Review of Paper [Yus14]

High-Performance Rendering of Realistic Cumulus Clouds Using Pre-computed Lighting

Huanxiang Wang
14333168

November 20, 2014

Contents

List of Tables **ii**

List of Figures **iii**

Chapter 1 Introduction **1**

1.1 Abstract 1

Chapter 2 Background **3**

2.1 Cumulus Clouds 3

2.1.1 What is cumulus clouds? 3

2.1.2 Why does the paper concentrate on rendering cumulus clouds? . . . 4

2.2 Existing cloud rendering techniques 4

2.2.1 Particle Systems (Billboards) 4

2.2.2 Ray Casting-Based Rendering (Ray Marching) 5

2.2.3 Rasterization-based Rendering (Volume Slicing) 6

2.3 Mathematical Background 6

Chapter 3 Algorithm and Implementation **9**

3.1 Pre-Computing Optical Depth 9

3.1.1 The Design 9

3.1.2 The Implementation 10

Appendix A Abbreviations **13**

Bibliography **14**

List of Tables

List of Figures

1.1 Overview 1

2.1 Single cumulus cloud 3
2.2 Billboards Clouds 4
2.3 Ray Marching Clouds 5
2.4 Volume Rendering Clouds 6
2.5 The scattering behaviour of lights through a cloud 7
2.6 Light scattering in the cloud 7
2.7 The phase function $P(\theta)$ 8

3.1 Four angels to describe a ray 9
3.2 Numerically integrate the optical depth of a ray 10
3.3 A typical noise texture 10

Chapter 1

Introduction

1.1 Abstract

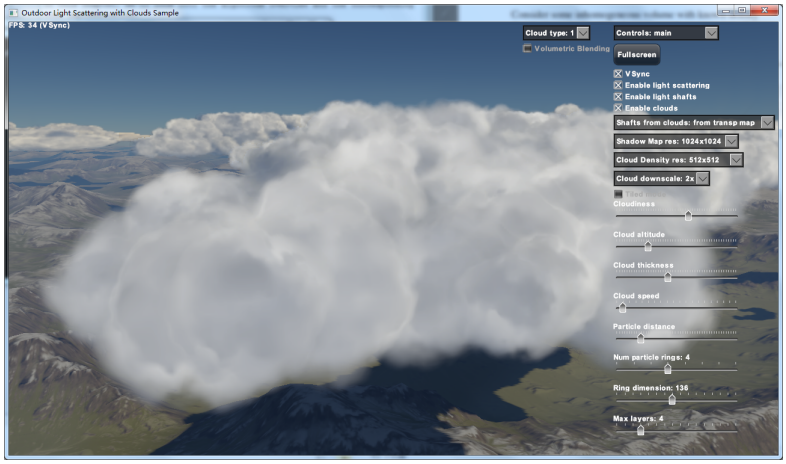

Figure 1.1: Overview

In the paper, the author introduced a new method for rendering realistic cumulus clouds in real time. The basic steps include:

1. Pre-computing optical depth.
2. Pre-computing scattering.
3. Generating a real-time lighting model from the above two steps.
4. Volume-aware blending the particles.
5. Controlling the level of details.

He also explained the mathematical background to solve the problem. The paper is interesting, but hard to understand for a person not in the field. I'm writing this review to make the author's idea easier to be understood. I will also supplement to some uncertain explanation of the author.

Chapter 2

Background

2.1 Cumulus Clouds

In the Introduction section of the paper, the author introduced their method and their rendering target briefly. I'm writing this section to supplement his introduction.

2.1.1 What is cumulus clouds?

Figure 2.1: Single cumulus cloud

Cumulus clouds is a type of clouds which is fairly close to the ground with an observable volume and visible and clear edges. Comulus clouds are often been noticed as "puffy" or "cotton" in the appearance, and they also generally have flat bases. As a low-stage kind of clouds, the general altitude of the cumulus clouds are less than meters, unless they are in the more vertical cumulus congestus form. As for the appearance, cumulus clouds usually appear by themselves, in certain forms, such as lines and clusters.

