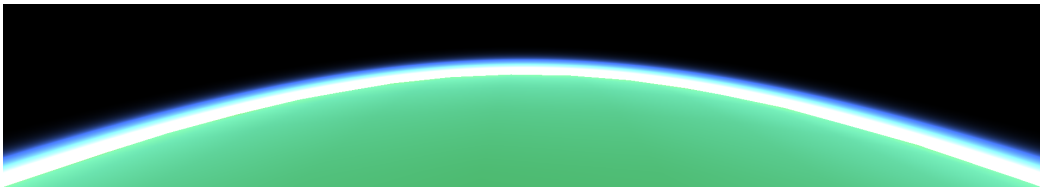
Selection of an Atmospheric Shading Model for Planetary Rendering

Justin Powell
Taylor University

July 2015



1 Abstract

For the accurate rendering of planetary bodies there are several things you need to take into consideration. The main points you must take into account being the terrain geometry, terrain textures, and finally the rendering of the atmosphere. This paper focuses on the Atmospheric rendering portion of planetary rendering. Two methods of rendering were considered: a physically based and an approximation based approach.This paper presents our findings on using the approximation based approach to solve a larger problem.

2 Introduction

For my senior project I am working on the generation and rendering of planetary bodies in a solar system. This includes the generation of terrain meshes, texturing, and atmospheric rendering. This paper focuses solely on atmospheric rendering and the ideas that go into creating a solution that looks convincing enough so as to enhance the scene.

My project as a whole involves the generation and rendering of planetary bodies. Everything from terrain to atmospheres must be taken into consideration. Including being generated and rendered in real time, meaning they must allow for an interactive environment. This limits the design of the algorithms such that I must account for generation times and rendering speeds for the final project. Something that looks great and yet only runs at two

frames per second will not work for my desired project goal. As such, compromises must be made in the end. This is where selecting an appropriate atmospheric rendering algorithm comes into play.

3 Theory

First of all we must understand just how an atmosphere gets its distinctive color scheme. The sky gets its color from a combination of a few different scientific principles. The two main factors are the Rayleigh and Mie scattering effects.[3]

Rayleigh scattering, in short, is the effect of light waves being scattered by minute particles in the atmosphere based on the wavelength of the wave. As light waves enter the atmosphere they collide with gas particles in the atmosphere and are scattered about. Figure 1 shows this scattering effect. As the wave travels further it encounters more and more particles and in turn is scattered again. Some waves are scattered away from the viewpoint of the observer while others are scattered back in, seemingly at random. So how does this scattering give the sky its color? The answer lies in the wavelength of the light that is scattered. Blue light has a very short wavelength on the spectrum, which causes it to be scattered more often over great distances. This blue light is scattered into the viewpoint of the observer giving the sky its blue color. One deviation from that is what happens during a sunrise or sunset. As the sun sits on the horizon they light waves must travel through much more atmosphere than when the sun is at its zenith. Over these greater distances we find that the short wavelength, blue light that was previously scattered into our viewpoint has now been scattered out, leaving the longer wavelengths of red and orange in its place.

The second main factor is Mie scattering. This type of scattering occurs mainly with slightly larger particle aerosols in the air. Particles such as pollution or other aerosols in the air work to scatter light mainly in the direction parallel to the direction that light enters the atmosphere. Looking at Figure 1 we can clearly see this mechanic in effect. Based on this we find that Mie scattering mainly affects the sky color in the direction of the sun in the sky. This is what gives the sun its glare when staring directly towards it.

There are other, smaller factors that affect the color of the sky that the Nishita article mention[2], but are not taken into consideration in this paper.

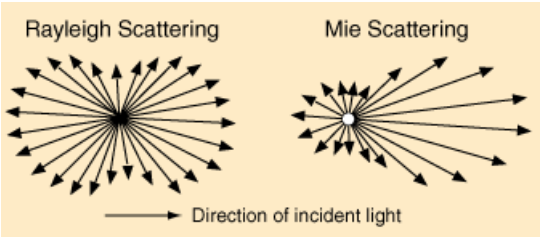


Figure 1: Rayleigh and Mie scattering on a particle[3].

