# Procedural Atmosphere

## Render atmospheric scattering in WebGL

Adam Alsegård*

TNM084 - Procedural Methods for Images
Linköping University - Media Technology
Live demo
Source code

January 20, 2018

**Abstract**

*This report presents the theory and implementation of how an atmospheric light scatter simulation can be rendered in real-time in a web browser with WebGL. Both Rayleigh- and Mie-scattering are simulated in the created scene. A procedural landscape, generated with simplex noise, is also rendered in the scene, as is a GUI where the user can interact with the environment with a number of settings. The report ends with a discussion of possible future improvements for the project.*

## I. Introduction

The heavens. A sky. Our atmosphere. Its display of colours can be enchanting, spectacular and also quite boring from time to time. Nobody would argue that it lacks variation in its display, but where does the colours actually come from? How can they be simulated in computer graphics? Is it possible to do the computations in real-time? Those are the questions that this report hopes to answer.

The aim of this project is to implement an example of atmospheric scattering in WebGL and render it in real-time together with a procedural generated landscape. If time permits shadows, clouds and fog will also be added to the scene. The main inspiration was this *"Badlands"* implementation (see Figure 1) by Rye Terrel [1]. My dream would be to implement something similar but render it in real-time.

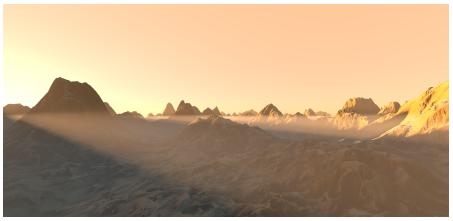

**Figure 1:** *Inspiration for the project.*

## II. Atmospheric scattering

Where does the colours in the sky come from? That is the first question of this project, and it has a quite straightforward answer. The colours of the heavens is due to the scattering of light within the atmosphere. Light from the sun travels through the Earth's atmosphere where it has some probability of scattering, either due to small particles of nitrogen/oxygen (also known as Rayleigh scattering) or due to larger particles known as aerosols such as water vapor, dust or pollution (a.k.a. Mie scattering) [1].

The perceived colour comes primarily from the Rayleigh scattering because of its strong wavelength dependence. Blue light (shorter wavelengths) scatters more quickly than green light, which in turn scatters more heavily than red light (longer wavelengths). The sky appears blue for the most part because the blue lights scatters all over the place, and thus reaches our eyes from all directions. While the sky turns red/orange at sunset/sunrise because as light travels far through the atmosphere, almost all of the blue and much of the green light is scattered away before it reaches our eyes, leaving just the reddish colors.

*adam.alsegard@gmail.com
[1] https://wwwtyro.github.io/badlands/

Mie scattering is also dependent on the ratio of wavelength to particle size, but the aerosols have enough size variation that this wavelength dependence becomes negligible for rendering purposes. Mie scattering instead appears as atmospheric haze and can turn the sky gray or cause the sun to have a large halo around it. Mie scattering can also be used to simulate light scattering from small particles of water and ice in the air to simulate effects like rainbows. That was however not a goal of this project and therefore the theory behind it was not further explored. There are also other scattering effects in the atmosphere but they are negligible for rendering purposes and thus ignored in this report.

## i. How to model scattering

The mentioned scattering effects have been implemented multiple times over by dozens of people. However, most implementations seems to refer to the same handful of papers. First we have Nishita et al. [2] whom developed the basic equations for rendering atmospheric scattering. Then Hoffman and Preetham [3] developed methods of calculating the equations in real time by assuming that the camera is always on or very close to the ground, and thus assuming that the atmosphere has a constant density at all altitudes, which simplifies the equations tremendously. Finally O'Neil wrote a *GPU Gems* chapter [4] on how to do it even better and faster on the GPU with modern shaders. This section will only cover some of the most basic parts of that algorithm.

First of, when modeling the scattering process, one has to divide the scattering into two sub-processes; accumulation and extinction. Extinction, or out-scattering, determines how much light is lost due to absorption and scattering while accumulation, or in-scattering, determines how much light is added to our path due to scattering of other light paths. When in-scatter is accumulated, we are only interested in light which is specifically scattered towards the observer. This is given by the "phase function" which determines the distribution of scattered light.

According to O'Neil [4] the phase function is as follows

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

where $\theta$ is the angle between the view direction and the light direction and $g$ is a constant that affects the symmetry of the scattering. Because Rayleigh scattering can be approximated by setting this constant to 0 it is also known as the *Mie scattering direction* in some implementations.

To calculate the extinction one uses the optical depth, which is the average atmospheric density across a ray from point A to point B multiplied by the distance between the points. My implementation follow O'Neil's suggestion to use the density 25% the way up from the ground as the average density. The atmosphere does not really have a fixed top but for practical purposes we assume there is one and scale it to 1 (with 0 being sea level). The top of the atmosphere is also where the atmosphere will be rendered, on the inside of a sphere.

