camera and the sun.

Contrary to the model presented in (Nishita et al., 1993), this model can work with a single sample, taking into a account a slight reduction in quality, if the hardware requires so. That said, it can also use multiple samples along the view ray to improve the quality of the result.

With these optimizations in place, and modern hardware, it is rather trivial to get a reasonably accurate model of the sky at interactive frame rates.

2.2.6 *Pre-computed models*

After the work of O'Neal, there was a shift into improving the visual fidelity of real time algorithms. For this, there was a focus in pre-computing as much information as possible.

As discussed before, (Nishita et al., 1993) proposes to pre-compute the optical depth of the sunlight as to avoid the real time computation of its corresponding integral. (O'Neil, 2005) takes this same idea one step further by also tackling the optical depth of the view ray, either by pre-computing values or using analytical function to approximate integrals.

Schafhitzel et al. (Schafhitzel et al., 2007) on the other hand pre-computed every single integral. To achieve this, a 3D texture is used to store the values for only not every possible view and light direction, but also every possible height. These values are stored in a 3D table that can be later fetched.

Just one year later Bruneton et al. (Bruneton and Neyret, 2008) took this concept yet one step further by pre-computing values for multiple sun positions. This extends the lookup table to a 4D texture, the sun angle being the 4th dimension in addition to the already existing view and light direction and height. Much like the model in (Nishita et al., 1996), it calculates the value in multiple passes, summing the value of *n* scattering order for each sunlight and viewing direction.

These pre-computed models and previous GPU models use, for the actual computation of the colors, virtually the same algorithms developed in (Nishita et al., 1993).

2.2.7 *Other improvements*

Models over the last few years are almost invariably an extension of a previous existing model.

Jörg Haber et al. (Haber et al., 2005) takes the concepts of (Nishita et al., 1993) and (Nishita et al., 1996) and centers the concentric spheres on the camera, instead of the Earth. It also takes into account more directions during the pre-computation, possible due to the much more powerful hardware.

Oskar Elek and Petr Kmoch developed a spectral version of (Bruneton and Neyret, 2008). Theoretically speaking the number of existing wavelengths is infinite. We, of course, cannot work with infinity, but it follows that using more than 3 wavelengths results in a more accurate result. Instead of doing the pre-computations in 3 wavelengths, Elek et al. perform them on 15 wavelengths and convert the results to RGB at the very end of the pre-computation phase (leaving the sky rendering phase unchanged)(Elek and Kmoch, 2010).

This method is beneficial to quality as the spectra involved in the computation, while not extremely spiky, are not linear. Using only 3 channels for the computation essentially linearly interpolates the results for the wavelengths between the chosen ones, introducing errors in the computation. As we have to use discrete values, the higher the number of sampled wavelengths, the smaller the error will be.

In 2012, Lukas Hosek et al. (Hosek and Wilkie, 2012) developed a very similar model to (Preetham et al., 1999), the main differences being an improved analytical formula and a change to a fully spectral model. It also added parameters for turbidity and the ground albedo, which is essentially a measure of how reflective the ground is.

In 2016, Bodare et al. (Bodare and Sandberg, 2016) moved into the direction of developing an implementation that took better advantage of modern software and hardware features, such as compute shaders and the DirectX11 API, to provide a relatively easy to use implementation in AAA games. With this goal in mind, it is imperative to keep the implementation both computationally and memory efficient, both significant goals of the work.

In 2020, Shihan et al. (Shihan et al., 2020) introduced one more real-time application focused optimization, in the form of adaptive level of detail (LOD). LOD is a common optimization in real time applications, that reduces quality in places where it is not necessary. The most common example is using lower polygon count models when they are very far away, and the detail would be completely lost in the rendering process. This paper aims to vary the amount and location of samples in the ray marching algorithm based on the position of the camera to improve performance without sacrificing quality.

Galaj et al. (Gałaj et al., 2021) took notice that up until that point every paper used the trapezoidal rule for approximating the integral necessary for the computation of the optical depth. In an attempt to make the approximation of this integral more accurate, two alternatives are presented, one based on the splines and the second on the Taylor expansion. When compared to Bruneton's work, which they chose as a comparison point, they managed higher fidelity while also reducing the memory requirements.

2.3 CELESTIAL BODIES POSITIONING

Calculating the position of celestial bodies for an arbitrary time is not a trivial task. Thankfully, specialists in the subject have developed public and free to use code bases that do all the relevant calculations.

The sun, moon and other planets in the solar system change position in relation to the Earth very significantly, and as such it makes sense to, with a given date and time, calculate its position in the sky directly, in the horizontal system, which gives us the most useful information for a specific observer. On the other hand, stars are essentially static when taking into account the scale of the distances between us and them, so it makes sense so precompute their relative position to Earth.

For this purpose, the scientific community also provides public databases of enormous collections of stars, with the required information, equatorial system, to calculate the position of each one in the sky.

In this section we explore the most common used coordinate systems when describing the position of celestial bodies, and present some star catalogs.

In geometry, a coordinate system is a system that uses one or more numbers, or coordinates, to uniquely determine the position of the points or other geometric elements on a manifold such as Euclidean space. The most commonly used, and known to most people, is the *Cartesian coordinate system*.

2.3.1 *The Cartesian coordinate system*

A Cartesian coordinate system for a three-dimensional space consists of an ordered triplet of lines (the axes, usually labeled *X*, *Y* and *Z*) that go through a common point (the origin), and are pair-wise perpendicular,

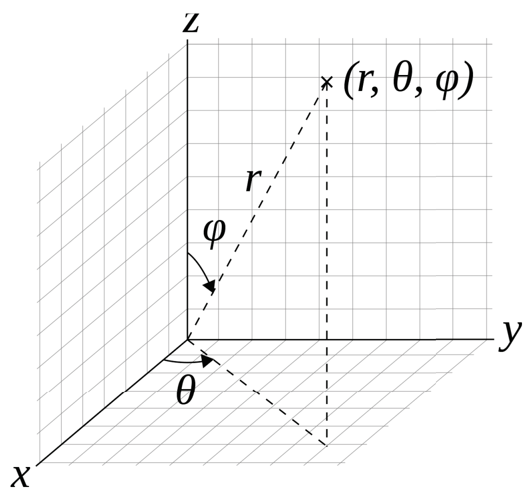


Figure 13: Example of a spherical coordinate system - Source: Andeggs (2022)

which means any which 2 axis selected are perpendicular to each other. Each coordinate of an arbitrary point *P* is defined as the distance from *P* to the hyperplane defined by the other two axes.

However, due to the way we naturally observe stars and other celestial objects, these are rarely positioned directly in the world using the Cartesian system.

2.3.2 *The spherical coordinate system*

The spherical coordinate system, of which we can see a representation in figure 13 is another fairly common way to specify the position of a point in a 3 dimensional space. A unique position is also defined by three values: the *radial distance* of that point from a fixed origin, its *azimuthal angle* measured from a fixed zenith direction, and the *polar angle* of its orthogonal projection on a reference plane that passes through the origin and is orthogonal to the zenith, measured from a fixed reference direction on that plane.

The use of symbols and the order of the coordinates differs among sources and disciplines. This dissertation will use the convention frequently encountered in mathematics: (r, θ, φ) gives the radial distance, azimuthal angle and polar angle.

It is worth noting that the entire 3D space can be mapped by varying the azimuthal angle between 0 and $2 * \pi$, and varying the polar angle between 0 and π. Any values beyond these, while valid, will also be representable by values inside the mentioned intervals.

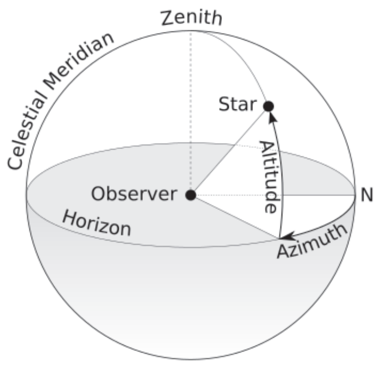


Figure 14: A representation of the horizontal coordinate system - Source: TWCarlson (2020)

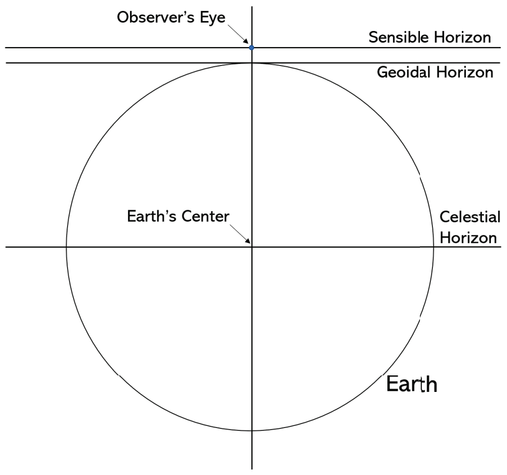


Figure 15: Representation of the multiple horizons used in astronomical coordinate systems

2.3.3 *Horizontal coordinate system*

The horizontal coordinate system borrows heavily from the spherical coordinate system to represent the position of celestial bodies in the sky, and is the system commonly used to describe the position of the celestial bodies. The horizontal coordinate system is a celestial coordinate system that uses the observer's local horizon as the fundamental plane to define two angles: altitude and azimuth. This celestial coordinate system divides the sky into two hemispheres: The upper hemisphere, where objects are above the horizon and are visible, and the lower hemisphere, where objects are below the horizon and cannot be seen, since the Earth obstructs views of them.

In figure 14, we can see an example of how to position a star in the sky. The altitude, the angle measured between the object and the horizon, positive towards the zenith. And the azimuth, the angle of the object around the horizon, measured from true north and increasing eastward, meaning that directly east the angle is 90º.

2.3.4 *Equatorial coordinate system*

Another commonly use coordinate system is commonly referred to as the equatorial system. The origin point is the center of the earth, making the celestial horizon the reference plane, as seen in figure 15.

Even though the horizontal system is more useful to the actual rendering of the scene, we also also have to take into consideration the equatorial system, as star databases use it to store its information.

It is trivial to see that, on a topocentric system, the position of a celestial body on the sky will be completely different depending on the position of the observer on the Earth. This would make it completely impossible to create a star database with its positions by using, for example, the horizontal system.

With this in mind, the equatorial system is functionally the same as the horizontal system, essentially containing only changes to the points of reference, to make it agnostic to location and time of recording of its position.

The observer can be imagined as if located at the center of the Earth. The angle that would correspond to the azimuth is called Right Ascension, and uses the celestial equator from the Sun at the March equinox measured Eastward as a point of reference. The angle that would correspond to the altitude is called ascension and is measured with the celestial horizon as the reference plane.

2.3.5 *Conversion between coordinate systems*

OpenGL uses the Cartesian coordinate system. As we obtain the position of the Moon and the Sun in the horizontal coordinate, we need to convert the values obtained. For this, we first need to establish some parallels between the spherical and horizontal coordinate systems.

The biggest difference between these two systems is the absence of a radius on the horizontal system, since it is only used to identify the direction of celestial bodies in the sky. However, to convert to the Cartesian system a radius is necessary. That said, we can use any arbitrary radius, as it is assumed the camera will always be near to the surface of the earth, meaning the absolute position of, for example, a star, is not relevant, needing only to be in the correct direction.

If we look at figures 13 and 14 side by side, it is easy to the similarities between them. Firstly, we can relate the origin point with the observer, and the reference plane with the horizon, as they represent the same concept in both systems. The azimuth is also essentially the same, where the horizontal system as the reference direction as true North. The polar angle and altitude also represent the same concept. However they use a different reference point. In the usual spherical system, the angles are measured from the Zenith, directly above the origin point. Altitude in the horizontal coordinate system is measured from the horizon. However, we can easily convert between them.

First, we need to realize that for a specific polar angle, it's negation will create a point mirrored by the zenith axis. The same mirror effect is true for the zenith 180º are added to its value.

With this in mind, the conversion from horizontal to spherical coordinates is done by subtracting 90º from the altitude to obtain the polar angle. However, this gives us the negation of the value we actually want. To fix this, we take the azimuth and add 180º to it, so we obtain the same point, by mirroring across the same axis twice.

From here, we can use the usual spherical to Cartesian conversion equations.

$$\begin{aligned} x &= r\ cos\varphi\ sin\theta \\ y &= r\ sin\varphi\ sin\theta \\ z &= r\ cos\theta \end{aligned} \tag{16}$$

On the other hand, conversion from the equatorial to the horizontal system is harder because it is needed to take into account the latitude, longitude, date and time, the calculations are considerably more complex. As explored in chapter 3, this conversion was performed by using a section of the SAMPA algorithm (NREL, 2012) that performs this exact conversion.

2.3.6 *Star magnitude*

In astronomy, magnitude is a unitless measure of the brightness of an object in a defined passband, often in the visible or infrared spectrum, but sometimes across all wavelengths.

The scale is logarithmic and defined such that each step of one magnitude changes the brightness by a factor of the fifth root of 100, or approximately 2.512(Brück, 1999). The brighter an object appears, the lower the value of its magnitude, with the brightest objects reaching negative values. For example, a magnitude 1 star is exactly 100 times brighter than a magnitude 6 star.

