Atmospheric Scattering - A Simulation in OpenGL implementing Rayleigh Scattering

Harits Nur Fauzan
KTH Royal Institute of Technology
Stockholm, Sweden
mhnf@kth.se

Qinbai
KTH Royal Institute of Technology
Stockholm, Sweden
baiq@kth.se

Benjamin Esdor
KTH Royal Institute of Technology
Stockholm, Sweden
esdor@kth.se

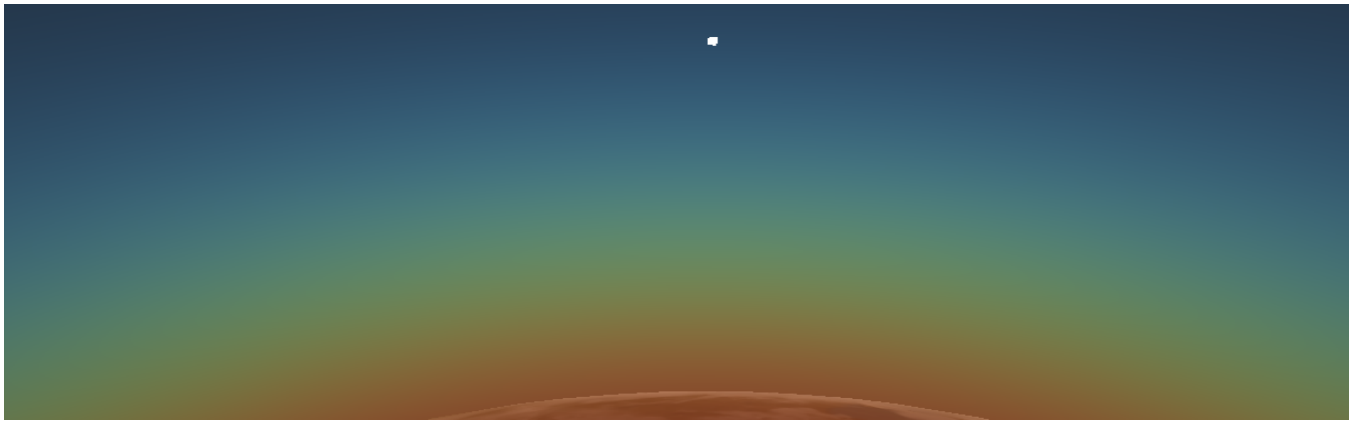

Figure 0: This image shows the Rayleigh scattering from the planet to the sun. At the bottom you can see the earth, surrounded by the atmosphere, which is scattered into several layers of light (blue, green and red).

Abstract

In this project we set out to create a simulation of the atmosphere in terms of how light that is emitted by a light source (in our case the sun) is scattered when the light enters the atmosphere. The main problem was to generate a realistic atmosphere with the proper light conditions according to the algorithms and equations presented by previous research. Rayleigh and Mie scattering are known methods which we investigated and finally implemented Rayleigh scattering. For the implementation we wrote code in C++ using the OpenGL library.

Keywords: Atmosphere, Rayleigh Scattering, OpenGL

1 Introduction

One difficult problem in the area of computer graphics has always been the generation of a realistic atmospheric scattering for outdoor or planetary events (Pharr and Fernando, 2005, Chapter 16). To place a single planet in the universe is not enough: there is a deep darkness surrounding it. If a camera perspective was fixed at the bottom of the ground and a tidal sequence was simulated on the horizon with a sun, no autonomous atmosphere would be established. In this paper we want to investigate the phenomenon of an atmosphere and how the light and the colors adapt and influence depending on the viewing position and the rays.