2.1.2 Why does the paper concentrate on rendering cumulus clouds?

Among the numerous kinds of clouds in the nature environment, cumulus clouds have the most pronounced volumetric feature, making the volumetric rendering more practical. In the final step of the proposed method from the paper, a volume-aware blending is performed. If we use other kind of clouds, such as stratus clouds or cirrus clouds, the volumetric feature can be hardly found, which is not able to be rendered as physically based clouds. This is because we will need a volume to calculate the light occlusion later on.

2.2 Existing cloud rendering techniques

The author of the paper talked about the related work that has been down and from his research we know that off-line rendering can be quite accurate about simulating the cloud, but it could take minutes to hours to draw a single frame. We'll be looking into the existing real time rendering methods in detail in this section.

2.2.1 Particle Systems (Billboards)

Figure 2.2: Billboards Clouds

Particles in video game industry are referenced as small units with independent rotations, and they are usually rendered by using a textured quad which represents the projection onto the view plane. They are rendered in back-to-front order, and also the blending for semi-transparent volumes are applied.
By rendering clouds with billboards as particle system, we treat the clouds as 3D objects,

and they face the camera all the time. We also generate shafts of light by making concentric semi-transparent containers. Here is a typical scenario for a flight game: the player controls an aircraft shuttling in and out of clouds. When the aircraft moves into a cloud, the imposter of that cloud will be split into 2 pieces, one piece in the front of the aircraft, and the other piece behind the aircraft.
This method has certain advantages, such as fast, efficient, and the programmer can easily manipulate the shapes and the locations of the billboards. However, as the clouds are represented in a quads facing the camera all the time, the scene will get unrealistic sometimes, for example, when an aircraft flies above the clouds, we will still see the clouds facing the camera as if the we are on the ground. Also the lighting on the billboard is usually precomputed and the clouds are static, which means that the clouds will not be able to adjust its appearance according to the environment's changes.

2.2.2 Ray Casting-Based Rendering (Ray Marching)

Figure 2.3: Ray Marching Clouds

By rendering clouds in the ray marching method, the typical way is to cast multiple rays into the scene and accumulate the densities of the volume within certain interval distances. We then represent the cloud density with 3D noise images. In order to calculate the illumination values, we usually apply a volume lighting model in real time or we can retrieve the illumination data from a pre-computed lighting structure, such as a look-up table.
This method can result in a fairly realistic output. However, it's not easy for a programmer to control the shape and location of the clouds. Also, in order to do aliasing, the ray marching steps will be tedious. As for the rendering result, the lightings is usually limited to single scattering, which is more or less not so realistic.

2.2.3 Rasterization-based Rendering (Volume Slicing)

Figure 2.4: Volume Rendering Clouds

In terms of rasterization-based rendering, volume slicing is the most realistic method for rendering clouds.
Volume slicing is quite straightforward, and it is used for rendering regular grids. The slices of the volume are usually aligned to XYZ axes and they are rendered in front-to-back or back-to-front order. Finally, we apply the view transformations and blending to the rendering pipeline. As we can see, volume slicing is not strictly based on rasterization methods, most algorithms of this field rely on the highly optimized texturing capabilities of the GPU.
By using this method, we can have a convincing light result, since it enables the multiple forward light scattering using slices. However, like ray marching method, it's difficult for a programmer to control the shape and location of the clouds. Also, in order to get a more realistic result, the volume will be sliced into tons of planes, which takes a lot of memory and execution time.

2.3 Mathematical Background

The author discussed the concepts of calculating the light transport in a certain medium and the mathematical solution to the concepts. As we know, the light is able to transport in a number of materials, such as air, fluid, and solid medium. Different materials of the medium will result in different amount of scattering, absorption and emission. We note the scattering coefficient as β_{Sc}, the absorption coefficient as β_{Ab}, the emission coefficient as β_{Em}. Now we'll introduce a new coefficient called the extinction coefficient β_{Ex}, which is the measurement of how strongly the light is absorbed in a medium. For any type of