Astronomers use two different definitions of magnitude: apparent magnitude and absolute magnitude. The apparent magnitude (m) is the brightness of an object as it appears in the night sky from Earth. Apparent magnitude depends on an object's intrinsic luminosity, its distance, and the extinction reducing its brightness. The absolute magnitude (M) describes the intrinsic luminosity emitted by an object and is defined to be equal to the apparent magnitude that the object would have if it were placed at a certain distance from Earth, 10 parsecs (1 parsec equals $3.08567758 \times 10^{16}$ meters) for stars.(Brück, 1999)

For example, our own sun has an absolute magnitude of 4.83. However, due to its relative proximity to Earth of only 0,0000047787 parsecs, has an apparent magnitude of -27.

2.3.7 *Sky catalogues*

Epochs

To understand how a sky catalogue is built, is important to first understand what an epoch is. In astronomy, an epoch or reference epoch is a moment in time used as a reference point for some time-varying astronomical quantity. It is useful for the celestial coordinates or orbital elements of a celestial body, as they are subject to perturbations and vary with time. Essentially, it's a fixed point in time that can be associated with the measurements stored in a star catalogue, that indicates at what point in time these measurements were true. This then allows the calculations to be made to translate these measurements to any other point in time, as will be discussed later.

The most commonly used Epoch today is J2000, which corresponds to the Gregorian date January 1, 2000, at 12:00 TT (Terrestrial Time).

Evolution of catalogues

Throughout the years, there have been a number of published sky catalogs. The first so called "full-sky catalogue" was the "Histoire céleste française" (de Lalande, 1801). It contained 47,390 stars up to magnitude 9, observed by Laland and his staff in the Paris observatory. As expected, due to the limited technology of the time, the number of stars is limited compared to what we can obtain today. It is also mostly limited to the northern stars, which makes it unsuitable for our purpose.

However it is worth pointing out that stars of magnitude 9 are already a lot dimmer than what a human can see with the naked eye.

The next big jump came with the Henry Draper Catalogue (Pickering et al.). Not only did it include the entire sky, it also went up to magnitude 10 and was the first to compile spectroscopic classifications, meaning it used photography to find stars that weren't in the visible spectrum. It was a compilation of almost 5 decades of work by a multitude of people, which ended up collecting information about 255000 stars.

A very prevalent catalogue is the *Yale Bright Sky Catalogue* (YBSC) (NASA). It only contains 9110 objects, of which 9095 are stars, 1 are novae or supernovae and four are non-stellar objects. It purposefully only lists all the stars of magnitude 6.5 or brighter, which is roughly every star visible to the naked eye from Earth. However, unlike it's predecessors, each entry contains a lot more information. Namely, the catalogue detailed each star's coordinates, proper motions, photometric data, spectral types, amongst other information.

Notable progress was done in the Guide Star Catalog (GSC), also known as the Hubble Space Telescope Guide Catalog (HSTGC)(Lasker et al., 1990). It was compiled from information obtained from the Hubble

Space Telescope, as the name implies, and contains 945,592,683 stars out to magnitude 21. This is the first full sky star catalog created specifically for navigation in outer space.

The Hipparchus catalogue (European Space Agency, b) was compiled from the data gathered by the European Space Agency's astrometric satellite Hipparchus, which was operational from 1989 to 1993. The catalogue was published in June 1997 and contains 118,218 stars. An updated version with re-processed data was published in 2007. It is particularly notable for its parallax measurements, which are considerably more accurate than those produced by ground-based observations. It is a very interesting catalogue as it walks the line between prevalence and number of stars. It has considerably more stars than, for example, the Yale Bright Star Catalogue, while not having the huge amount of stars contained in the GSC or in the current biggest star catalogue, the Gaia Catalogue.

Gaia is a European space mission providing astrometry, photometry, and spectroscopy of stars in the Milky Way. Data for significant samples of stars outside the milky way and Solar system objects, like asteroids and planetary satellites, is also made available. The Gaia Archive (European Space Agency, a) contains deduced positions, parallaxes, proper motions, radial velocities, and brightnesses. At the time of writing, the Gaia catalogue contains information on 1.811.709.771 stars.

For this work, the HYG catalogue was chosen (Nash), created by David Nash. It isn't a new star catalogue by itself, but a compilation of the Yale Brightest Stars Catalogue, the Hipparchus Catalogue and the Gliese Catalogue, another catalogue similar to the YBSC, which focus on the most prevalent stars in the sky.

This database has 25 fields, most of which do not have an application for this work, like for example, the star radial velocity. The fields relevant to this work are:

- ra, dec: These two values, the right ascension and declination in the equatorial coordinate system, in hours and degrees respectively, describe the position of the star in the sky
- proper: A proper name of the star, when available. This could be used for vizualization.
- dist: The star distance in parsecs
- mag, absmag: Magnitude and absolute magnitude of the stars, used to render the stars with an appropriate luminosity.
- con: the standard constellation abbreviation that contains the star

Catalogue	Number of stars	Star Position	Apparent Mangnitude	Absolute Magnitude	Other fields
Histoire céleste française	47390	Horizontal coordinates at the time of observation at the location of the observatory, with instructions for conversion to equatorial coordinates	Up to 9	Not included	None
Henry Draper Catalogue	225300	Equatorial coordinates for epoch J1900	Up to 9	Not included	Spectral type
Yale Bright Star Catalogue	9110	Equatorial coordinates for epoch 1900 and epoch J2000	Up to 7	Included	Cross-referenced ID of multiple other catalogues, spectral type, proper motions, parallax, radial velocity,
Guide Star Catalogue	945,592,683	Equatorial coordinates for varied epochs	Up to 21	Included	Table has 28 fields, many specifically targeted towards space navigation, used to accurately position the Hubble space telescope
Hipparcus Catalogue	118218	Equatorial coordinates for varied epochs	Up to 11	Included	This is a complex catalogue with 78 columns. First catalogue to accurately measure positions, distances, motions, brightness and colors
Gaia	1.811.709.771	Equatorial coordinates for varied epochs	Up to 21	Included	This is the most complex and extensive catalogue in existence. Contains deduced positions, parallaxes, proper motions, radial velocities, and brightness measurements, as well as multiplicity, photometric variability, and astrophysical parameters

Table 1: Summary of the information present in each mentioned star database

Part II

CORE OF THE DISSERTATION

3

COMPOSING THE SKY

In this chapter, the steps required to rendering the sky, both in day and night time are presented.

We will first see how to calculate the influence of the sun in the colors of the sky. 3.1, followed by a description on how the locations of the moon 3.2, the planets 3.4 and the stars 3.3 are computed so that they can be added to the scene.

Finally we will discuss some implementation specific functionalities 3.6.

3.1 COMPUTING THE INFLUENCE OF THE SUN IN THE COLOR OF THE SKY

The color of the sky is calculated on a pixel by pixel basis, and each one has no bearing on neighboring pixels. The following description is therefore related to the calculation of a single pixel, and is repeated for all pixels on screen.

3.1.1 *Algorithm Overview*

We can calculate the color of a pixel with the information about where the observer lies, the direction of the rays coming from the sun, the view ray for the pixel under consideration, and the intersection point of this view vector with the top of the atmosphere.

Initially, an arbitrary number of points are selected along the view ray. The larger the number, the more accurate the simulation will be, but we quickly run into a diminishing returns barrier at around 10 samples, where any more precision is practically indistinguishable to the naked eye.

The algorithm starts a loop over all the points. For simplicity, let's focus on the first iteration of the loop, so we can later generalize the process to work across all points on the view ray ($VRay$).

Consider the first point on the $VRay$, and call it *V1*. The segment between the origin O and this point will be referred as *SegV1*.

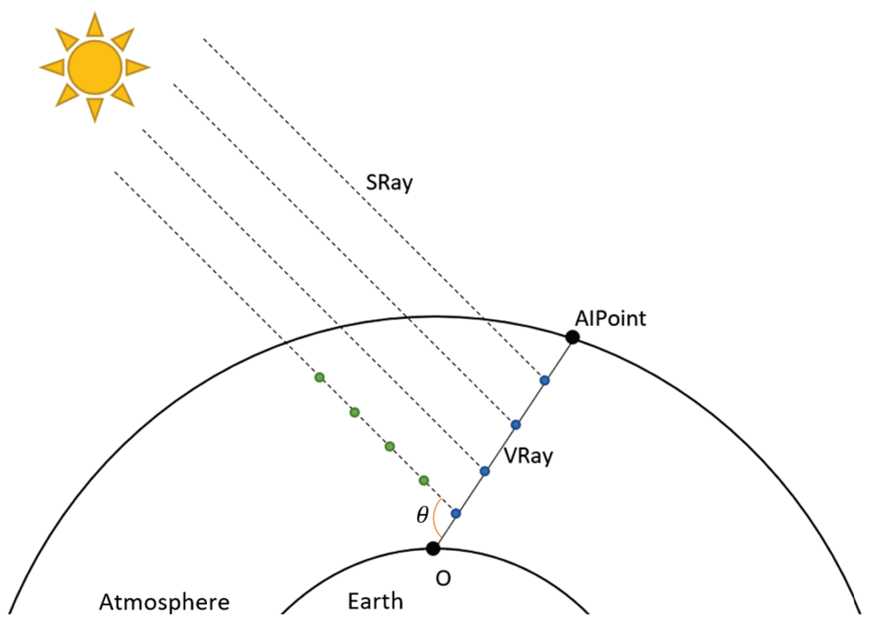


Figure 16: Diagram of the calculation of the color of the sky

Next we calculate the optical density of a ray pointing towards the sun, from *V1* to the top of the atmosphere.

This calculation is done for 3 different wavelengths, with the possibility for the user to change these values. By default, $700nm$, $530nm$ and $470nm$ are used, which are fairly common values for red, green and blue, respectively.

The higher the wavelength, the lower the attenuation will be. In turn, as explored in section 2.1.2, lower wavelengths will spread more broadly when they encounter a particle. This phenomenon is the main contributor to the colors of the sky, as shown in section 2.1.3.

With this, we know how much light reaches *V1* for each wavelength of light. Now we need to calculate how much light that reaches this point will also reach the camera, which is where the phase functions come in.

As discussed in section 2.1.2, the phase function describes changes in direction when light hits a particle. We can use them to calculate how much light will scatter towards an arbitrary direction, when coming from another arbitrary direction. There are multiple phase functions approximations that were proposed during the years. For this particular implementation, a Rayleigh approximation 6 is used for Rayleigh scattering:

$$phase_R(\theta) = \frac{1}{4\pi}\frac{3}{4}(1 + cos^2(\theta)) \tag{17}$$

and the Henyey-Greenstein phase function 6 is used for Mie scattering.

$$phase_M(\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2gcos(\theta))^{\frac{3}{2}}} \tag{18}$$

where θ is the angle between the two relevant directions, and *g* is a user defined value. To get a realistic result, g must be set to a value very close, but not equal to 1. By default, the application uses 0.999.

By multiplying the value of the attenuated light by the value given by the phase function, we get the amount of light that leaves *V1* in the direction of the camera.

We now need to compute the attenuation between *V1* and the camera. Instead of computing the optical density outright for that segment, we calculate the optical density of the entire $VRay$ using the sample points as the sample points for the approximation of the integral, and using the partial sum at each point as the optical density up to that point.

This process is repeated for every selected point along *VRay*. The sum of the result of each of these calculations gives us the color of the sky in the direction of *VRay*.

3.1.2 *Computing attenuation along an arbitrary path*

The attenuation along a particular path can be calculated by using the following equation:

$$att_T(s,\lambda) = exp(-\frac{4\pi K}{\lambda^4}\int^s \rho_T(p)dp) \tag{19}$$

where $\rho(h) = exp(-h/H_{den})$, h being the target height and H_{den} being a reference height, representing the maximum height where we can expect to find particles. ρ represents the density of particles at the particular height h. K is a constant, equal to the air density at sea level, which is $1.204kg/m^3$.

The integral present in this function cannot be exactly evaluated, therefore needing to be solved with numerical methods. To approximate this integral, the trapezoidal rule is used.

Let's look at path S, as we can see in figure 17. We know the length L of this path, and have a way to calculate the height in the atmosphere at each point H_x.

The positions of the sampling points can be calculated in two different ways: linearly or exponentially. The former gives us points along the path that are equally spaced, while the latter picks more points in the beginning of the path, increasing the spacing the more along the path we are. This can be advantageous in a particular case. If we assume the camera is located near the surface of the Earth, this means we will need to calculate the attenuation along a path that starts near the surface and go up to the top of the atmosphere. As the density of the atmosphere drops off with height, its influence in the final result goes

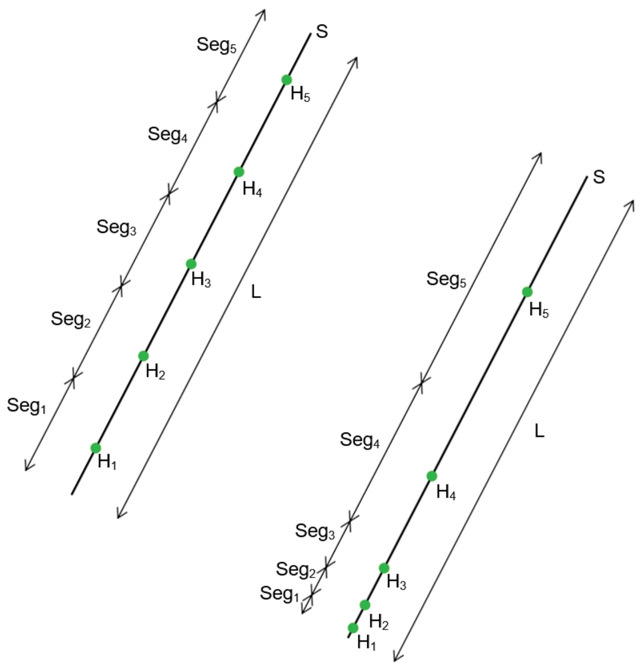


Figure 17: Sampling of points along a ray. Left: linear sampling. Right: Exponential sampling

down. Therefore, having more sampling points near the surface of the Earth may give an higher accuracy with no impact to performance.