The two most common forms of scattering in the atmosphere are called Rayleigh scattering and Mie scattering. The Rayleigh and Mie scattering is a phenomenon that describes how atmospheric scattering works; how the sunlight is being scattered around by particles and molecules when it enters the atmosphere, resulting in a blue sky during the daytime and red-orange sky during sunrises and sunsets. Rayleigh scattering describes how light interacts with smaller molecules such as oxygen and Mie scattering describes the interaction for larger molecules such as pollution and aerosols (Nishita et al., 1993). Rayleigh scattering scatters light with shorter wavelengths (blue, then green, finally red) more heavily, thus resulting in the blue sky we all know. Mie scattering scatters light equally regardless of the wavelength, causing the sky to look gray during a hazy day. We would like to simulate this phenomenon using the model introduced in (Pharr and Fernando, 2005, Chapter 16). by using C++ and OpenGL to simulate the Atmospheric Scattering in a simulation. The finished implementation can be accessed on our GitHub Repository[1] [2].

2 Related Work

The model by O'Neil assumes that the camera is always held very close to the ground in order to assume a constant density throughout all altitudes, which simplifies the scattering equation from (Nishita et al., 1993). Based on these foundation, we are going to implement the following three equations:

[1] Git repo: https://github.com/mharitsnf/AtmosphericScattering
[2] Git repo main development: https://github.com/mharitsnf/AtmosphericScattering/tree/harits-dev

The Phase Function:

$$F(\theta, g) = \frac{3 * (1 - g^2)}{2 * (2 + g^2} * \frac{1 + \cos^2 \theta}{(1 + g^2 - 2 * g * \cos \theta)^{\frac{3}{2}}} \quad (1)$$

The Out-Scattering Equation:

$$t(P_a P_b, \lambda) = 4\pi * K(\lambda) * \int_{P_a}^{P_b} exp\left(\frac{-h}{H_0}\right) ds \quad (2)$$

The In-Scattering Equation:

$$I_F(\lambda) = I_S(\lambda) * K(\lambda) * F(\theta, g) * \int_{P_a}^{P_b} \left(exp\left(\frac{-h}{H_0}\right) * exp(-t(PP_c, \lambda) - t(PP_a, \lambda))\right) ds \quad (3)$$

Sebastian Lague has created an implementation of this chapter and published it on his YouTube channel (Lague, 2020). He used Unity to implement the simulation and had a more sophisticated setup with other post-processing applications such as the ocean and the clouds.

3 Implementation and Result

We present the steps taken in the implementation of the atmospheric scattering simulation in OpenGL. We used OpenGL for our graphics API due to its extensive library and it being a middle-level API would be beneficial for our learning as well. A more thorough explanation on the technicalities can be accessed on the project blog [3].

3.1 Linking The Post-Processing and The OpenGL Environment

The interactivity of the application is handled on OpenGL and is run on CPU. We implemented a basic interaction through camera movement and rotation for the user to experience the application from multiple perspectives. The OpenGL is also responsible for rendering the models as well.

The post-processing environment runs on shaders and have a separate camera model that cannot be directly translated from OpenGL and C++. Both environments should be interactable through camera movements and rotations, however, initially, they are not linked. To have the application working, a linking is needed.

To keep the input stream centralized, we keep receiving input from the OpenGL part, and send the camera data to the shader every frame instead. The data sent from the OpenGL part are the camera position and the camera rotations, i.e., the yaw and pitch angles.

The camera model in the post-processing environment consists of the ray origin and the ray direction. The camera position data sent from the OpenGL part acts as the ray origin, while the ray direction is derived from the screen texture coordinates. The screen texture coordinate needs to be adjusted by moving the (0,0) position to the center of the screen and changing the aspect ratio of the screen.