4 Previous Works

For this project I looked at two main forms of atmospheric scattering algorithms that had been done before. First was a physical approach described in the Nishita paper[2]. The second approach is an approximation algorithm designed by Sean O'Neil and is the algorithm we decided to implement in this paper[1].

4.1 Physical Based Solution

The Nishita paper[2] describes a method for coloring the atmosphere viewed from space taking into account atmospheric scattering as well as cloud and sea coloring. Focusing on the atmosphere portion of the paper we have a useful conglomeration of equations and formulas that are used to calculate the color of the atmosphere. The physical approach described in this paper calls for the calculation of the scattering caused by both Rayleigh and Mie scattering. Using the help of multiple lookup tables to store pre-calculated values, scattered light is found by integrating over the optical depth of the light ray to determine the color of both the atmosphere and the earth.

The algorithm takes several things into consideration to simplify the calculations. For the atmosphere they assume that multiple scattering is ignored due to the large number of calculations required and the negligible effect, absorption by the ozone layer is ignored, the density of air molecules and aerosols vary exponentially with altitude, and finally that light travels in a straight line through the atmosphere. However even with these simplifications they algorithm is still very calculation intensive and complex. For a detailed look at the equations and formulas refer to the Nishita paper.

4.2 Approximation Based Solution

The O'Neil solution[1] refers to the Nishita algorithm and simplifies the equations that are used. It eliminates the lookup table from the scattering equation and simplifies other aspects as well to create a solution that can run in real time and at interactive frame rates. To eliminate the lookup table O'Neil uses two functions to replace the x and y dimensions of the lookup table based on his analysis of the data in the lookup table.

To calculate the color of a vertex in the atmosphere mesh the process starts by finding the camera location and vertex location in space. A ray is built between the two and is sampled at multiple points along the length. At each point the scattering and attenuation is calculated based on the height of the point and angle of the light and camera. Several coefficients are used during the calculations for the Rayleigh and Mie scattering as well as the energy of the sun.

5 Solution Decision

For my project I decided to implement the O'Neil solution for atmospheric rendering. After reviewing the two algorithms I found that the O'Neil approximation was suited just fine for my application requirements, mainly being the ease of generating different atmospheres and calculation of the rendering color. As stated previously, this application must be able to run in real-time and at an interactive level. This requires the calculations to be simple and concise so as to not impact the frames rendered per second.

In both solutions each visible vertex of the atmosphere mesh must undergo several calculations to determine the final coloring of the sky in that area. However, as O'Neil states in his description of his algorithm he is able to reduce the number of calculations per vertex to a much smaller number than that of the Nishita algorithm[1]. This drastic reduction in calculation requirements is definitely a must in interactive rendering, especially when you take into consideration the hardware that the software will be running on. Lower end systems may not be able to handle all the calculations that are required of the Nishita solution or other physically based algorithms. When comparing the results of the two algorithms I noticed that even with

the higher calculation requirements of the Nishita solution, the resulting atmospheres looked very similar in final coloring. This led me to conclude that The O'Neil solution would be a more efficient path to go down in the short time I had to work over the summer months.

6 Implementation

The implementation of the O'Neil shader is relatively well documented by O'Neil himself[1] as well as in several other implementations of the algorithm you can find on the web. The shader is split into four different parts that make up the program as a whole, and each part split into its respective vertex and fragment shader. We must take into account the atmosphere coloring while within the atmosphere, when looking at the atmosphere from space, the ground while in atmosphere, and finally the ground while in space. These four parts as a whole make for a seamless experience when moving from the ground to space.

6.1 Atmosphere Mesh

There are two meshes in use for the atmosphere in my project. One is used when within the radius of the atmosphere and another when outside the radius. The reasoning for having two different designs is explained below.