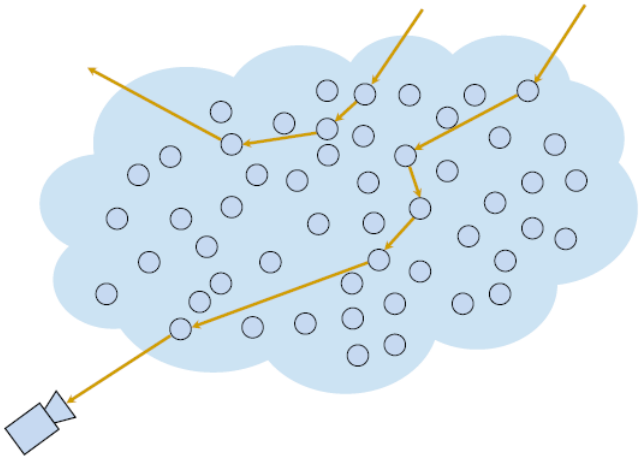

Figure 2.5: The scattering behaviour of lights through a cloud

medium, we have:

$$\beta_{Ex} = \beta_{Sc} + \beta_{Ab} \tag{2.1}$$

The author implied that cloud are purely scattering medium, which means $\beta_{Ab} = \beta_{Em} = 0$ Hence, the extinction coefficient for the cloud is the scattering coefficient, which indicates $\beta_{Ex} = \beta_{Sc}$. Since we only have one coefficient to worry about, we'll note the extinction coefficient simply as β in the rest of the review, and it will also be referred to as the scattering coefficient. A schematic can be found in the figure below.

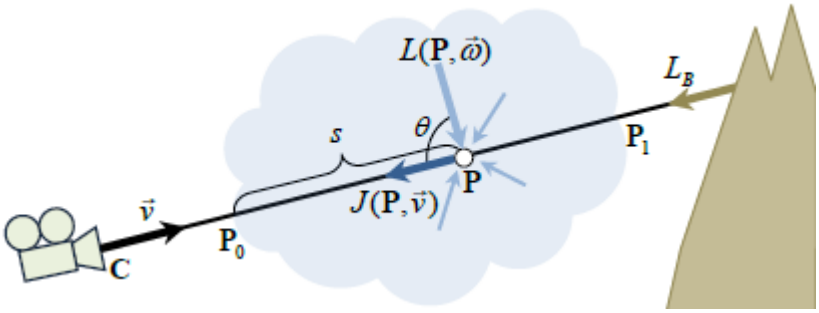

Figure 2.6: Light scattering in the cloud

Now let's say, we have a point **A** and point **B**, the normal direction from point **A** to **B** is $\overrightarrow{r} = \frac{B-A}{|B-A|}$. We note the current integration point, which has one step size from point **A** to **B**, as **P**. The author then indicated that the optical depth $\tau(A, B)$ between points A and B is the integral of the extinction coefficient on the path:

$$\tau(A, B) = \int_A^B \beta(P) \cdot ds \tag{2.2}$$

In addition, the light gets attenuated while propagating through the cloud, so let's assume that the intensity of the light is reduced by a factor of $e^{-\tau}$.
In Figure 2.6, the camera is in point **C**, the viewing direction is $\overrightarrow{v}$. The author proposed that the final light L_{In} getting into the camera can be calculated in the following equation:

$$L_{In}(C, \overrightarrow{v}) = \int_{P_0}^{P_1} e^{-\tau(P,P_0)} \cdot J(P, \overrightarrow{v}) \cdot \beta(P) \cdot ds \tag{2.3}$$

As we can see in Figure 2.6, P_0 is the entry point of the view ray into the cloud, and P_1 is the exit point. $J(P, \overrightarrow{v})$ is the total radiance of light at point **P** towards the camera:

$$J(P, \overrightarrow{v}) = \int_{\Omega} L_I n(P, \overrightarrow{\omega}) \cdot P(\theta) \cdot d\omega \tag{2.4}$$

Here, the author introduced $P(\theta)$ as the phase function describing the probability of a photon being scattered from the incident direction to the outgoing direction $\overrightarrow{v}$, where θ is the angle between two.