The calculation isn't done directly for the points, but by getting the length of each segment.

For the linear calculation, the length of all the segments is the same, and equal to L divided by the number of points.

For the exponential, as discussed, the length of the segments increases as you go along. First, a *quotient* is calculated, which equals $L^{1/n}$, n being the number of sample points. Let's call *current* the sum of the lengths of the previous segments already calculated. So after the first iteration current equals the length of *Seg1*. After the second it equals the length of *Seg1* plus the length of *Seg2*, and so on. The segment length is then calculated as $current * quotient - current$. For the first segment, *current* is initialized to 1. Each point lies at the mid point of each segment.

Now that we have the points, the integral can be approximated by calculating the function inside the integral at each point, and multiplying the value by the length of the corresponding segments. The sum of all these values is the approximate result of the integral.

By plugging these values into equation 19, at the sample points to approximate the integral we obtain the value we were looking for. This is a value between 0 and 1 that represents the quantity of light that "gets trough" that segments unchanged. This should be multiplied by the light intensity of at the beginning of the path to know the intensity at the end of the path.

It is important to note that this function is dependent on the wavelength of the light, which will be important when computing the color.

3.1.3 *NREL's Solar and Moon Position Algorithm*

For the calculation of both the sun and moon positions, the NREL (2012) SAMPA algorithm is used.

The SAMPA (Solar And Moon Position Algorithm) algorithm was developed by the National Renewable Energy Laboratory (NREL). This algorithm calculates the solar and lunar zenith and azimuth angles in the period from the year -2000 to 6000, with uncertainties of +/- 0.0003 degrees for the Sun and +/- 0.003 degrees for the Moon, based on the date, time, and location on Earth.

The algorithm takes as inputs the date, hour and the latitude and longitude of the observer and returns the values of the position of the Sun and the Moon, in relation to the observer, in the horizontal coordinate system, described in section 2.3.3. This is the first step of the pipeline, and the output values are stored in a data texture to be used in the subsequent steps.

Due to the distance from the Sun to the Earth, the light rays that reach the Earth are essentially parallel. Sun rays that reach the Earth form a maximum angle of about 0.53 degrees. This value is small enough that taking this into consideration would make no difference to the final result. Therefore, it will be ignored to simplify the process.

After adding the SAMPA C source code to the project, use of this API is very simple. We only need to create a variable of the provided struct and fill the relevant fields, and call a single function.

The $''sampa''$ struct is made up of two other structs and 16 values related to solar irradiances, which we are not interested and will therefore ignore. The two structs in question are $''spa''$, that stores values related to the sun, and $''mpa''$ which stores values related to the moon. However, all the relevant inputs are done in the $''spa''$ struct, and the $''mpa''$ struct will only contain outputs. These inputs include values that are relevant to us, as well as values that are mandatory to fill as there are no default meaningful values, and the code will fail otherwise. Here, the relevant values to the simulation are shown. The full inputs necessary can be found in appendix A.1

```
    sampa_data sampa;  //declare the SAMPA structure
    int result;

    //enter required inputs values into SAMPA structure
    sampa.spa.year       = 2023;        //Range: -2000 to 6000, int
    sampa.spa.month      = 5;           //Range: 1 to  12, int
    sampa.spa.day        = 12;          //Range: 1 to  31, int
    sampa.spa.hour       = 1;           //Range: 0 to  24, int
    sampa.spa.minute     = 33;          //Range: 0 to  59, int
    sampa.spa.second     = 0;           //Range: 0 to <60, double
    sampa.spa.timezone   = 0;           //Range: -18 to 18 hours, double
    sampa.spa.longitude  = 143.36167;   //Range: -180 to 180 degrees, double
    sampa.spa.latitude   = 24.61167;    //Range: -90 to 90 degrees, double

    //call the SAMPA calculate function and pass the SAMPA structure
    result = sampa_calculate(&sampa);
```

The variable "result" contains a report code indicating if the operation was successful, and the output is written to fields in the structs. There are 4 fields of interest for the purposes of the simulation:

- $sampa.spa.azimuth\_astro$ - sun's azimuth
- $sampa.spa.e$ - sun's altitude
- $sampa.mpa.azimuth\_astro$ - moon's azimuth
- $sampa.mpa.e$ - moon's altitude

3.1.4 *Direction of the view ray*

To calculate the color for a given pixel, we need to know the direction of a ray that goes through that particular pixel after leaving the camera. This ray is marked as *VRay* in figure 16, and will be denoted as *dir* in this section.

This ray goes from the observer, point O, to the top of the atmosphere. We know for a fact the ray will always intersect the atmosphere border sphere at some point, as it is assumed that the observer is always within the atmosphere, which simplifies part of the process.

To calculate the direction of this ray we can place an imaginary grid in front of the camera, as seen in figure 18. Each element of the grid corresponds to one of the pixels of the screen, with its correspondent view ray.

Let's imagine now a Cartesian coordinate system centered in the camera, where the negative z-axis is the camera view direction, the up-vector is the y-axis and the right-vector is the x-axis. The imaginary grid is a parallel plane to the XY plane of this coordinate system.

The vector from the camera to the center of the screen grid is a unit vector with the same direction as the viewing direction. This provides the ray direction for the pixel in the center of the screen. To get the direction of other points on the grid, we need to vary the pos_x and pos_y values, as in equation 23. The range of these values is dependent on the FOV and $ratio$ between the horizontal and vertical resolution.

Each pixel on the screen can be identified by its UV coordinate. As such, we can attribute these same coordinates to the imaginary grid, as if it was the user's screen. Let (u, v) be the UV coordinates of a point on the grid, *camUP* the up-vector of the camera, *camRIGHT* the right-vector of the camera and *ratio* the ratio between the horizontal and vertical resolution of the screen. The vector to a specific pixel is calculated as:

$$verticalAperture = tan(FOV * 0.5) \tag{20}$$

$$(t, c) = (u, v) * 2 - 1 \tag{21}$$

$$\begin{aligned} pos\_x &= t * ratio * verticalAperture \\ pos\_y &= c * verticalAperture \end{aligned} \tag{22}$$

$$dir = \textit{camUP} * pos\_y + \textit{camRight} * pos\_x + camView \tag{23}$$

Note that this method only works for an FOV lower than 180 degrees, as when we approach 180 degrees, which corresponds to PI, equation 20 tends to infinity. This is perfectly acceptable as a fairly natural FOV usually resides somewhere between 60 and 90 degrees.

3.1.5 *Computing the length of the view ray*

The distance from point O to $AIPoint$, as seen in figure 16 is needed as this distance is part of the optical density calculation, hereafter denoted as d, which is also the length of *VRay*.

Let's consider figure 19, where O is the origin point, and C is the center of the sphere. We have the direction of the ray, which will be referenced as *dir*, as a normalized vector, and the goal is to calculated the distance between O and $AIPoint$, the point at which the ray intersects the top of the atmosphere.

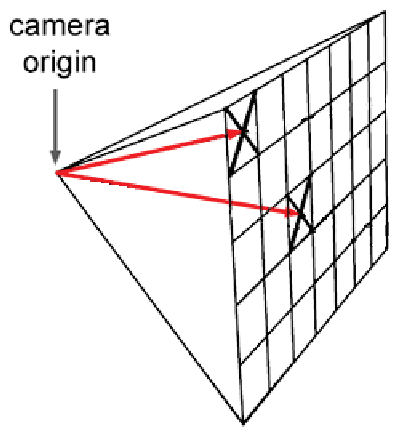


Figure 18: A representation of the selection of camera rays. Source: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview

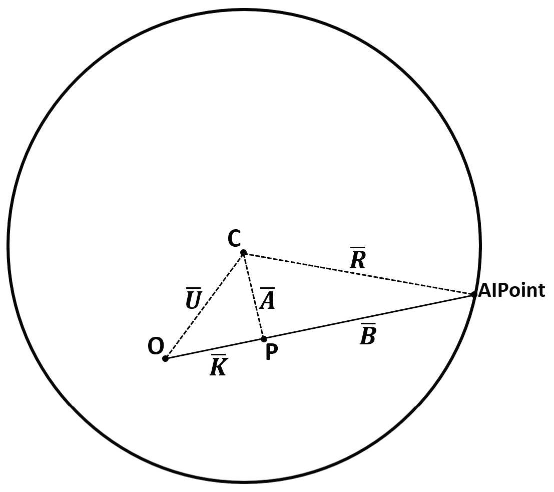


Figure 19: Representation of how to calculate the distance from a point inside a sphere to it's surface, in a specified direction, where the projection lies on the view ray

For these calculations, we assume the center of the Earth is also the center of the current coordinate system. With that in mind, it's easy to see that the vector $\vec{U}$ is $-O$.

We can now calculate the length of the vector $\vec{K}$, which we will call *k*, as the dot product of *dir* and $\vec{U}$. Keep in mind this is a signed distance. So if the projection P doesn't fall on the view ray, this value will be negative.

As $\vec{K}$ shares its direction with *dir*, we can calculate P as

$$P = O + k * dir \tag{24}$$

It is also trivial to calculate the length of $\vec{A}$. As one of its end points is C, the value of the vector is equal to point P. Since $\vec{R}$ is a radius of the sphere, its length is also know.

As we can see in figure 19, $\vec{A}$, $\vec{B}$ and $\vec{R}$ form a right triangle, with $\vec{R}$ being the hypotenuse. This means we know that:

$$r^2 = a^2 + b^2 \tag{25}$$

where a, b and r are the lengths of the vectors $\vec{A}$, $\vec{B}$ and $\vec{R}$, respectively. From the previous equation follows:

$$b = \sqrt{r^2 - a^2} \tag{26}$$

Finally, the value of d is given by

$$d = b + k \tag{27}$$

As we can see in figure 20, this exact algorithm will work even if the projection of the center of the sphere doesn't lie on the ray, as *k* will be negative and will subtract from *b*.

3.1.6 *Algorithm summary*

Combined, these will make up the majority of code necessary to calculate the influence of the sun in the color of the sky.

The algorithm can be resumed as follows:

- For each pixel:
  - Calculate the direction of the view ray 3.1.4

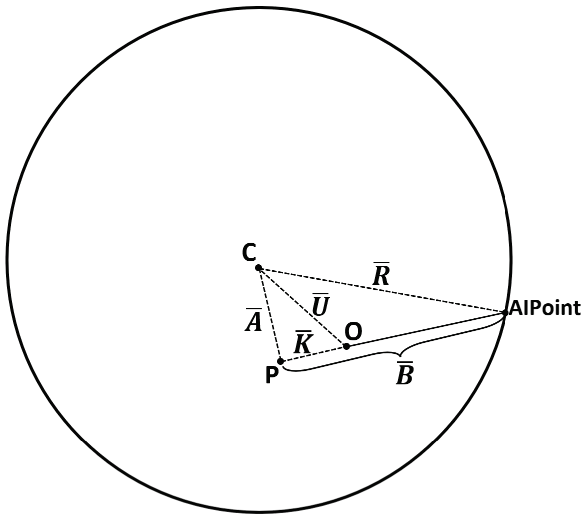


Figure 20: Representation of how to calculate the distance from a point inside a sphere to it's surface, in a specified direction, where the projection doesn't lie on the view ray

- Calculate the distance to the top of the atmosphere 3.1.5
- Calculate the sampling points on the view ray (points V) 2.1.1
- For each point V:
  * Calculate the distance to the top of the atmosphere from the point V in the direction of the sun 3.1.5
  * Calculate the optical density of the previous ray 3.1.2
  * Calculate the phase function value based on angle between the view ray and the sun ray 3.1.1
  * Calculate the light that reaches the camera based on the optical densities and phase function 3.1.1
  * Sum the result of all points

3.2 DRAWING THE MOON

3.2.1 *Diffuse representation of the moon*

The moon is drawn as a sphere with a texture (figure 21). Due to the distance, any lack of detail in the terrain is not noticeable.