When creating the planetary mesh for the project I settled on the quad sphere based mesh design for the terrain that can be seen in Figure 2. Between a quad sphere design and a uv sphere design I found the quad sphere to have less distortion overall leading to a more even distribution of vertices. I use this same mesh for the atmosphere when the camera position is in space, meaning the user has moved outside of the atmosphere's radius. When looking at the atmosphere from space this is sufficiently tessellated to draw a good looking atmosphere.

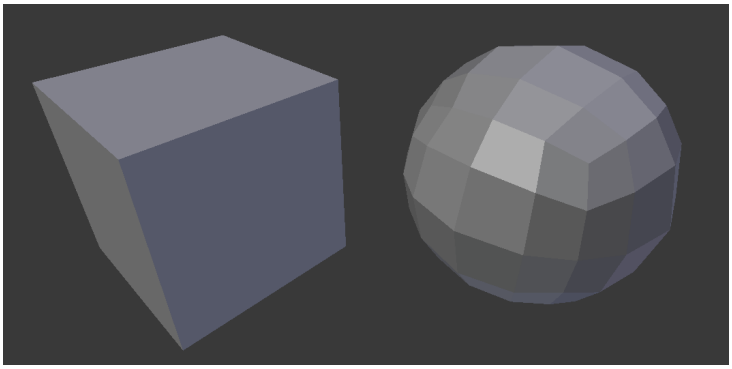
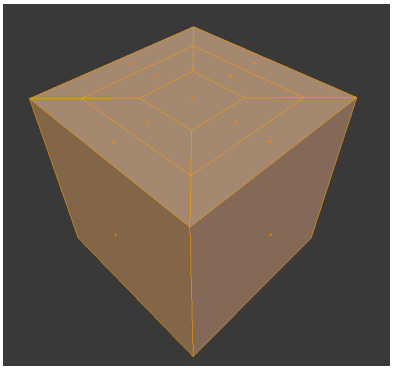


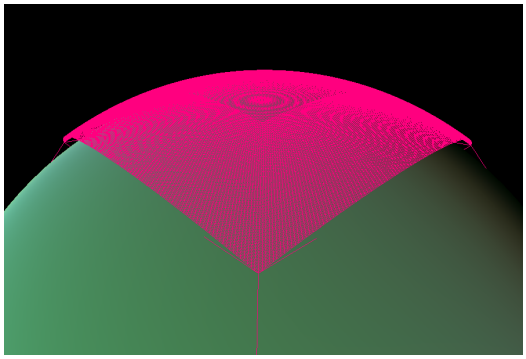
Figure 2: The quad sphere mesh starts off as a simple cube. To get the final shape we tessellate the cube and then map the vertices to a sphere.

There is however some problems with simply using a quad sphere based mesh for all the atmospheric rendering. I found that to achieve the fewest number of artifacts when in atmosphere, the level of tessellation for the atmosphere mesh must be quite high, leading to a great number of vertices and faces that must be drawn and stored in memory. To solve this problem, rather than reusing the same quad sphere mesh, during the mesh generation process I take one side of the cube and modify its design before tessellation begins which can be seen in Figure 3.a. Adding more faces to the side of the cube I can achieve a higher concentration of vertices and faces with a lower overall level of tessellation. Leaving out the other unneeded sides from the tessellation I am left with one highly tessellated side of the cube that effectively covers the view area from the surface of the planet.

By rotating this mesh with the camera around the planet I can achieve a seamless effect and we are in the end left with a much nicer looking result from the atmosphere shader. Another benefit of using this modified mesh is the proportion between the number of vertices that are stored in the mesh and the number of vertices of the mesh that are drawn to the screen. By having all the vertices condensed into one side of the mesh the concentration leads to a many more vertices in the visible area with a much lower level of tessellation. The quad sphere mesh at seven levels of tessellation leads to a total 98,306 vertices and 98,304 faces for the whole mesh. However at seven levels of tessellation for the modified atmosphere mesh we have 82,185 vertices and 81,924 faces, the majority of which are concentrated in the visible side of the mesh. This concentration can be seen in Figure 3.b.