The cited papers also have complex equations for in- and out-scattering with nested integrals that this report will not go into detail or trying to explain, as said papers do it much better.

# III. Implementation

There are multiple other implementations of sky scattering that can be found. Some examples of implementations are [5] by *Florian Boesch*, [1] by *Mike Lodato*, [6] by *Scratch a pixel* and [7] by *Patapom*. My implementation has a foundation that is a mixture of several other implementations. I found tips and trick here and there and from that I sculpted my own implementation of the equations. Because many developers are quite bad at documenting their code this lead me to spending a lot of time trying to figure out what different "magic constant" actually meant.

The scene is created and rendered with *THREE.js*, but as it is very bulky to work with shaders in three (the shaders have to be written as a continuous string) I used *glslify* to handle shaders. *Node.js* is used for installation, *browserify* is used to bundle the depen-

dencies together and *budo* is used to start a live server that listens to updates during development.

## i. Sky shaders

The atmosphere is created as a sphere buffer geometry with the material from the shaders attached to the inside, so the sphere surrounds the scene (see figure 2).

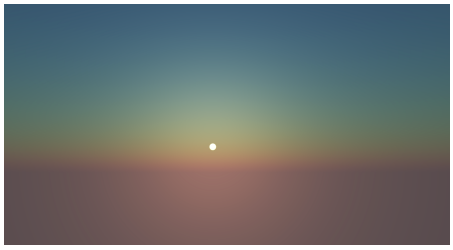

**Figure 2:** *Atmospheric scattering during sunrise.*

High precision floats was used as much of the colour variation would be lost otherwise. HDR quality is still not achievable in the browser, but to improve the colours a bit a tone-mapping from *FilmicWorlds* was used [2].

To hide the sun and light when it is supposed to be on the other side of the globe a horizon extinction was added. The sun itself is rendered as a disc with a colour display of its own.

## ii. Generate a landscape

A simple procedural landscape was also added to the scene as a plane buffer geometry. The height of the landscape is generated with simplex noise in a set of shaders to create a height map. The generated texture is then processed by another set of shaders to produce a normal map. Both of which is sent to the "real" landscape shader together with a pair of color textures to produce the final ground (see Figure 3).

In OpenGL these maps would be stored as frame buffer objects (FBOs) but the way I did it in WebGL was to create the maps as *WebGLRenderTargets* and then attach the map as a material on a quad and let an ortographic camera render the shader output onto the quad, which would then be stored in the material. Later this material could be sent to another pair of shaders and be sampled.

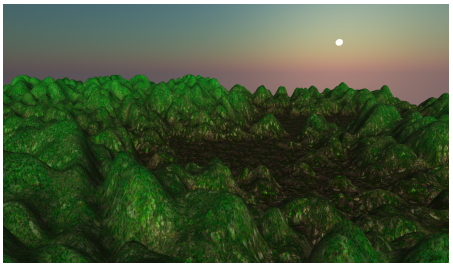

**Figure 3:** *Procedural landscape.*

The landscape has two different textures. One quite murky for the lower ground and one lighter for the higher parts. The displacement height decided which texture to use where. These textures also have a normal map and a specular map each that are used in the shading. The shading was then a simple Blinn-Phong model.

I wanted to use the tangent and bi-tangent of the landscape together with the normal map to create a more visually pleasing result, but I had one persistent error that I didn't manage to solve it before time ran out.

## iii. GUI

A graphical user interface was created with *dat.gui* where the user can change a number of parameters that are sent to the shaders as uniform variables. This makes the program interactive as the settings takes effect immediately during runtime.

For the sky calculations the user can control the turbidity (which is the cloudiness/haziness of a fluid), luminance, Rayleigh scattering coefficient, Mie scattering coefficient and Mie scattering direction (a.k.a. the $g$ constant).

There is also settings for the placement of the sun. The user can choose to set the spherical angles manually or if the sun should rise automatically so the user can watch the sunrise for example. There is also a "dummy sun" that can be switched on if the user has lost track of where the sun is.

Likewise can the landscape be turned on

[2] http://filmicworlds.com/blog/filmic-tonemapping-operators/

and off, as well as a *GridHelper* that was added to help with navigation initially. There are also a couple of parameters for the landscape generation in the settings, one of which will animate the landscape. This is not a very realistic feature, as it would be better suited for water simulations or another fluid but I thought it was fun. The final GUI can be seen in Figure 4.