[3]Project blog: https://docs.google.com/document/d/1VtaNd2OQC4ndTJ4I$_r$*ilJaA5PS*81*mg*9*ZAMXCypCPn*2*M*/*edit?usp = sharing*

3.2 Generating the Atmosphere

The atmosphere is completely written on the post-processing environment with data provided from OpenGL. The base of the atmosphere can be written using the ray-sphere intersection algorithm, which makes use of the implicit function of a sphere as following:

$$x^2 + y^2 + z^2 = R^2 \quad (4)$$

The left side of the equation are the coordinates of the Cartesian point while the right side is the radius of the sphere. The coordinates can be any point on the surface of the sphere, which would be the information we need. A ray/point can be represented as:

$$P = O + Dt \quad (5)$$

And hence we can substitute $x^2 + y^2 + z^2$ with $O + Dt$ to get a point within the sphere's surface and get the following equation:

$$O^2 + (Dt)^2 + 2ODt - R^2 = O^2 + D^2t^2 + 2ODt - R^2 = 0 \quad (6)$$

The equation above takes form of the following quadratic equation:

$$f(x) = ax^2 + bx + c \quad (7)$$

Which can be solved by finding the roots with the following equation:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a} \quad (8)$$

$$\Delta = b^2 - 4ac \quad (9)$$

In equation 9, Δ is called the discriminant. The discriminant can then be used to determine whether the equation has two, or one, or no root. Having two, one, or zero root means that there are two, one, or zero points along the ray that intersects the sphere, respectively. By having the points that intersects the sphere, we can determine the length from the camera to the sphere and from the entry point to the exit point. The result can be seen in figure 1.

3.3 The Scattering Equations

The mathematical function of light scattering involves integration. For all of the integrations, we use approximation using weighted sum calculated in a loop to substitute the integration as it would be computationally expensive to calculate the integration.

The first equation to implement is the phase function, which models how a molecule scatters the light according to the angle of the light ray and the view ray and returns the transmittance at a point. There are two type of scattering models, which are Rayleigh and Mie scattering. For this project, the Mie scattering model will be omitted because the equation is similar with just a coefficient difference (Nishita et al., 1993). Rayleigh scattering model is more fitting as well as it models

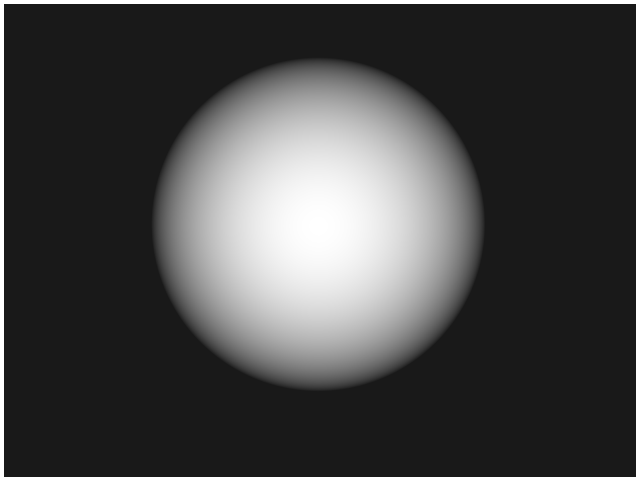

Figure 1: Atmosphere base

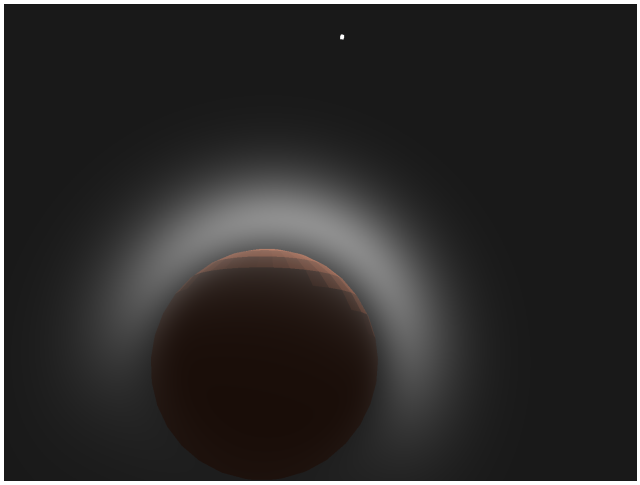

Figure 2: Atmosphere with scattering implemented

air molecules smaller than the light rays while the Mie scattering model describes larger molecules such as aerosols, which would be fitting for clouds (**?**).