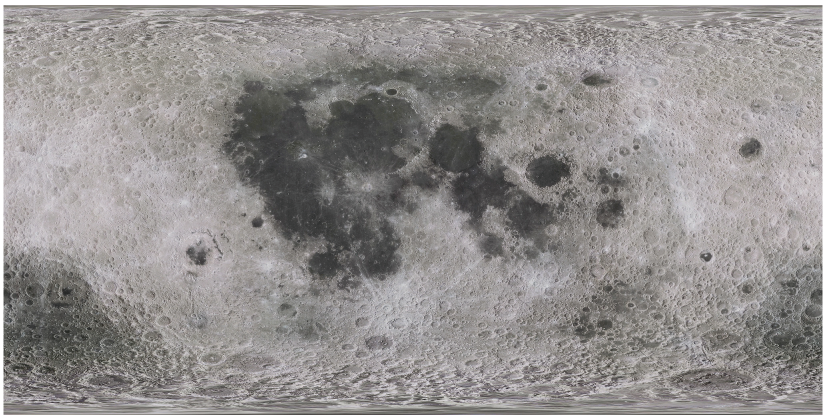


Figure 21: The texture used for the moon

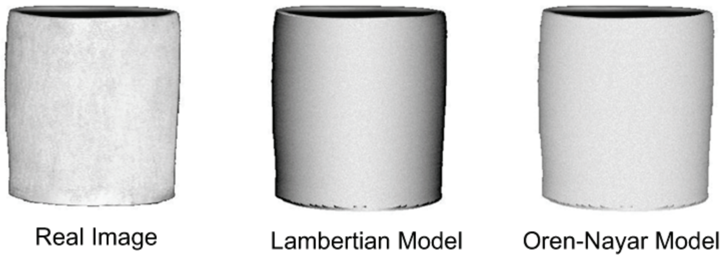


Figure 22: Comparison between a real photograph, a render using the Lambertian model, and using the Oren-Nayar model. Source: `https://www.cs.columbia.edu/CAVE//projects/oren/`

The moon is first drawn by itself, without taking into account atmospheric scattering. It is drawn to a texture to be combined with the other elements later in the process, as can be seen in figure 23.

The shader uses the Oren–Nayar reflectance model, shown to accurately predict the appearance of a wide range of natural surfaces, such as concrete, plaster and sand. It is also a rather good match for the appearance of the moon.

As we can see in figure 22, the typical Lambertian model darkens the image too quickly as the object curves away from the camera. The Oren-Nayar model mitigates this effect, giving us a considerably more realistic result. (Columbia University: Department of Computer Science)

To get the direction of the light that illuminates the moon, the position of the moon is subtracted from the position of the sun. This gives us a vector that points from the sun to the moon, which, of course, coincides with the direction of the light that illuminates the moon.

As for the rotation of the Moon, it is synchronized with its orbit around the Earth, meaning, if the Moon's orbit was a circle, it would always show the same face to the Earth. However, due to its elliptical nature, the apparent rotation slightly fluctuates. When the moon is closest to Earth, its rotation is slower than its journey through space, and allows observers to see an additional 8 degrees on the eastern side. When

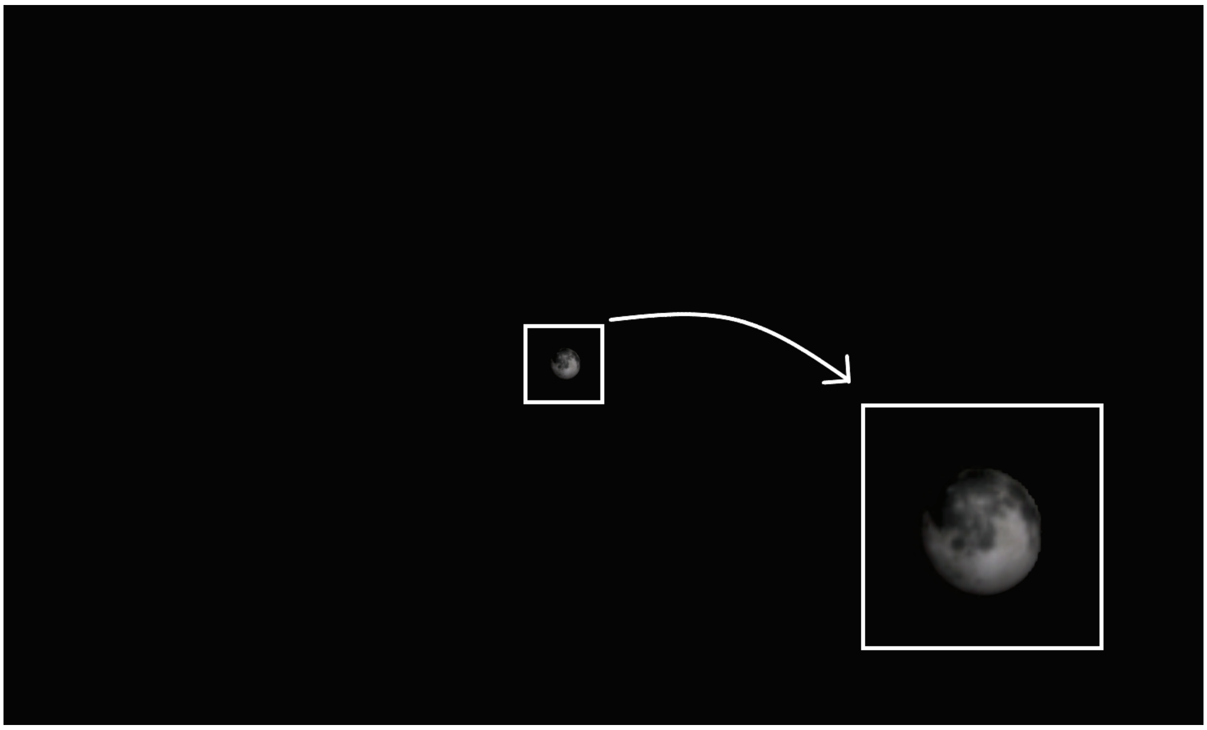


Figure 23: Example of the temporary texture to which the moon is drawn to, before attenuation is computed

the moon is farthest, the rotation is faster, and an additional 8 degrees are visible on the western side, as depicted in figure 24

3.2.2 *Light attenuation for the moon*

The light reflected from the moon is, of course, also affected by the atmosphere. Some of the light that would have reached the observer otherwise is scattered away, leaving only the light that managed to follow a relatively straight path, resulting in the attenuation of the light and a slight color shift, due to different wave lengths being attenuated to different degrees. Applying this effect is, however, quite simple, given the work already discussed in section 3.1.2.

This operation is applied to every pixel of the texture resulting from the rendering of the moon. If the texture is a perfect black, meaning no attenuation needs to applied that pixel can just be ignored, saving resources. On the other hand, if the pixel isn't black, the following steps can be applied:

- Calculating the direction of that particular pixel as discussed in section 3.1.4
- Calculating the length of the ray that goes from the observer to the top of the atmosphere, as discussed in section 3.1.5

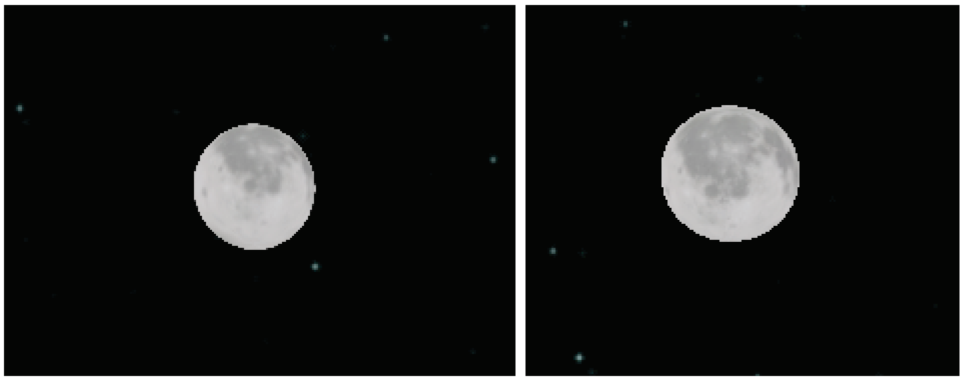


Figure 24: Rotation of the moon 15 days apart. Sun direction was ignored for clarity

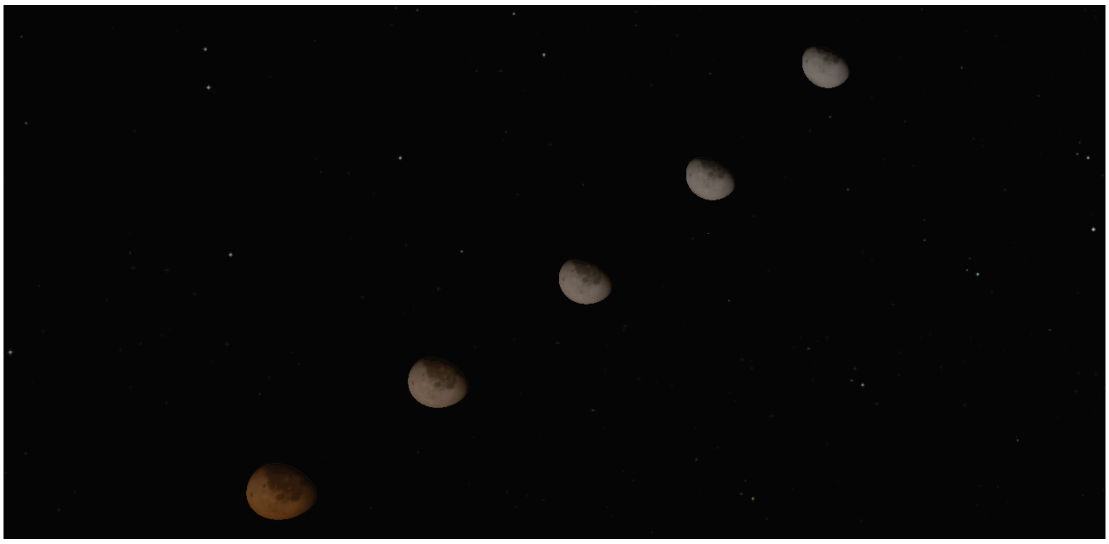


Figure 25: The effect of the attenuation on the moon as it distances itself from the horizon

- Calculating the attenuation of light along that ray, as discussed in section 3.1.2

The resulting color shift of this effect can be seen in figure 25.

3.3 DRAWING THE STARS

This section details the steps to render the stars. It starts by discussing the geometric approach, followed by the positioning of the star on the screen. To conclude, some considerations are presented regarding the brightness of the star representation.

Figure 26: Texture used for the stars

3.3.1 *Geometry of the star*

An obvious course of action would be to place a sphere at the location of each star. However, considering how small this sphere would appear in the final image, this would be a waste of resources. Considering there are almost 120000 stars in the chosen database, a large amount of vertices starts to add up quickly, and some level of detail approach would have to be used.

So instead, a simple quad with a texture of star, shown in figure 26, is used in its place.

The quad needs to act as a billboard, i.e. to be facing the camera regardless of its position. To perform a rotation, we need the correspondent axis and rotation angle. Essentially we're looking to align the normal of the quad with a vector that points from the location of the star to the camera.

The axis of rotation can be computed as:

$$axis = normalize(quad\_normal \times desired\_normal) \tag{28}$$

and the angle of rotation can be computed as:

$$angle = \arccos(normalize(quad\_normal) \cdot normalize(desired\_normal)) \tag{29}$$

where $quad\_normal$ is the normal of the quad, $desired\_normal$ is the desired normal after rotation, which is a vector from position of the quad to the camera, and $normalize$ is a function that normalizes a vector.

The end result is a texture to which all the star billboards are drawn.

3.3.2 *Conversion between equatorial and horizontal coordinate systems*

As described in subsection 2.3.7, two of the fields of the star catalogue are the star's right ascension and declination. These two values describe the position of a celestial body in the sky.

Let's take, for example, the star Polaris. We can see in the selected star catalogue (Nash) that its right ascension is 2.529750 hours (37.94 degrees) and the altitude is 89.2 degrees. As a reminder these values are for the epoch J2000 , which the SAMPA algorithm defaults to. Let's assume we want to calculate the location of the star in the sky for someone located at University of Minho, specifically the Gualtar campus, with longitude 41.561 degrees and latitude -8.397 degrees, on the 15th of January, 2023 at 16:00.

This conversion starts by determining the observer's local sidereal time (LST). The LST represents the hour angle of the vernal equinox measured from the observer's location. It indicates the position of the celestial meridian (a line connecting the north and south points in the sky) at a given time.

Next, we calculate the hour angle (HA), the angular distance measured along the celestial equator from the observer's meridian to the object of interest. It indicates how far east or west the object is from the observer's location.

With the two previous values we can calculate the altitude, which represents the angular distance above the observer's horizon. Finally, we can calculate the azimuth and declination, the two values we want that compose the horizontal coordinates.

In order to achieve this we are once again using the SAMPA library. From a high level standpoint, it is possible to split the SAMPA code in two parts: the first one that computes the position the sun/moon as a right ascension and declination, in the equatorial system for the epoch J2000, and a second one that converts these values to the corresponding horizontal system coordinates, as described in the previous subsection.

When it comes to the inputs, we need to first understand that, in simple terms, the contents of the "spa" struct described in section 3.1.3 can be divided into three sections: the input section, that the user is expected to fill; the intermediate-output section, which holds values mid calculation, and the output section, with the values the creators of this tool expected a user to need. For the current purpose the "spa" structure will be used as the complete SAMPA struct is bigger and unnecessary.

Two of these intermediate-output values are the right ascension and declination, the available values that store the position of a star. Simply put, one of the intermediate steps of the SAMPA algorithm is the calculation of the right ascension in declination. We can use this to our advantage by filling out these intermediate values ourselves, and only running the code that would normally run after they are calculated such that the translation will be performed.

