Sky Rendering: Atmosphere and Cloud Rendering

HE Qihao, Li PingJiang
Group 04

1 Introduction

Rendering of the sky scene is a very important aspect of large-scale game aiming for photo-realistic effect, for example, volumetric cloud effect in Horizon: Zero Dawn [1] and atmosphere rendering technique deployed in the unreal engine [2]. So we are trying to devote our project to this topic, as we have found a great project based on WebGL2 from Eric Bruneton's "Precomputed Atmospheric Scattering" [3] on atmosphere rendering, what we are trying to do is to focus on adding cloud rendering to his project.

2 Methodology

In our project, we only consider two main components of a sky scene, one is the atmosphere rendered using Eric Bruneton's method, another is the cloud rendered by ourselves. We will elaborate on our understanding of Bruneton's method in subsection 2.1, and our own implementation in subsection 2.2.

2.1 Atmosphere Rendering

Overall, it uses ray tracing to render the atmosphere, that is it starts from where the camera is located and towards the viewing direction. Whenever it intersects with the atmosphere (The surface of the sphere), the fragment shader will computer the intersection of the ray with that particular position, and eventually output that pixel.

The scattering effect through out the atmosphere needs to be considered. For computing this effect, one commonly used method is to build a Single Scattering Look Up Table (LUT) based on position of camera, the position of the sun, the viewing direction and the viewing height [4]. When performing multi-scattering effect, the LUT needs to be computed for several iterations. One big drawback is that computing several iterations of LUT is expensive and slow, and whenever the outer environment changes, like the position of sun moves, the LUT has to be re-computed.

In order to make the computation more efficient, we learnt that geometric series can be used to approximate the visual effect. It assumes that the scattering effect of the current iteration is the effect of previous iteration multiplied by a constant ratio. Then we only need the first iteration of LUT, and the rest is simply the summation of a geometric series, which makes the computation much more efficient [5].

Besides simply rendering the atmosphere, it also considers the effect of sun and sky. For the sun, we need to consider its visibility, which can be achieved by calculating whether the sun ray can be reached by verifying whether there is intersecting point between the view point and the sun. Figure 1 left is an example, the small white circle represents the sun.

For the outer sky, we mainly consider its ambient light, and at the intersection points between the margin of spherical planet and the outer sky, it will multiplied by some view ambient factors to make the transition looks smoothly [6]. Figure 1 right is an example for the sky.

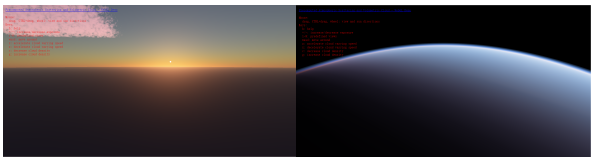

Figure 1: Left is the sun, right is the outer sky

2.2 Cloud Rendering

In this section, we will follow the sequence we are taking in shaping our cloud from start to end, like described in our presentation, altogether there are three steps: 2.2.1 on basic volume rendering using ray marching, 2.2.2 on the noise function approach we took to procedurally generate moving cloud, and some final improvements of the density noise value we are sampling inside the box. Two main works we are referring to to render our own cloud are Wang YiYun's cloud-generation project [7] on volumetric cloud ray-marching algorithm and Daniel Scherzer's SHADER project [8] on fractal noise generation algorithm. All implementations on the cloud rendering by us are in the demoFS.glsl file ranging from line 89 line 300.

2.2.1 Ray Marching a Cloud

Ray marching is a fairly efficient rendering technique commonly used for various participating media including smoke and cloud [9], and we are using this approach exactly for our cloud contained in a box. Such a box is defined in the 3d space through its dimensions and 3d position with 2 vectors without use of vertices from vertex shader, since we are rendering a full-screen quad.

By defining a RayIntersectBox function, we can detect the intersection between camera and light rays with the box. Then we shoot a camera ray that marches inside the box with RenderVolume function, and along the path of the camera, we sample at an interval light received at points inside the volume, which is a value returned by lightMarch function, where we shoot a light ray that marches towards sun direction at these sampling points. As both rays marches through the volume, they got attenuated as the volume transmittance is decreasing. For this stage, we assume homogeneous density distribution inside the box, and transmittance is thus attenuated at the same amount given a constant travelling distance inside the box, which is shown in Figure 2.

2.2.2 Value Noise Function to Shape Cloud

Since the box looks nothing like a cloud, we are making the density inside the box following a distribution generated by summing up multiple octaves of noise value function results at the point

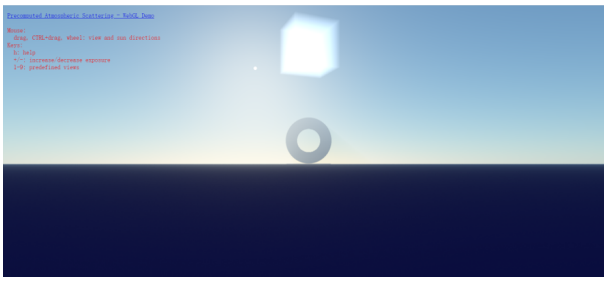

Figure 2: Ray Marching a simple homogeneous box

of sampling with its 3d position coordinate value p following the chain of functions:

$$hash- > noise- > fBm- > densityFunc$$

That means at each octave, the amplitude of the noise function result is decreased by half, and the noise function parameter, the positional value p is doubled, and the results summed up in total 4 octaves, returned to densityFunc as the density value at the point p.