The phase function is taking the optical depth into account, which is a value that describes how many air particles are there in the path of the light along the ray. if the optical depth is zero, the light will not be scattered and all will arrive at the point. However, if we gradually increase the optical depth, the amount of light scattered will be exponentially decreased (**?**). We use an exponential function to simulate the phase function as follows.

$$transmittance = e^{-opticalDepth} \tag{10}$$

The next equation to implement is the out-scattering equation, which explains how much light had been scattered away by the atmosphere. In our case, the point can be any point within the atmosphere or the camera itself. The function will only be called if the ray goes through the atmosphere and the function will return the optical depth along the ray. Within the function, we sampled 20 points from the ray and we calculated the local density of each point. The function returns the total density from all of the point that had been calculated, as that density value would be used to calculate the phase function.

Algorithm 1 Out-Scattering Equation

Input: Ray origin, ray direcrtion, ray length
Output: The density along the ray

1: Initialize the first point as the ray origin
2: Set the step size by equally dividing the range between each points
3: Initialize the output values
4: **for** *each points* **do**
5: Calculate local density at current point
6: Add the local density to the output value
7: Move the point to the next point
8: **end for**
9: Return the density along the ray

The atmosphere will become denser when closer to the surface of the planet, and become exponentially less dense the higher up it is from the surface. Thus, the local density at point can be calculated using the following equation:

$$localDensity = e^{-atmosphereHeight * densityFallof}$$

The last equation to implement is the in-scattering equation, which describes how much light enters a point after being scattered away. The out-scattering function is called inside this function as we need to take into account how much light has been scattered away when it traveled from the sun to the point, and when it traveled from the point to the camera. The result can be seen in figure 2.

Algorithm 2 In-Scattering Equation

Input: Ray origin, ray direcrtion, ray length, original color
Output: RGB color

1: Initialize the first point as the ray origin
2: Set the step size by equally dividing the range between each points
3: Initialize the output values
4: **for** *each points* **do**
5: Calculate the length from the point to the sun
6: Calculate the optical depth from the point to the sun
7: Calculate the optical depth from the point to the camera
8: Calculate the transmittance using the phase function
9: Calculate the local density at that point
10: Add to the output value by multiplying the current local density, transmittance, scattering coefficients, and step size
11: Move the point to the next point
12: **end for**
13: Return the RGB color

3.4 Adding The Earth Model

To finalize the sphere, we include the Assimp-Importer-Library in order to import a planetary texture file and map

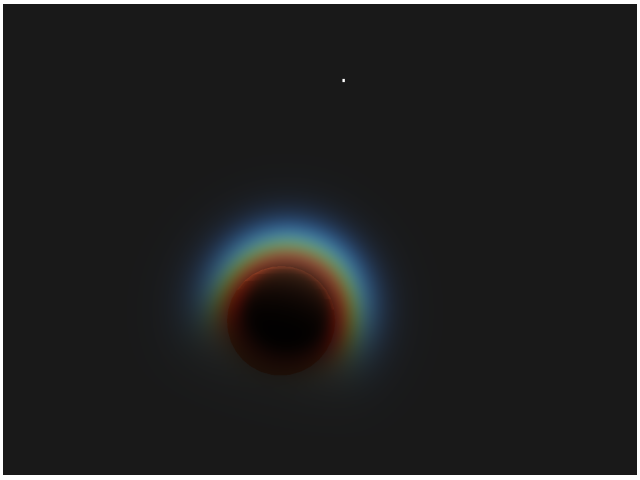

Figure 3: Atmosphere with scattering and color implemented

it onto our sphere. Our decision fell upon a texture of the earth from the webpage 123rf.com [4].