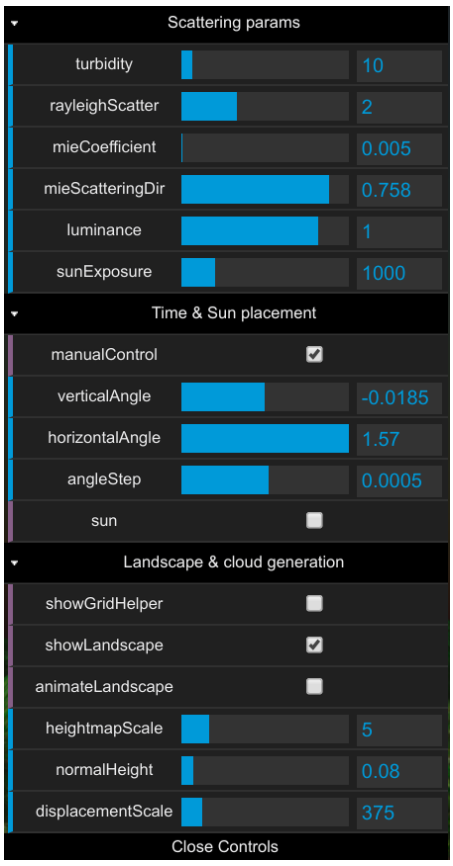


**Figure 4:** *GUI with interactive settings.*

## IV. Results

Most of the results are already shown in Section III but here are a few more screen-shots from the program. Figure 5 shows a composition of the final scene whereas Figure 6 shows an example of a sunrise.

## V. Discussion

Light scattering is a very fascinating subject. Unfortunately for me it is also very time-consuming to understand. One can easily see how people can research light scattering for years and write books and technical papers about all the equations. This project on the other hand only brushed the surface of the subject.

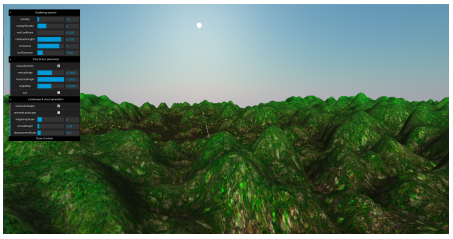

**Figure 5:** *Final scene.*

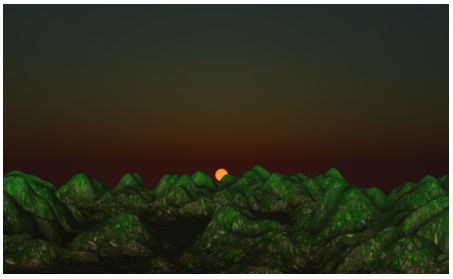

**Figure 6:** *Sunrise over the procedural landscape.*

As the scattering was the main objective I spent most time on the atmosphere and only added the landscape late in the project. Which in turn lead to that there are still a few problems with the landscape. The tangents are not correct as mentioned which leads to problems with the normals. Hopefully I can fix this in the near future but to be able to submit before the deadline I had to skip it for now.

However, an atmosphere was produced and also a procedural landscape, which was the goal from the start. It's not exactly what I hoped for but since the majority of the available time was put into trying to understand the scattering implementation not much was left for the landscape. Unfortunately that also meant that there was not enough time to implement procedural clouds, fog or shadows as initially hoped. Not in the scope of this course at least.

### i. Future development

There is still a long way to go before I reach the beauty of the "Badlands" that I had as

inspiration. For that I need a couple more shaders to deal with procedural clouds, fog and shadow volumes. Then one could explore equations for such things as participating media, ambient occlusion and more advanced landscape generation. It would for example be interesting to be able to switch type of noise in the settings and maybe even set the sample parameters during runtime.

Of course with every new step it will be more and more difficult to keep rendering in real-time. It would however be very interesting to see how much you can cram into the project before it's no longer possible to render at a reasonable high fps, while still being able to navigate and look around in the scene!

## References

[1] Mike Lodato. Drawing skies in WebGL. `http://zvxryb.github.io/blog/2015/07/10/atmosphere/`. Online; Accessed 15 January 2018; Published 10 July 2015.

[2] 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 '93, pages 175–182, New York, NY, USA, 1993. ACM.

[3] Naty Hoffman and Arcot J Preetham. Rendering outdoor light scattering in real time. *ATI Technologies Inc.*, 2002.

[4] Sean O'Neil. Chapter 16: Accurate atmospheric scattering. *GPU Gems 2*, 2005.

[5] Florian Boesch. Advanced WebGL - Part 2: Sky rendering. `http://codeflow.org/entries/2011/apr/13/advanced-webgl-part-2-sky-rendering/`. Online; Accessed 15 January 2018; Published 13 April 2011.

[6] Scratch a pixel. Simulating the Colors of the Sky. `https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/simulating-sky/simulating-colors-of-the-sky`. Online; Accessed 15 January 2018; Published: Not accessible, but after 2014.

[7] Benoît Mayaux (Patapom). Real-Time Volumetric Rendering. `https://patapom.com/topics/Revision2013/Revision%202013%20-%20Real-time%20Volumetric%20Rendering%20Course%20Notes.pdf`. Online; Accessed 15 January 2018; Published April 2013.