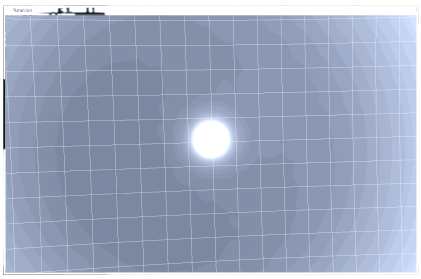
(a) The modified atmosphere mesh design.



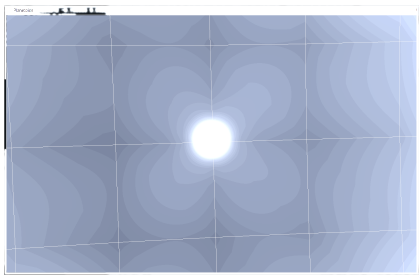
(b) The modified mesh in action.

Figure 3: The modified atmosphere mesh which rotates, centered around the camera.

The difference between the two meshes can be seen in Figure 4. Figure 4.a and 4.b shows high contrast views of the coloring effects caused by the O'Neil algorithm when looking straight up towards the sun. The artifacts caused by low tessellation that are seen in the quad sphere mesh are significantly reduced in the modified atmosphere mesh.



(a) Atmosphere mesh at 7 levels of tesselattion.



(b) Quad sphere mesh at 7 levels of tessellation.

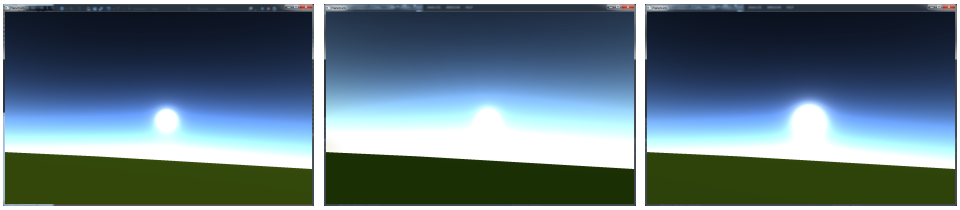
Figure 4: A high contrast view of the differences between the quad mesh and the modified atmosphere mesh.

6.2 Shader Variables

Next I would like to talk about some of the various details of the actual shader implementation regarding the various variables that go into the final result of the algorithm. There are several variables that greatly affect the final outcome, these being:

- Kr
- Km
- eSun
- nSamples
- invWavelength

The values Kr and Km stand for the Rayleigh and Mie scattering constants. These values affect the amount of scattering of each type that occurs within the shader. The greater these values are the greater the effect of the scattering that occurs.



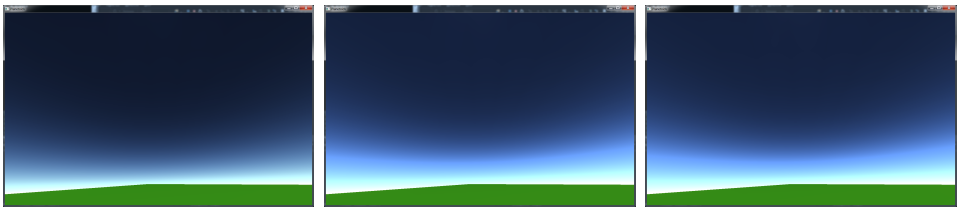
(a) Kr = .0025 Km = .001. (b) Kr = .0050 Km = .001. (c) Kr = .0025 Km = .005.

Figure 5: An example of how changing the Kr and Km values affects the rendering.

eSun is the value representing the energy of the sun that is taken into consideration when calculating the color. The higher the value the brighter the colors we get in the sky.

nSamples is what affects the performance of the algorithm the greatest and also quickly has diminishing returns. This value controls how many

points the scattering calculations are done for. Some examples are provided in Figure 6 that show the effect the number of samples has on the atmosphere.