But that is not the end, cloud in this way is not only still but still has a very sharp shape. It is straight-forward to make it moving by introducing a small variance to the positional value p by using the system time we pass in the shader, which can be controlled by the user in run-time. For the second problem, it is also not very hard, we simply have to erode the cloud further, in this part, we refer to Yiyun's work [7] on shape erosion based on height percentage quadratic equation:

$$Erosion = SAT(h * c1) * SAT((1 - h) * c2)$$

With SAT being the saturation function that clamps value to interval [0.0,1.0], h the heightPercentage of a point in the box given by its distance to the box bottom divided by the box length along the y dimension, c1, c2 two constants randomly picked, and the returned Erosion being a scalar between [0.0, 1.0] that applies on the density value obtained by the fBm (fractal Brownian motion) function earlier. Also, it is a choice made by us to skip points with density value lower than a threshold, also adjustable during run-time by the user, so that we erodes the cloud shape further. It turns out the result is better than the one without further erosion as shown in Figure 3. And a Demo Video also available for this result.

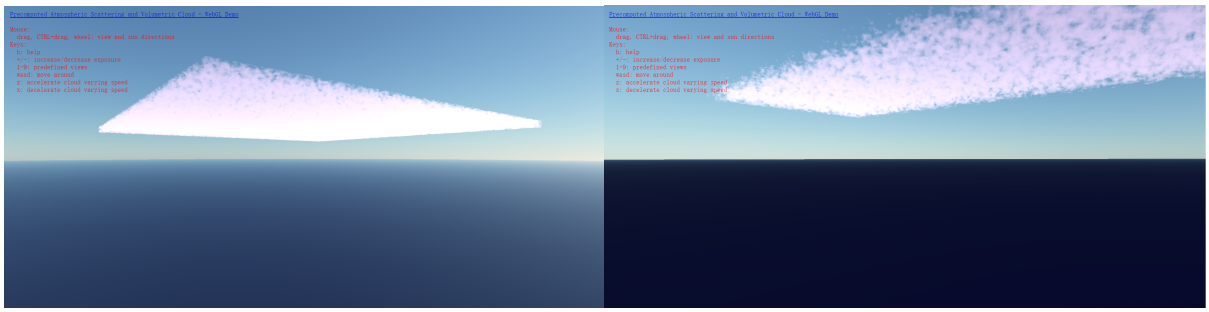

Figure 3: Left is before the Erosion, right is after the Erosion

3 Conclusion and Future Work

In conclusion, we have investigated Bruneton's work in atmospheric rendering, and devoted our own efforts on cloud rendering. And we anticipate many potential improvements to our cloud. Firstly, we may sample noise from 3d textures instead of generating it on-the-fly to improve both performance and cloud shape and detail, meaning that artists and users can have editable clouds by changing the textures directly. Secondly, we could further combine the textures of Bruneton's Atmosphere LUTs with our cloud, though this will be possible only after we optimized our cloud. So the sky rendering is really full of amazing stuff, and we will continue working on this path to get something better!

References

[1] Andrew Schneider. The real-time volumetric cloudscapes of horizon zero dawn.

[2] Sébastien Hillaire. A Scalable and Production Ready Sky and Atmosphere Rendering Technique. *Computer Graphics Forum*, 39(4):13–22, 2020.

[3] Eric Bruneton and Fabrice Neyret. Precomputed Atmospheric Scattering. *Computer Graphics Forum*, 27(4):1079–1086, June 2008.

[4] Tomasz Gałaj and Adam Wojciechowski. A study on numerical integration methods for rendering atmospheric scattering phenomenon. *Open Physics*, 17(1):241–249, 2019.

[5] Tomoyuki Nishita, Yoshinori Dobashi, and Eihachiro Nakamae. Display of clouds taking into account multiple anisotropic scattering and sky light. *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques - SIGGRAPH '96*, 1996.

[6] T. Walker, S.-C. Xue, and G. W. Barton. Numerical Determination of Radiative View Factors Using Ray Tracing. *Journal of Heat Transfer*, 132(7), 04 2010. 072702.

[7] Yiyun Wang. Cloud generation. https://github.com/wangyiyun/Cloud-Generation, 2020.

[8] Daniel Scherzer. Shader. https://github.com/danielscherzer/SHADER, 2022.

[9] Kun Zhou, Zhong Ren, Stephen Lin, Hujun Bao, Baining Guo, and Heung-Yeung Shum. Real-time smoke rendering using compensated ray marching. *ACM Transactions on Graphics*, 27(3):1–12, aug 2008.