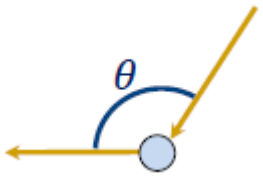

Figure 2.7: The phase function $P(\theta)$

The author mentioned that the equation 3 is hard to solve, and he used **Cornette-Shanks** function to approximate the phase function:

$$P(\theta) \approx \frac{1}{4\pi} \cdot \frac{3(1-g^2)}{2(2+g^2)} \cdot \frac{1+\cos^2\theta}{(1+g^2-2g\cos\theta)^{\frac{3}{2}}} \tag{2.5}$$

Chapter 3

Algorithm and Implementation

In this chapter, we'll look into the algorithm overview and the corresponding code implementation.

3.1 Pre-Computing Optical Depth

The first step is to pre-compute the optical depth using Equation 2.2: $\tau(A, B) = \int_A^B \beta(P) \cdot ds$ for each ray to the boundary of a certain particle. The author assumed that the camera will always stay outside the volume. Then, the goal of this step is to calculate the optical depth integral for all possible camera positions and orientations.

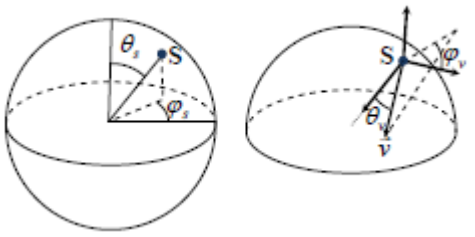

Figure 3.1: Four angels to describe a ray

3.1.1 The Design

To describe a ray, we need 4 angles, 2 angles describing the start point on the particle sphere, and 2 angles describing the view direction. Because of the 4 parameters, a 4D look-up table is necessary. As shown in the figure above, $\varphi_S \in [0, 2\pi]$ and $\theta_S \in [0, \pi]$ are used to specify the start point of the ray. While $\varphi_v \in [0, 2\pi]$ and $\theta_v \in [0, \frac{\pi}{2}]$ are used to specify the direction of the ray.

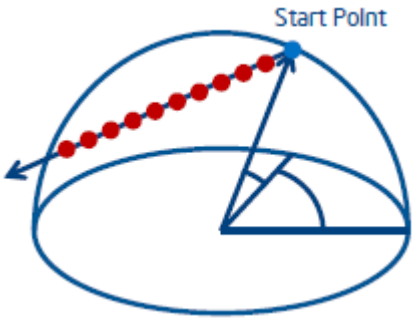

Figure 3.2: Numerically integrate the optical depth of a ray

Then we use Equation 2.2: $\tau(A, B) = \int_A^B \beta(P) \cdot ds$ to accumulate the optical depth of a ray. Now, here is the interesting part of how to retrieve the extinction coefficient $\beta(P)$. The author used 4D look-tables which contains several 3D noises to get the extinction coefficient. The figure below shows one typical noise texture.