With this in mind, these become the fields that are relevant to us, as shown in listing 3.1

```
    spa_data spa;   //declare the SAMPA structure
    int result;

    //enter required inputs values into SAMPA structure

    spa.year          = 2023;       //Range: -2000 to 6000, int
    spa.month         = 1;          //Range: 1 to  12, int
    spa.day           = 15;         //Range: 1 to  31, int
    spa.hour          = 16;         //Range: 0 to  24, int
    spa.minute        = 0;          //Range: 0 to  59, int
    spa.second        = 0;          //Range: 0 to <60, double
    spa.timezone      = 0;          //Range: -18 to 18 hours, double
    spa.longitude     = 41.561;     //Range: -180 to 180 degrees, double
    spa.latitude      = -8.397;     //Range: -90 to 90 degrees, double

    spa.spa_alpha     = rightAscension
    spa.spa_delta     = declination

    //call the SAMPA calculate function and pass the SAMPA structure
    result = spa_calculate(&sampa);  //modified version
```

Listing 3.1: Sampa structure for the required inputs

The outputs we want, the azimuth and elevation, are contained in the $spa.azimuth\_astro$ and $spa.e$ fields, respectively.

For the particular example started in the previous section, the code returns an azimuth of -0.0135312 and a altitude of -0.837514.

In turn, these can be translated to the Cartesian Coordinate System, using the equation for spherical coordinates in section 2.3.5. When using spherical coordinates, r is the distance from the point of reference to the calculate position. For this purpose, this distance is basically arbitrary. Since the camera doesn't freely move around the world, and the difference in the size of stars is basically imperceptible, placing the quad closer or farther away nets no tangible difference.

Once again, for this particular example, we get values of 4.02125, 267.725, -297.165 for x, y and z respectively, for an r of 400. With an r of 400, the stars sit close to the edge of the render distance, so that they will always be behind any other elements we draw on the scene.

3.3.3 *Star Brightness*

One field present in the selected star catalogue (Nash) is the apparent magnitude of the star. This describes how bright the star appears to us on Earth, where the lower the value, the brightest it is. The naked human eye can see stars about as dim as magnitude 7.

The star catalogue contains stars as bright as -26.7, which is our own sun, and as dim as 19.7. If we exclude the Sun, the brightest star on the catalogue, as well as the brightest star in the night sky, is Sirius, with a brightness of -1.4. The magnitude of the dimmest star visible to the naked eye is around 7.

Based on the magnitude of a star, we can converted this value to its equivalent in lux with equation 30

$$E_{lux} = 10^{(-14.18 - magnitude)/2.5} \tag{30}$$

As with the moon, described in section 3.2.2, a post processing step is applied to the resulting texture, applying the effects of attenuation to their light.

3.4 DRAWING THE PLANETS

Similarly to the moon and the stars, the planets are first drawn to a separate texture, to be combined in the final result at a later point in the pipeline.

3.4.1 *Position of the planets*

The position of a celestial body that orbits another, like the Solar System's planets orbit the Sun, can be calculated from their orbital elements, a set of values unique to that body that describes their trajectory. The algorithm used for this thesis was developed by Paul Schlyter (Schlyter). In his website, he presents a tutorial on how to calculate these values, as well as some tips on how to implement it in code. This algorithm does a fair amount of simplifications, resulting in a rather small amount of code necessary, while keeping a very respectable accuracy, of about 1 arc minute, i.e., about 1/60th of a degree. This should result in errors that are below or just around the size of a pixel, and will have no noticeable impact on the final result.

Paul Schlyter's website provides the aforementioned orbital elements for each planet, and the entire algorithm is reduced to two functions: one that calculates the position of the sun, and another one that uses the output of the previous function to calculate the position of each planet, by passing it its specific orbital elements and the desired date and time. This code can be seen in appendix A.2.

In the code, these orbital elements were saved with the use of a struct, with 3 extra fields representing the right ascension and declination that we want to compute, and the distance to the planet.

First, we have to calculate the date as a number of fractional days since midnight, January 1st of 2000, which will be called d. This can be obtained by passing the year, month, day and decimal hour to the $compute\_day$ function.

With this value, and the orbital elements of the sun, we can calculate its position with the $computePositionSun$ function, which takes in those two arguments. This function will return to us 2 values, which we will call xs and ys and represent the rectangular geocentric coordinates of the sun.

Finally, we call the $computePositionPlanet$ function, which takes xs, xy, d and the orbital elements of the desired planet as arguments. This function populates the planet's right ascension, declination and distance values inside the struct. These values are in an equatorial coordinate systems.

With the right ascension and declination of the planet computed, as well as the distance to said planet, first we need to convert to the horizontal coordinate system, and finally converted again to Cartesian coordinates,similarly to what was as shown in section 3.3.2.

Let's take, for example, the 11th of April, of 2023. According to https://starwalk.space/pt/news/what-is-planet-parade, Mercury, Uranus, Venus and Mars should align to a certain degree on the sky. If we plug that date into the implemented algorithm, we can see these values for each planet:

Planet	RA	Dec
Mercury	217.201	16.720
Venus	237.521	20.246
Mars	279.049	23.012
Jupiter	200.101	7.268
Saturn	155.845	-11.290
Uranus	224.684	15.963
Netpune	176.602	-2.750

Table 2: Right ascension and Declinations of the planets on the 11th of April, 2023

As we can see, especially for Mercury, Venus and Uranus, their angles are quite close together, making them appear quite close in the sky.

3.4.2 *Rendering the planets*

Each planet is rendered as a sphere, with a texture obtained from Solar System Scope (INOVE).

The size of the sphere is calculated from the planet's size and its distance to the Earth. An object that is twice as close and is half the size essentially has the same apparent size. With this in mind, we can divide

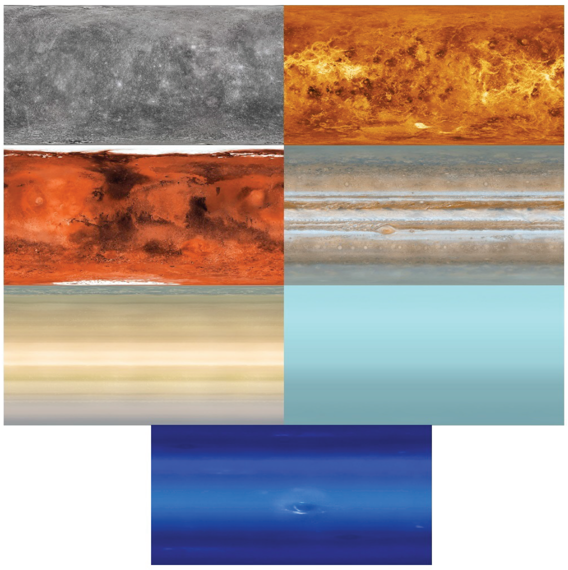


Figure 27: The textures used for the planets. Left to right, top to bottom: Mercury, Venus, Mars, Jupiter, Saturn, Uranus, Neptune

both the size of the planet and the distance to it by 5 million, so the values land below 1000. This value was chosen as the closest a planet ever gets to the Earth is around 61 million kilometers and the farthest is 4700 million kilometers. This is done because if the value of the far plane, that is, the maximum distance OpenGL will draw geometry, is too big, it will create a lot of precision issues.

As with the moon, described in section 3.2.2, a post processing step is applied to the resulting texture, applying the effects of attenuation to their reflected light, as seen in image 25.

3.5 COMBINING THE TEXTURES WITH THE DAY SKY

As mentioned in previous sections, the moon, stars and planets are drawn to separate textures, to be combined as the finished result at a later stage.

For this, it's important to highlight that the result of the calculations described in section 3.1, further mentioned in this section as sky calculations, aren't bound in the typical RGB space, the value obtained for each pixel instead representing an arbitrary intensity.

With this in mind, we can see that if we multiply all the values by any arbitrary amount bigger than 1, we don't lose information, as what really matters is the relations between each pixel, rather than the absolute value.

In contrast to that, the moon, stars and planets textures, when drawn, are already in the typical RBG range, as they are stored as image textures.

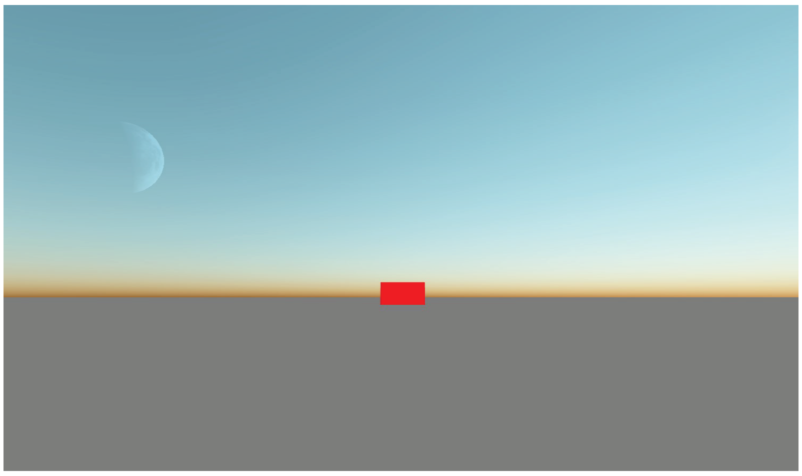


Figure 28: Barely visible moon during the day

In real life it is obvious to see the light that comes from the sun is a lot brighter than any other source. To mirror this effect in the rendering of the sky, the result of the sky calculations are multiplied by a fairly large value. The value chosen for this multiplication was 30, found by experimentation. To the result of the contribution of the sky, the values of the 3 intermediate textures, bounded from 0 to 255, are directly added, pixel by pixel. This means that the contributions of the lights from sources other than the sun have a lot less weight on the final result than said sun. In practice, this means that the moon is barely visible during the day at specific times, as shown in figure 28, and the stars and planets aren't visible at all until dusk, see figure 29.

To convert these values to a usable range, exposure tone mapping (Vries, 2014) is applied to bring most of the values to between 0 and 1. The resulting value has to be clipped for proper display.

$$result = pow(vec3(1.0) - exp(-input * exposure), vec3(1.0/2.2)); \tag{31}$$

In equation 31, *input* is the result of the sky calculations and exposure is a configurable value that will control the overall brightness of the result. This equation also performs gamma correction.

3.6 APPLICATION FUNCTIONALITY

The source code for both developed application can be found at GitHub.

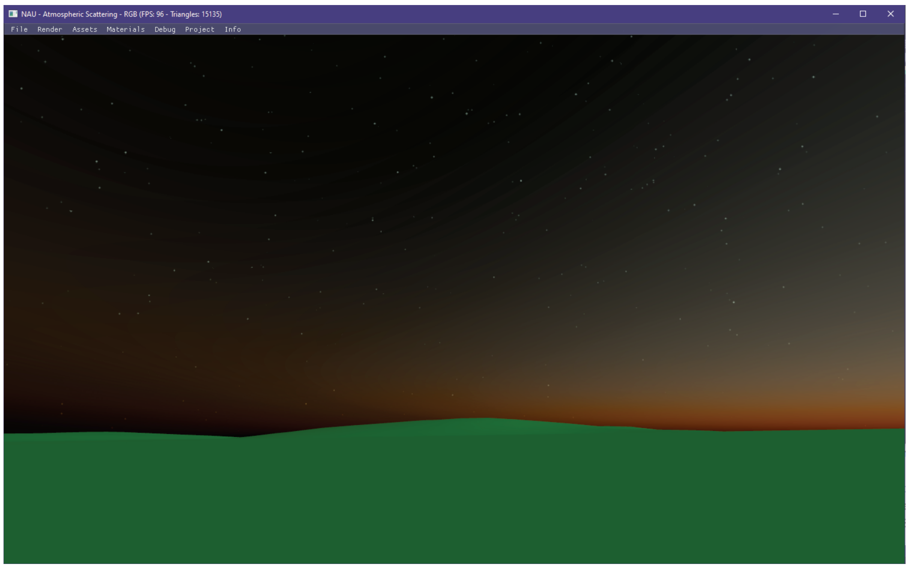

NAU - Atmospheric Scattering - RGB (FPS: 96 - Triangles: 15135)
File
Render
Assets
Materials
Debug
Project
Info

Figure 29: Rendering the sky during dusk

3.6.1 *Desktop*

The Nau3D engine was chosen `https://github.com/Nau3D/nau` for development of a desktop version. Nau3D is an OpenGL based engine developed at *Universidade do Minho*, by teachers and students alike. It is a rather simple engine when it comes to features, as it doesn't provide any pre-built effects or graphical pipelines. However, it massively simplifies the implementation of OpenGL shaders, as it quietly handles all the necessary C++ code in the background, allowing a complete focus on the shaders themselves. It reduces the necessary code to a comparatively simple XML file that tells the engine which shaders to use in which order, alongside some data structures. It also automatically calculates useful values like the projection and view matrices. From there, it handles all the compiling and linking, copying of data to the GPU and the render loop without any intervention from the user.

3.6.2 *Android smartphone*

An Android app was also developed. *Android Studio*, the "default" IDE for Android apps, developed by Google itself, supports OpenGL development. Specifically, it supports the OpenGL ES library. OpenGL ES is a cross-platform API for rendering 2D and 3D graphics mainly targeted towards embedded and mobile systems. It consists of a well-defined subset of desktop OpenGL more suited for low-power devices, such as a phone or tablet.

As stated, the OpenGL ES API is a subset of the desktop OpenGL API, meaning that a direct translation from one to the other isn't always possible. That said, in recent years OpenGL ES dropped a lot of its limitations, spurred on by the explosive development of hardware that allows even devices like a phone to be considerably powerful. As such, it contains practically all the core features of desktop OpenGL, and after having the code fully developed for the desktop version of the app, translating it to work with OpenGL ES was a rather trivial ordeal, when it comes to the shaders themselves.