(a) nSamples = 2 (b) nSamples = 10 (c) nSamples = 50

Figure 6: The difference between 2 and 10 samples is noticeable. However the difference between 10 and 50 samples is not noticeable enough to be worth the performance hit.

The effect the number of samples has on the final outcome is noticeable between the low number of samples and a medium number of samples. However once you get to the high number of samples the effects are indescribable between a medium number of samples and a high number. The main difference between the two is the frame rate that we get when running the application. The diminishing returns that nSamples displays, gives us reason to leave nSamples quite low. Figure 7 shows this relationship.

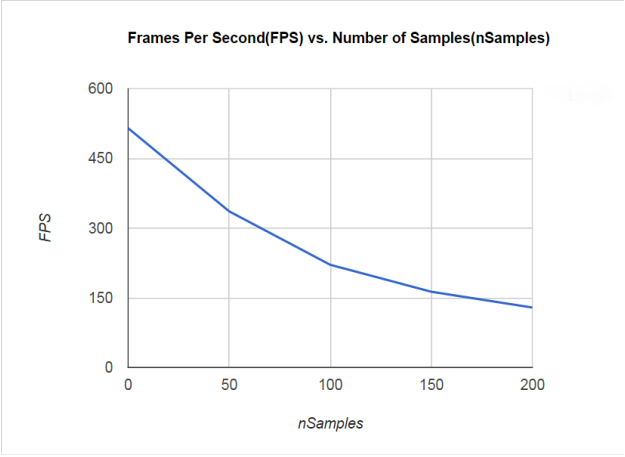
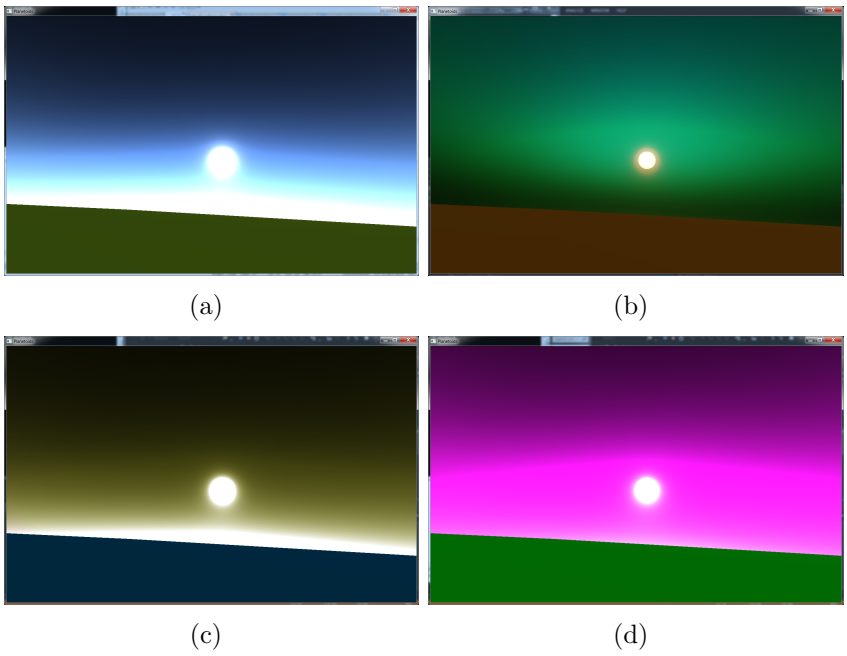


Figure 7: As nSamples increases, frame rate goes down.

150 frames per second may seem like quite a large number, however you must take into consideration the scope of the project. This is only a small portion of what will have to be rendered every second. Eventually the terrain mesh will be much more complex and other factors will come into play that will greatly affect the number of items that will need to be rendered to the screen every frame.

Finally we have the InvWavelength value. The r, g, and b values of this three dimensional vector represents the three wavelengths that are scattered in the algorithm. Figure 8 shows how changing these values affects the color of the atmosphere allowing us to get different looking results by varying these values.