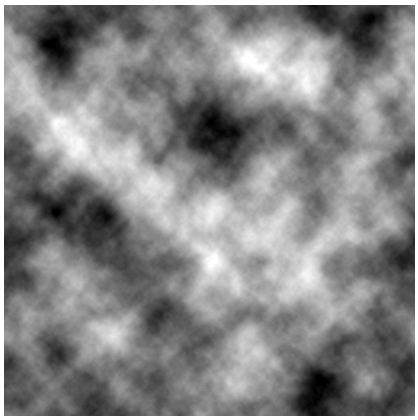

Figure 3.3: A typical noise texture

3.1.2 The Implementation

After carefully explaining the design of the first step, let's have a look at how the actual related function in the shader looks like. Please note that the rendering API is **DirectX**, hence the shader is written in **HLSL**.

```
// This shader computes level 0 of the maximum density mip map
float2 PrecomputeOpticalDepthPS(SScreenSizeQuadVSOutput In) : SV_Target
{
    float3 f3NormalizedStartPos, f3RayDir;
    OpticalDepthLUTCoordsToWorldParams( float4(ProjToUV(In.m_f2PosPS),
        g_GlobalCloudAttribs.f4Parameter.xy), f3NormalizedStartPos, f3RayDir );

    // Intersect view ray with the unit sphere:
    float2 f2RayIsecs;
    // f3NormalizedStartPos  is located exactly on the surface; slightly move start pos
        inside the sphere
    // to avoid precision issues
    GetRaySphereIntersection(f3NormalizedStartPos + f3RayDir*1e-4, f3RayDir, 0, 1.f,
        f2RayIsecs);

    if( f2RayIsecs.x > f2RayIsecs.y )
        return 0;

    float3 f3EndPos = f3NormalizedStartPos + f3RayDir * f2RayIsecs.y;
    float fNumSteps = NUM_INTEGRATION_STEPS;
    float3 f3Step = (f3EndPos - f3NormalizedStartPos) / fNumSteps;
    float fTotalDensity = 0;
    float fDistToFirstMatter = -1;
    for(float fStepNum=0.5; fStepNum < fNumSteps; ++fStepNum)
    {
        float3 f3CurrPos = f3NormalizedStartPos + f3Step * fStepNum;

        float fDistToCenter = length(f3CurrPos);
        float fMetabolDensity = GetMetabolDensity(fDistToCenter);
        float fDensity = 1;
#if DENSITY_GENERATION_METHOD == 0
        fDensity = saturate( 1.0*saturate(fMetabolDensity) + 1*pow(fMetabolDensity,0.5)*(
            GetRandomDensity(f3CurrPos, 0.15, 4, 0.7 )) );
#elif DENSITY_GENERATION_METHOD == 1
        fDensity = 1.0*saturate(fMetabolDensity) + 1.0*pow(fMetabolDensity,0.5)*(
            GetRandomDensity(f3CurrPos, 0.1,4,0.8)) > 0.1 ? 1 : 0;
#elif DENSITY_GENERATION_METHOD == 2
        fDensity = GetPyroSphereDensity(f3CurrPos);
#endif

        if( fDensity > 0.05 && fDistToFirstMatter < 0 )
            fDistToFirstMatter = fStepNum / fNumSteps;

        fTotalDensity += fDensity;
    }
    if( fDistToFirstMatter < 0 ) fDistToFirstMatter = 1;
    return float2(fTotalDensity / fNumSteps, fDistToFirstMatter);
}
```

The integration function is **PrecomputeOpticalDepthPS**. Firstly, it decomposes the given projection position and the 4 angle parameters from inhomogeneous coordinates into 2 vec3 positions, the first one **f3NormalizedStartPos** describes the start position of the ray, the second one **f3RayDir** describes the ray direction. Then the intersecting view

ray inside the unit sphere is calculated. Since the start position is just on the surface of the sphere, in order to avoid precision issues, the author moved the start position slightly into the sphere by the amount of **f3RayDir*1e-4**. Then the integration is done inside the **for loop**. **DENSITY_GENERATION_METHOD** is defined as: 0 represents radial fall-off + 3D noise, 1 represents 3D noise + thresholding, 2 represents pyroclastic style. **fDensity** is the current density of this step. Finally, the total density **fTotalDensity** for a ray is accumulated.

3.2 Pre-Computing Scattering

Appendix A

Abbreviations

Short Term	**Expanded Term**
DNS	Domain Name System
DHCP	Dynamic Host Configuration Protocol
...	...

Bibliography

[1] Cumulus cloud
http://en.wikipedia.org/wiki/Cumulus_cloud

[2] Billboarding
http://www.flipcode.com/archives/Billboarding-Excerpt_From_iReal-Time_Renderingi_2E.shtml

[3] Raymarching clouds Shader
https://www.shadertoy.com/view/XslGRr

[4] Dobashi et al (2000) A Simple, Efficient Method for Realistic Animation of Clouds

[5] Hufnagel R., Held M. (2012) A survey of cloud lighting and rendering techniques Journal of WSCG 20, 3 (2012), 205216.

[6] Mark J. Harris (2002) Real-Time Cloud Rendering for Games

[7] Joe Kniss, Simon Premoze, Charles Hansen, Peter Shirley, Allen McPherson A Model for Volume Lighting and Modeling