Nevertheless, the Android version needed some extra work due to the lack of a dedicated graphical engine such as Nau3D, meaning the supporting code had to be written. However, this was a considerably smaller effort, for two reasons. Firstly, as mentioned, the shaders were already written and required little to no adaptation. Secondly, Android studio supports Java development, and many of the OpenGL calls are abstracted for the end user, removing mainly memory management concerns, one of the demanding aspects of a program. This means the code was much simpler and shorter than it's equivalent C++ desktop equivalent.

A small caveat is that technically speaking, implementation of OpenGL ES support is down to maker of the specific GPU contained on a particular phone, not to the Android version. In practice, it's unlikely to find a phone that doesn't support OpenGL ES. As of June 2022, Google estimated that 93.24% of Android phones supported at least OpenGL ES 3.0 (Google), the version used for this paper, while every other surveyed smartphone supported version 2.0.

Version 3.0 was chosen for this implementation as it featured a big jump in features and a change in semantics that made it much closer to the desktop version, such that it is worth using it and taking a small hit to the compatibility of the app.

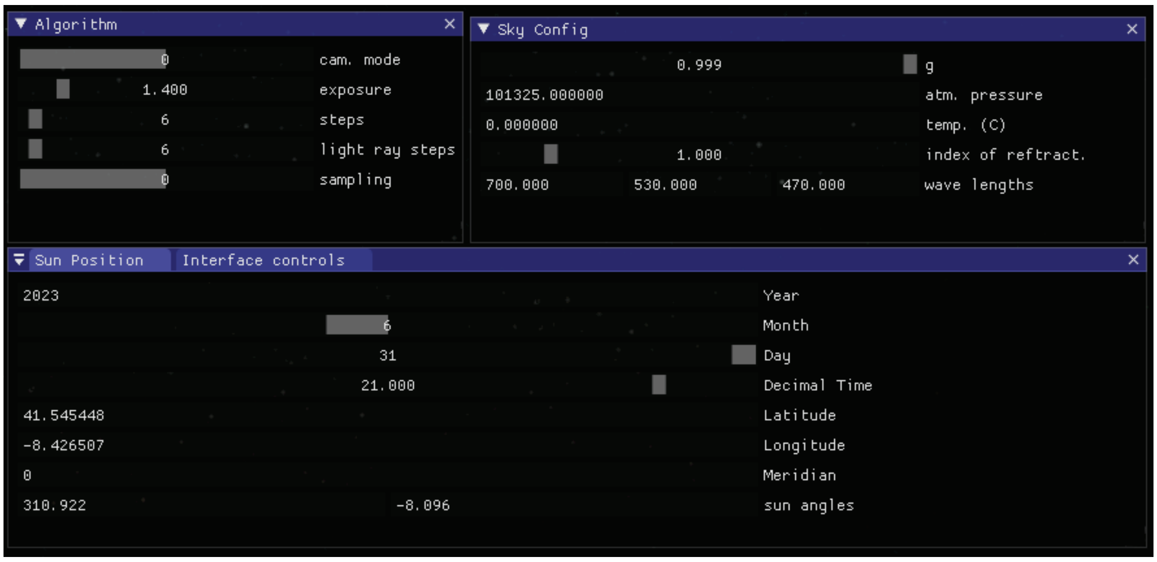
3.6.3 *User interface*

The application allows the user to, in real time, input values to set the date and time as well as select aspects of the simulation.

The interface is a collection of sliders, for values that have a very defined range like time of day, and number/text inputs, for values that have a very big range, an undefined range or very small steps for which the granularity of a slider is not enough.

The desktop application uses the interface provided by the engine, as seen in figure 30.

In a similar vein, the Android interface is a collection of the default input methods provided by the Android toolkit, which occupies a portion of the right of the screen, as seen in figure 31

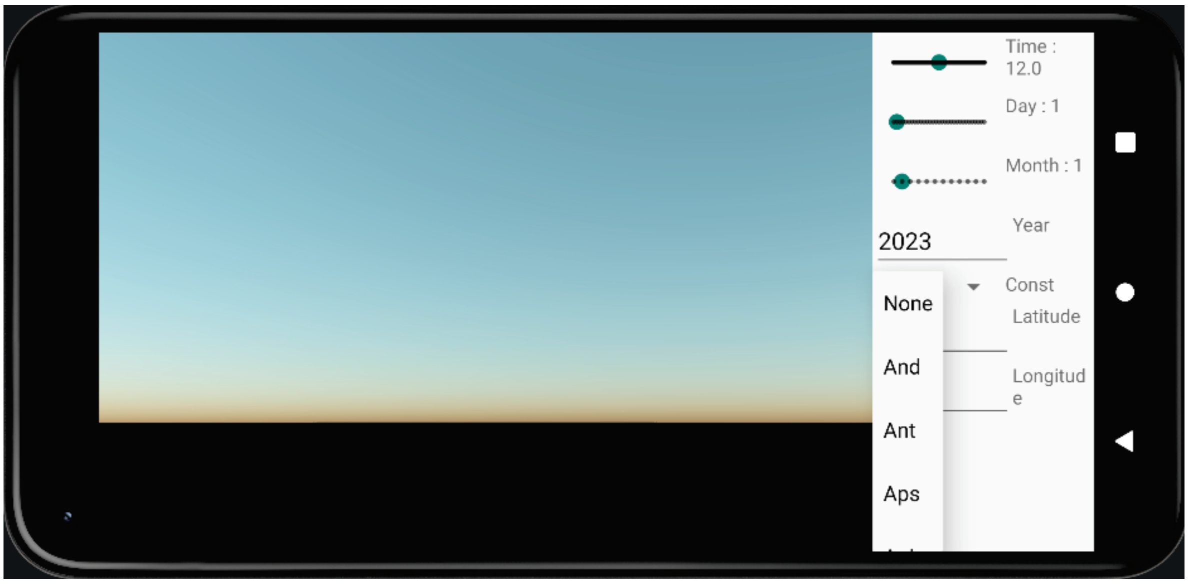


```
▼ Algorithm                                    ×
                0              cam. mode
      1.400                    exposure
          6                    steps
          6                    light ray steps
                0              sampling

▼ Sky Config                                                       ×
                      0.999                              g
101325.000000                                            atm. pressure
0.000000                                                 temp. (C)
                      1.000                              index of reftract.
700.000        530.000        470.000                    wave lengths

▼ Sun Position    Interface controls                                    ×
2023                                                      Year
                         6                                Month
                         31                               Day
                         21.000                           Decimal Time
41.545448                                                 Latitude
-8.426507                                                 Longitude
0                                                         Meridian
310.922                        -8.096                     sun angles
```

Figure 30: User interface for the desktop application



Time : 12.0
Day : 1
Month : 1
2023 Year
Const
None
And
Ant
Aps
Latitude
Longitud e

Figure 31: User interface for the android application

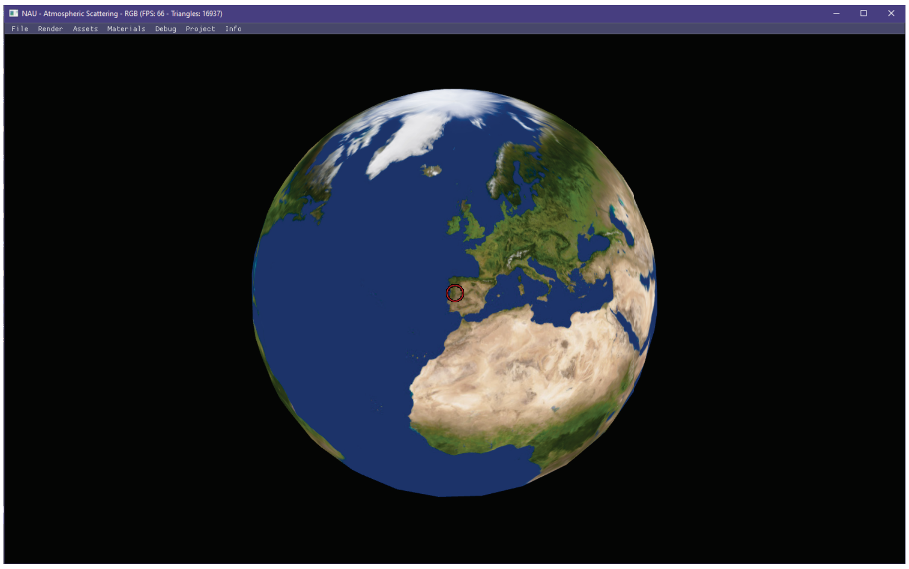

NAU - Atmospheric Scattering - RGB (FPS: 66 - Triangles: 16937)
File
Render
Assets
Materials
Debug
Project
Info

Figure 32: Selection of a location on a map

3.6.4 *Selecting the location of the observer on a map*

To allow for an easier input of the latitude and longitude of a user, a way of selecting said user location on a map was implemented.

A sphere is placed at the origin at the world, textured as the Earth obtained from INOVE. A red circle is added in an orthogonal manner in the middle of the viewport, as we can see in figure 32.

The camera behavior is what is often referred to as an orbit camera. In practice, this means that the view vector of the camera is always directed towards the center of the planet, keeping it permanently in view. In this specific case, the camera is set to a fixed distance from the sphere.

While the use case is different, the position of the camera is, in practice, defined in the horizontal coordinate system, detailed in section 2.3.3. If we make the parallel with figure 14, the observer becomes the Earth, and the star becomes the camera. This makes it very easy to calculate what longitude and latitude lie in the center of the camera, since these are directly proportional to the azimuth and altitude of the camera.

For the longitude, we only need to subtract PI from the azimuth angle before converting it from radians to degrees, since the former varies from -180º to 180º, while the latter varies from 0 to $2 * \pi$.

For the latitude, we only need to convert from radians to degrees.

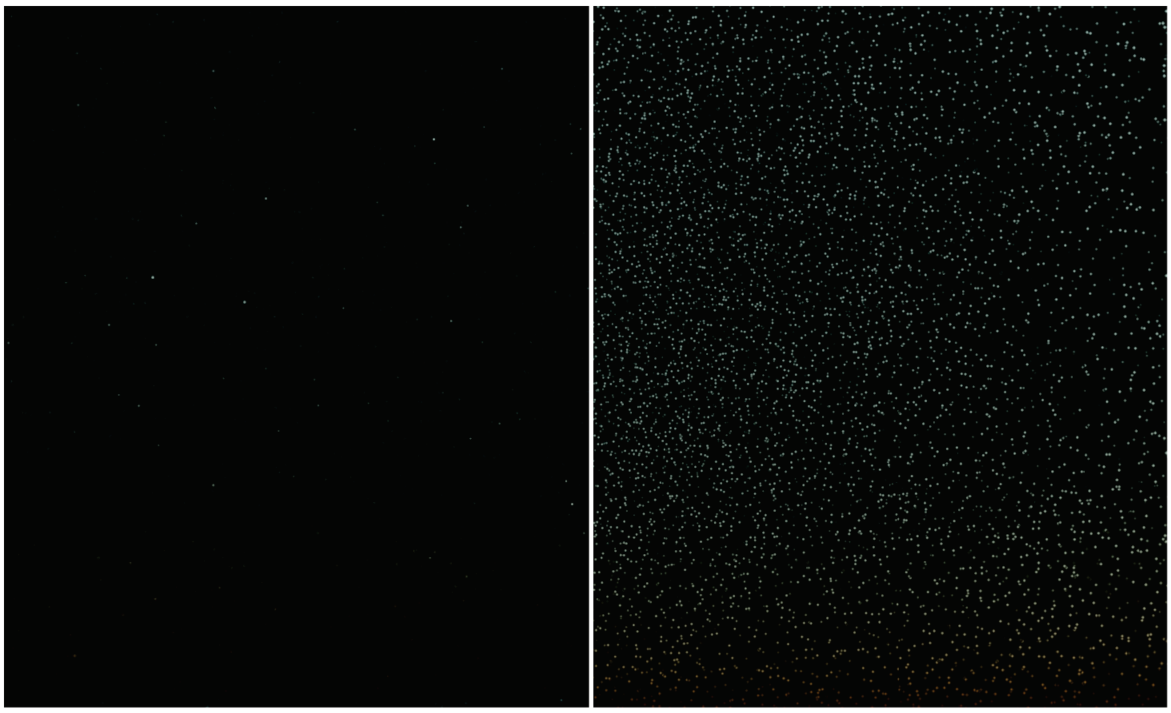


Figure 33: Default view of the sky (left) and view with minimum visible brightness adjusted (right)

3.6.5 *Star visibility control*

The developed application gives the user the option to choose the magnitude of the dimmest star that should be visible. This value is by default 7, which is approximately the dimmest star we can see with the naked eye.

The output is then scaled so that the dimmest star in the specified range has a brightness of 0, the brightest star has a brightness of 1, and other stars fall in between. This effect can be seen in figure 33.

3.6.6 *Planet visibility control*

As in real life, most of the planets are essentially invisible to the naked eye. With that in mind, for visualization purposes, a slider was added to the interface that allows the user to scale the planets up, as can be seen in figure 34.