(a) (b)

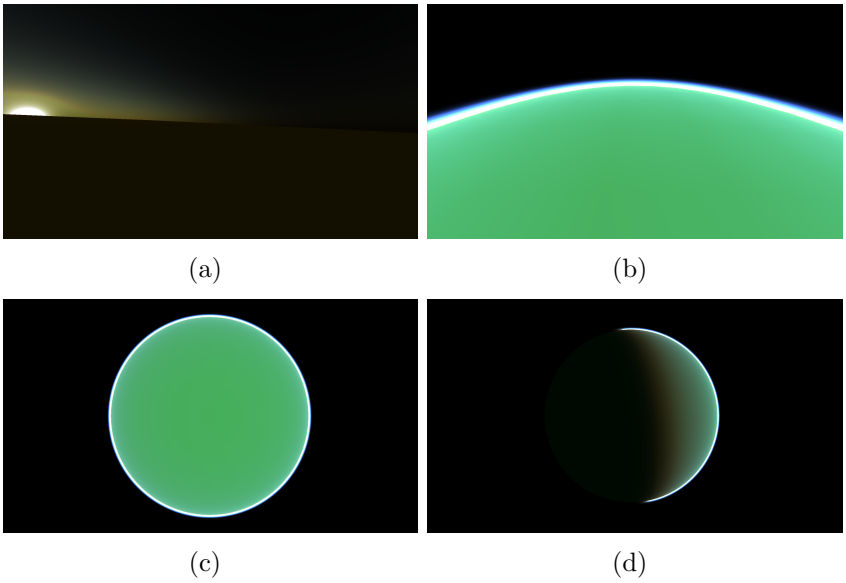
(c) (d)

Figure 8: Many different colors can be displayed by modifying the invWavelength values.

For a more detailed explanation on how these variables fit into the O'Neil shader please refer to the article in GPU Gems 2[1].

7 Results

I am fairly happy with the end results of my efforts at implementing the O'Neil algorithm. The resulting atmosphere looks nice, and renders quickly on my 2.3GHz i7 3610QM and NVIDIA GeForce GTX 670m getting well over the frames per second that could be considered interactive. The atmosphere looks good both from within the atmosphere and from in space looking down.



(a) (b)

(c) (d)

Figure 9: Examples of final results from both in atmosphere and in space.

8 Future Work and Conclusion

In the end I am happy with the O'Neil solution for this project. It gets quick and easy results that still look good while running at interactive speeds. In the future however I would like to take another crack at this problem and perhaps attempt to implement something akin to the Nishita physical based solution. As I learn more about shaders in my work going forward I would like to try my hand at implementing something on my own. To be able to compare and contrast the results of the O'Neil solution and one I implement myself interests me greatly, mostly because only when I can implement it on

my own will I fully understand all the intricacies of writing you own shader from the ground up. There are also a few things I would like to implement further with the solution I currently have. I feel as though there are some parts that could use some work such as fine tuning of the shaders, ironing out a couple rare bugs, and other various things. I would also like to implement some things such as clouds or auroras.

I feel as though I've only breached the tip of the iceberg when it comes to the O'Neil solution and I would like to further understand it better than I currently do. However as it stands I have reached a point where I can consider the original goal of the research to be fulfilled. I have an easy and cheap solution to the atmospheric rendering problem that can be easily modified and translated to a great number of randomly generated planets. This solution allows me to easily modify variables to create a number of interesting and different looking atmospheres that are usable by a planet generator.

Bibliography

[1] O'Neil, Sean. "Chapter 16. Accurate Atmospheric Scattering." *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation.* Upper Saddle River, NJ: Addison-Wesley, 2005. N. pag. Print.

[2] Nishita, Tomoyuki, Takao Sirai, Katsumi Tadamura, and Eihachiro Nakamae. "Display of the Earth Taking into Account Atmospheric Scattering." *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '93* (1993): n. pag. Web.

[3] "Blue Sky." *HyperPhysics*. Web. 05 Aug. 2015. http://hyperphysics.phy-astr.gsu.edu/hbase/atmos/blusky.html.