3.5 Adding Color to The Atmosphere

The Rayleigh scattering model scatters the light ray at different rates, and the ratio is $1/wavelength^4$, resulting to the scattering coefficients (Pharr and Fernando, 2005, Chapter 16). We separated the wavelength into three separate red, green, and blue channels. For each channels, we set the values to 700, 530, and 440 nanometers respectively. We can get the scattering coefficients by inserting the values to the aforementioned equation. The scattering coefficients is calculated on C++ and will be sent to the shader program. The coefficients is used to calculate the in-scattering value and the transmittance value as seen in equation 11. The final result can be seen in figure 3.

$$transmittance = e^{-opticalDepth * scatteringCoefficients} \tag{11}$$

4 Future Works

In this section, we will discuss the limitations of our simulation and how to improve it. We used the simulation of the phase function with an exponential function to model the scattering at a point as we only use Rayleigh scattering in this project. For implementing both Rayleigh and Mie scattering, it would be fitting to implement the phase function as the function is used for both models with a difference in the coefficients.

We created a simplistic simulation of the atmosphere by omitting a lot of other important atmospheric phenomena. For a more sophisticated simulation, other atmospheric phenomena can be simulated as well, including clouds, oceans, and fogs.

We did not utilize depth within the post-processing shader in this project as we could not make it work within the project. As a result, the rays that are being shot from the camera may ignore the planet and do not intersect with it. Due to this, the atmosphere may not look too realistic when seen from the outer space. An implementation of the simulation that uses a depth texture would solve this issue.

We also experienced an issue that the atmosphere might drift away when the camera is located far away from the planet. We suspect that this was due to incorrect ray direction mapping between the C++ camera and the post-processing camera, but we were not able to fix this issue. Further examination in this issue might be needed to clearly identify the problem.

In our study, we focused on the implementation of the scattering and not on the realistic details of the earth. But possible future work would be to evaluate our implementation in perceptual aspects. Some studies chose to show the participants several sets of the rendered stimuli of their work, such as (Gigilashvili et al., 2021) and (Toscani et al., 2020), and have them choose the one perceptually close to either the reference or reality. We may also produce different sets of images with varying density, view angles and light strengths. Another approach conducted by Um et al. included a reference video from a selected real-world setup to create a robust and reliable evaluation of perception (Um et al., 2017). Although their setup is tailored for measuring the perception of liquid simulation methods, their approach of creating a visual accuracy score can be implemented in other types of rendering, such as in this project.

5 Workload Distribution

We divided the project as following: Benjamin is responsible for setting up the C++ project and adding the color to the atmosphere. Harits is responsible for adding the ray-sphere intersection and implementation of the scattering. Qinbai is responsible for setting up the camera and creating an importer for the OBJ models.

[4] https://www.123rf.com/photo_151534030_world-texture-satellite-image-of-the-earth-high-resolution-texture-of-the-planet-with-relief-shading.html

References

Davit Gigilashvili, Weiqi Shi, Zeyu Wang, Marius Pedersen, Jon Yngve Hardeberg, and Holly Rushmeier. 2021. The Role of Subsurface Scattering in Glossiness Perception. *ACM Transactions on Applied Perception*, 18(3):10:1–10:26, May.

Sebastian Lague. 2020. Coding Adventure: Atmosphere, August.

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

Matt Pharr and Randima Fernando. 2005. *GPU Gems 2: Programming techniques for high-performance graphics and general-purpose computation (gpu gems)*. Addison-Wesley Professional.

Matteo Toscani, Dar'ya Guarnera, Giuseppe Claudio Guarnera, Jon Yngve Hardeberg, and Karl R. Gegenfurtner. 2020. Three perceptual dimensions for specular and diffuse reflection. *ACM Trans. Appl. Percept.*, 17(2), may.

Kiwon Um, Xiangyu Hu, and Nils Thuerey. 2017. Perceptual Evaluation of Liquid Simulation Methods. *ACM Transactions on Graphics*, 36(4):1–12, August. arXiv:2011.10257 [cs].