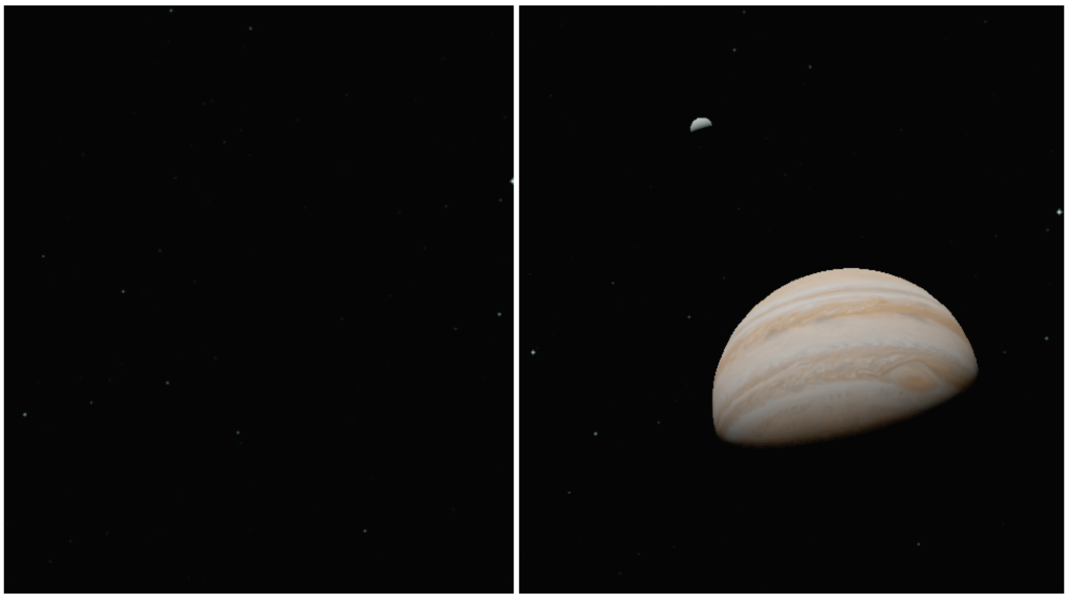


Figure 34: The same view of the sky, at normal (left) and enlarged planet scale(right)

3.6.7 *Constellations*

In the modern strict sense of the word, a constellation is an area of the sky with defined boundaries, and example of which we can see in figure 35. This means that any star within this region belongs to that constellation, meaning that for the selected star catalogue each constellation typically exceeds a count of 1000 stars.

Of course, when most people think of a constellation they think of the groupings of stars that form a perceived pattern or outline, typically representing an animal, mythological subject, or inanimate object, which we can also see in the same figure.

The 88 official constellations, which cover the entire sky dome with contiguous borders, were defined in the early 20th century by the International Astronomical Union (IAU), and while many others were defined throughout the ages, mostly for a more spiritual end, only the ones defined by the IAU will be considered (Union).

In the application, the user can select a specific constellation from a drop down menu, which will result in what we can see in figure 36.

Firstly, only the stars within that constellation are drawn, which is as simple as only rendering the quads of the stars that belong that constellation, an information present in each line of the star catalogue.

As for the artwork, I could not find any resource online that described the constellations by its connections, that is, a list of stars that are connected to each other.

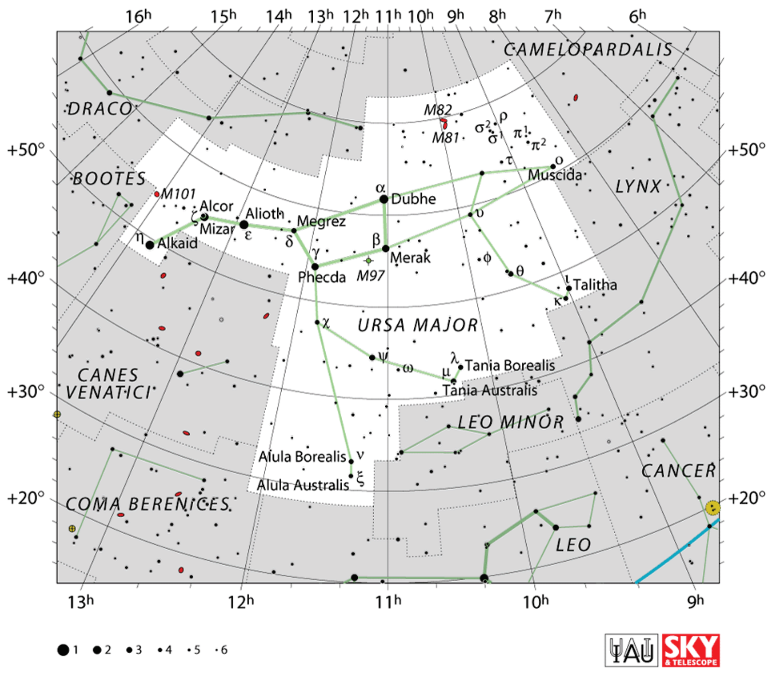


Figure 35: The boundaries of the Ursa Major constellation. Source: IAU

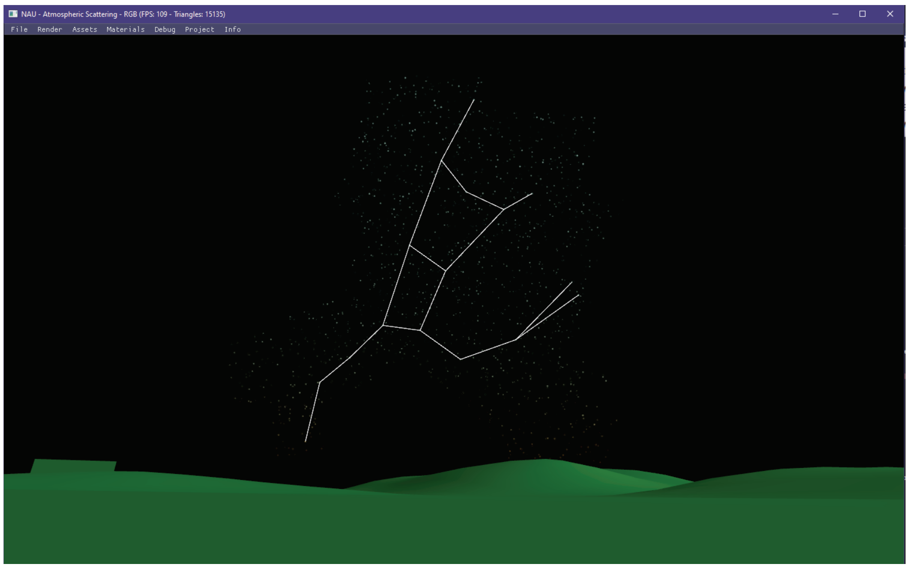

NAU - Atmospheric Scattering - RGB (FPS: 109 - Triangles: 15135)
File Render Assets Materials Debug Project Info

Figure 36: Focusing on the Ursa Major constellation, as well as its corresponding artwork

So, for each constellation, a file that contains this information was created. For this, I had to manually cross reference the constellation images in IAU, similar to figure 35, and get, for each star in the constellation, its identifier in the Hipparchus catalogue, from the Wikipedia page on each constellation, which has a table with both these informations, the Hipparchus identifier and the identifier of the star in the constellation, usually identified by a Greek letter and/or name.

Here we can see a small section of this file showing two constellations:

```
Ant
53502,51172
51172,46515
Aps
81065,81852
81852,80047
80047,72370
```

Each pair of values in a line represents two stars that should be connected for the constellation detailed directly above it. When the user selects a constellation, a line is drawn between each pair of stars, and overlaid over the final result, including the day sky.

3.6.8 *Placing the virtual camera based on a smartphone's position sensors*

An interesting use case for the use of this app in a smartphone is the use of its position sensors to place the virtual camera in the application. To be more precise, this would mean that pointing a phone at the sky, as if taking a picture, would show the corresponding render of the sky in the phone's screen. For this we need to know the phone's position in relation to the Earth. Luckily, essentially any smartphone has the necessary sensors to align itself in relation to Earth's magnetic field.

Google provides in its developer documentation a standard code sample on how to poll these sensors (Google, 2023). This sample fills a 3 position float array with 3 values: an azimuth, a pitch and a rotation, each describing the phone's position in relation to one of the phone axis.

For the OpenGL camera, what we need is a view direction, that is, the vector that represents the direction to which the camera is facing. The android API provides easy access to the position sensors of the smartphone, including the geomagnetic sensor that allows us to position the phone in relation to the Earth. As described in the position sensors' Android Documentation, we can get the device's orientation as a trio of values we'll call "orientation angles", composed of azimuth, pitch and roll, in that order.

To get these values, we first must make sure the Java class that corresponds to the main activity of the application implements the *SensorEventListener* interface. In this class initialization, we get a reference to the SENSOR_SERVICE of the smartphone, represented in Java by the "SensorManager" class.

Following this, we register listeners to both the accelerometer and magnetic field sensor. In this context, a listener is a function that will poll the sensor at regular intervals. The intervals are predefined, and in this implementation the *SENSOR_DELAY_GAME* was chosen, which Google recommends for real time interactive applications. The specific value is smartphone vendor dependent. This should be done in the "onResume" function, so these listeners are registered every time the app comes into focus. They should also be unregistered when the app is "paused", to account for the app losing focus. These can be seen in listing 3.2. We also respectively *resume* and *pause* the openGLView, the Android openGL abstraction.

```

    protected void onResume() {
        super.onResume();
        openGLView.onResume();

        Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
        if (accelerometer != null) {
            sensorManager.registerListener(this, accelerometer,
                    SensorManager.SENSOR_DELAY_GAME);
        }
        Sensor magneticField = sensorManager.getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
        if (magneticField != null) {
            sensorManager.registerListener(this, magneticField,
                    SensorManager.SENSOR_DELAY_GAME);
        }
    }

    protected void onPause() {
        super.onPause();
        openGLView.onPause();
        sensorManager.unregisterListener(this);
    }
```

Listing 3.2: Handling the registering of the Android sensor listeners

Finally, we need to implement the *onSensorChanged* method. This method will trigger every time a change in the sensor is detected, in conjunction with the listener functions, and will copy the values the sensor is outputting to a local array in the class.

We now have the values of the sensors, but they are raw data with little meaning for a human. However, the *SensorManager* class provides two utility functions to fix that.

The first one is *getRotationMatrix*, which transforms the local coordinates of the phone into global coordinates in relation to the Earth. The second one is *getOrientation*, which computes the device's orientation as 3 values, the orientation angles mentioned previously in this section, from the rotation matrix calculated in the previous step. These two steps can be seen in the listing 3.3.

```java
public void onSensorChanged(SensorEvent event) {
    if (event.sensor.getType() == Sensor.TYPE_ACCELEROMETER) {
        System.arraycopy(event.values, 0, accelerometerReading,
                0, accelerometerReading.length);
    } else if (event.sensor.getType() == Sensor.TYPE_MAGNETIC_FIELD) {
        System.arraycopy(event.values, 0, magnetometerReading,
                0, magnetometerReading.length);
    }
    updateOrientationAngles();
}

public void updateOrientationAngles() {
    // Update rotation matrix, which is needed to update orientation
angles.
    SensorManager.getRotationMatrix(rotationMatrix, null,
            accelerometerReading, magnetometerReading);

    SensorManager.getOrientation(rotationMatrix, orientationAngles);
}
```

Listing 3.3: Code to read and transform sensor data

To place the camera in the world, we need to, from the orientation angles, obtain a 3D vector that represents the direction the camera is pointing at, which we'll call "view vector". This vector is in fact quite simple to obtain using the equations from section 2.3.5.

4

CONCLUSIONS AND FUTURE WORK

4.1 CONCLUSIONS AND PROSPECT FOR FUTURE WORK

This work focus on displaying the celestial elements accurately, in terms of position, given a latitude, longitude and date. Accurately simulating the positions of celestial elements in the sky at an arbitrary time is a challenge that involves multiple disciplines of study.

Multiple free publicly available tools and resources were used, namely a star catalogue and 2 algorithms to calculate the position of the moon, sun and planets.

This work mainly aims to provide a description on how to accurately compute the position of various celestial elements in the sky. For visualization purposes, two applications were developed, one for desktop and one for Android.

These applications take into account atmospheric scattering during the day and atmospheric attenuation during the night. In terms of visualization, besides the accurate positioning of the celestial bodies, it also adds some non realistic features, such as changing the scale of planets, control the brightness of stars and draw constellation art work. While non realistic, these features allow for a better perception of the celestial elements, and let us leverage the fact that we are in a virtual environment.

Future work could include the visualization of deep sky elements, such as other galaxies, nebulae and star clusters, and differentiation of stars based on their spectral type. While not the focus of this work, it could also be of value to implement a more accurate multiple scattering algorithm. Presenting information about the celestial body which is hovered with the mouse would also be an interesting addition, substantially increasing interactivity. Finally, obtaining performance metrics for each component of the visualization would be informative, as well as allow for an easier customization of the simulation for a low powered system, if so desired.

BIBLIOGRAPHY

Andeggs. 3d spherical coordinates, 2022. URL https://en.wikipedia.org/wiki/Spherical_coordinate_system#/media/File:3D_Spherical.svg. [Accessed February 3rd, 2023].

GUSTAV Bodare and EDVARD Sandberg. *Efficient and Dynamic Atmospheric Scattering*. PhD thesis, MS dissertation, Chalmers University of Technology, 2016.

Eric Bruneton and Fabrice Neyret. Precomputed atmospheric scattering. *Computer Graphics Forum*, 2008.

M.T. Brück. The cambridge concise history of astronomy. *Endeavour*, 23, 1999. ISSN 01609327. doi: 10.1016/s0160-9327(00)80036-8.

Columbia University: Department of Computer Science. Oren-nayar reflectance model. URL https://www.cs.columbia.edu/CAVE//projects/oren/. (accessed: 07.10.2022).

J.J.L.F. de Lalande. *Histoire céleste française*. Number vol. 1 in Histoire céleste française. De L'Imprimerie De La République, 1801.

Yoshinori Dobashi, Tomoyuki Nishita, Kazufumi Kaneda, and Hideo Yamashita. A fast display method of sky colour using basis functions. *The Journal of Visualization and Computer Animation*, 8(2):115–127, 1997.

Yoshinori Dobashi, Tsuyoshi Yamamoto, and Tomoyuki Nishita. Interactive rendering of atmospheric scattering effects using graphics hardware. pages 99–107, 2002.

Oskar Elek and Petr Kmoch. Real-time spectral scattering in large-scale natural participating media. pages 77–84, 05 2010.

European Space Agency. Gaia archive, a. URL https://gea.esac.esa.int/archive/. (accessed: 03.02.2023).

European Space Agency. The hipparcos and tycho catalogues, b. URL https://www.cosmos.esa.int/web/hipparcos/catalogues. (accessed: 03.02.2023).

Tomasz Gałaj, Filip Pietrusiak, Marek Galewski, Rafał Ledzion, and Adam Wojciechowski. Hybrid integration method for sunlight atmospheric scattering. *IEEE Access*, 9:40681–40694, 2021.

Google. Distribution dashboard. URL https://developer.android.com/about/dashboards#OpenGL. (accessed: 07.10.2022).

Google. Position sensors, 2023. URL https://developer.android.com/guide/topics/sensors/sensors_position. (accessed: 03.02.2023).

Jörg Haber, Marcus Magnor, and Hans-Peter Seidel. Physically based simulation of twilight phenomena. *Transactions on Graphics, v.24, 1353-1373 (2005)*, 24, 10 2005.

Lukas Hosek and Alexander Wilkie. An analytic model for full spectral sky-dome radiance. *ACM Trans. Graph.*, 31(4), jul 2012.

INOVE. Solar textures. URL https://www.solarsystemscope.com/textures/. (accessed: 18.11.2022).

Kazufumi Kaneda, Takashi Okamoto, Eihachiro Nakamae, and Tomoyuki Nishita. Photorealistic image synthesis for outdoor scenery under various atmospheric conditions. *The Visual Computer*, 7, 1991.

R. Victor Klassen. *Modeling the Effect of the Atmosphere on Light*, volume 6. Association for Computing Machinery, 1987.

Barry M. Lasker, Conrad R. Sturch, Brian J. McLean, Jane L. Russell, Helmut Jenkner, and Michael M. Shara. The Guide Star Catalog. I. Astronomical Foundations and Image Processing. 99:2019, June 1990. doi: 10.1086/115483.

Diogo Lopes and António Ramires Fernandes. Atmospheric scattering - state of the art. 11 2014.

Gustav Mie. Beiträge zur optik trüber medien, speziell kolloidaler metallösungen. *Annalen der Physik*, 330(3):377–445, 1908. doi: https://doi.org/10.1002/andp.19083300302. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/andp.19083300302.

NASA. Yale bright star catalogue. URL http://tdc-www.harvard.edu/catalogs/bsc5.html. (accessed: 03.02.2023).

David Nash. The astronomy nexus. URL https://www.astronexus.com/hyg. (accessed: 17.11.2022).

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, SIGGRAPH 1993, Anaheim, CA, USA, August 2-6, 1993*, pages 175–182. ACM, 1993.

Tomoyuki Nishita, Yoshinori Dobashi, Kazufumi Kaneda, and Hideo Yamashita. Display method of the sky color taking into account multiple scattering. *Pacific Graphics*, pages 117–132, 1996.

NREL. Sun and moon positioning algorithm. 2012. URL http://archive.gamedev.net/archive/reference/articles/article2093.html.

Sean O'Neil. Real-time atmospheric scattering. 2004. URL http://archive.gamedev.net/archive/reference/articles/article2093.html.

Sean O'Neil. *GPU gems 2 : programming techniques for high-performance graphics and general-purpose computation*, volume 2, chapter 16 - Accurate Atmospheric Scattering. 2005.

R. Perez, R. Seals, and J. Michalsky. All-weather model for sky luminance distribution—preliminary configuration and validation. *Solar Energy*, 50(3):235–245, 1993.

Edward Charles Pickering, Annie Jump Cannon, and Harvard College Observatory. Henry draper catalogue. URL https://openlibrary.org/books/OL20434324M/The_Henry_Draper_Catalog. (accessed: 07.10.2022).

A. 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, SIGGRAPH 1999*, 1999.

T Schafhitzel, Martin Falk, and Thomas Ertl. Real-time rendering of planets with atmospheres. *Journal of WSCG 2007*, 15:91–98, 2007.

Paul Schlyter. How to compute planetary positions. URL https://stjarnhimlen.se/comp/ppcomp.html. (accessed: 18.11.2022).

Tan Shihan, Zhang Jianwei, Lin Yi, Liu Hong, Yang Menglong, and Ge Wenyi. Adaptive volumetric light and atmospheric scattering. *Plos one*, 15(11):e0242265, 2020.

Hon. J.W. Strutt. Xv. on the light from the sky, its polarization and colour. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(271):107–120, 1871.

TWCarlson. A schematic diagram of the terms "azimuth" and "altitude" as they relate to the viewing of celestial objects., 2020. URL https://en.wikipedia.org/wiki/Horizontal_coordinate_system#/media/File:Azimuth-Altitude_schematic.svg. [Accessed February 3rd, 2023].

International Astronomical Union. The constellations. URL https://www.iau.org/public/themes/constellations/. (accessed: 03.02.2023).

U.S. Naval Observatory and H.M. Nautical Almanac Office. Astronomical almanac online. URL http://asa.hmnao.com/. (accessed: 07.10.2022).

Joey de Vries. Hdr, 2014. URL https://learnopengl.com/Advanced-Lighting/HDR. (accessed: 20.06.2023).

Georg Zotti, Alexander Wilkie, and Werner Purgathofer. A critical review of the preetham skylight model. In *WSCG*, pages 23–30. University of West Bohemia, 2007.

Part III

APPENDICES

A

LISTINGS

A.1 COMPLETE SAMPA INPUTS

These inputs are necessary as while they do not change our simulation in a meaningful way the app will crash otherwise, as default values are not provided by SAMPA.

```c
    sampa_data sampa;  //declare the SAMPA structure
    int result;

    //enter required input values into SAMPA structure

    sampa.spa.year          = 2009;          //Valid range: -2000 to 6000, int
    sampa.spa.month         = 7;             //Valid range: 1 to  12, int
    sampa.spa.day           = 22;            //Valid range: 1 to  31, int
    sampa.spa.hour          = 1;             //Valid range: 0 to  24, int
    sampa.spa.minute        = 33;            //Valid range: 0 to  59, int
    sampa.spa.second        = 0;             //Valid range: 0 to <60, double
    sampa.spa.timezone      = 0;             //Valid range: -18 to 18 hours, double
    sampa.spa.delta_ut1     = 0;             //Fractional second difference between UTC and UT, derived from observation only, reported in http://maia.usno.navy.mil/ser7/ser7.dat
    sampa.spa.delta_t       = 66.4;          //Difference between earth's rotation and terrestrial time. Only obtained from observation, reported in http://maia.usno.navy.mil/ser7/ser7.dat, double
    sampa.spa.longitude     = 143.36167;     //Valid range: -180  to  180 degrees, double
    sampa.spa.latitude      = 24.61167;      //Valid range: -90   to   90 degrees, double
```

```
  sampa.spa.elevation      = 0;             //Valid range: -6500000 or
 higher meters, double
  sampa.spa.pressure       = 1000;          //Annual average local pressure
  in millibars, valid range: 0 to 5000, double
  sampa.spa.temperature    = 11;            //Annual average local
 temperature, valid range: -273 to 6000 degrees Celsius
  sampa.spa.atmos_refract = 0.5667;        //Atmospheric refraction at
 sunrise and sunset (0.5667 deg is typical), valid range: -5 to 5 degrees,
  double
  sampa.function           = SAMPA_NO_IRR;  //Whether solar irradiances
 should be calculated, SAMPA_ALL or SAMPA_NO_IRR;

  //call the SAMPA calculate function and pass the SAMPA structure
  result = sampa_calculate(&sampa);
```

A.2 PLANET POSITION'S CODE

The following code is used to calculate the position of the planets, based on Paul Schlyter's work (Schlyter).

```

typedef struct elements {
  double N;
  double i;
  double w;
  double a;
  double e;
  double M;
  double RA;
  double DEC;
  double r;
}*Elements;

float compute_day(int year, int month, int day, float hour) {
  int id = 367 * year - 7 * (year + (month + 9) / 12) / 4 - 3 * ((year + (
   month - 9) / 7) / 100 + 1) / 4 + 275 * month / 9 + day - 730515;
  float d = id + hour / 24;
  return d;
}
```

```
void computePositionSun(Elements elements, float d, double* xs, double* ys)
    {
  double E = elements->M + elements->e * (180 / PI) * sind(elements->M) *
    (1.0 + elements->e * cosd(elements->M));

  double xv = cosd(E) - elements->e;
  double yv = sqrt(1.0 - elements->e * elements->e) * sind(E);

  double v = atan2d(yv, xv);
  double r = sqrt(xv * xv + yv * yv);

  double lonsun = v + elements->w;

  *xs = r * cosd(lonsun);
  *ys = r * sind(lonsun);
}

void computePositionPlanet(Elements elements, float d, double xs, double ys,
     double *distance, double *heliox, double* helioy, double* helioz) {

  double ecl = 23.4393 - 3.563E-7 * d;
  double E;
  if (elements->e < 0.06) {
    E = elements->M + elements->e * (RADEG)*sind(elements->M) * (1.0 +
    elements->e * cosd(elements->M));
  }
  else {
    int n = 0;
    double E0 = 0.0;
    double E1;
    while (n < 100) {
      E1 = E0 - (E0 - elements->e * RADEG * sind(E0) - elements->M) / (1 -
    elements->e * cosd(E0));
      if (abs(E1 - E0) < 0.001) {
        break;
      }
      else {
        E0 = E1;
      }
      n++;
```

```
        }
        E = E1;
        if (n == 100) std::cout << "Didn't converge" << std::endl;
    }

    double xv = elements->a * (cosd(E) - elements->e);
    double yv = elements->a * (sqrt(1.0 - elements->e * elements->e) * sind(E)
    );

    double v = atan2d(yv, xv);
    double r = sqrt(xv * xv + yv * yv);

    double xh = r * (cosd(elements->N) * cosd(v + elements->w) - sind(elements
    ->N) * sind(v + elements->w) * cosd(elements->i));
    double yh = r * (sind(elements->N) * cosd(v + elements->w) + cosd(elements
    ->N) * sind(v + elements->w) * cosd(elements->i));
    double zh = r * (sind(v + elements->w) * sind(elements->i));
    *heliox = xh;
    *helioy = yh;
    *helioz = zh;

    double lonecl = atan2d(yh, xh);
    double latecl = atan2d(zh, sqrt(xh * xh + yh * yh));

    double lon_corr = 3.82394E-5 * -d;

    lonecl += lon_corr;

    xh = r * cosd(lonecl) * cosd(latecl);
    yh = r * sind(lonecl) * cosd(latecl);
    zh = r * sind(latecl);

    double xg = xh + xs;
    double yg = yh + ys;
    double zg = zh;

    double xe = xg;
    double ye = yg * cosd(ecl) - zg * sind(ecl);
    double ze = yg * sind(ecl) + zg * cosd(ecl);

    double RA = atan2d(ye, xe);
```

```
  double Dec = atan2d(ze, sqrt(xe * xe + ye * ye + ze * ze));
  double rg = sqrt(xe * xe + ye * ye + ze * ze);
  *distance = rg;

  elements->RA = RA;
  elements->DEC = Dec;
  elements->r = rg;

}
```

These functions are used as follows, for the example of Mercury:

```
Elements sunElements = (Elements)malloc(sizeof(struct elements));
sunElements->N = 0.0;
sunElements->i = 0.0;
sunElements->w = 282.9404 + 4.70935E-5 * d;
sunElements->a = 1.000000;
sunElements->e = 0.016709 - 1.151E-9 * d;
sunElements->M = 356.0470 + 0.9856002585 * d;

Elements mercuryElements = (Elements)malloc(sizeof(struct elements));
mercuryElements->N = 48.3313 + 3.24587E-5 * d;
mercuryElements->i = 7.0047 + 5.00E-8 * d;
mercuryElements->w = 29.1241 + 1.01444E-5 * d;
mercuryElements->a = 0.387098;
mercuryElements->e = 0.205635 + 5.59E-10 * d;
mercuryElements->M = 168.6562 + 4.0923344368 * d;

  double xs, ys;
computePositionSun(sunElements, d, &xs, &ys);

  double distance, xh, yh, zh;
  computePositionPlanet(mercuryElements, d, xs, ys, &distance, &xh, &yh, &
 zh);
```

After this, the Right Ascension and Declination are present in the fields RA and DEC of the $mercuryElements$ struct, respectively.