# Review of Paper [Yus14]

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

**Huanxiang Wang**
14333168

November 21, 2014

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Abstract

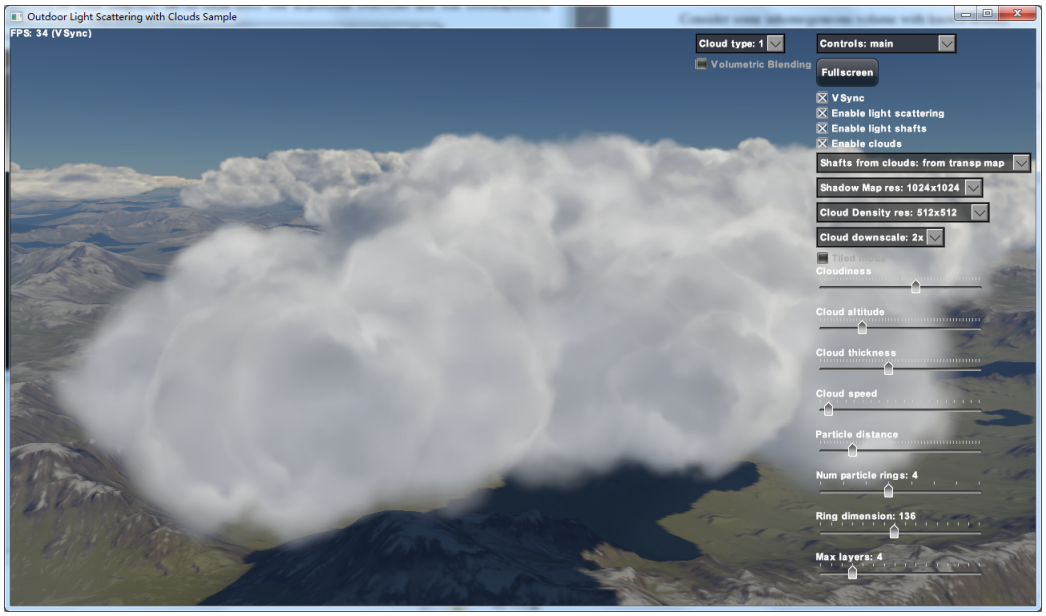


Figure 1.1: Overview

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. I'm writing this review to make the author's ideas easier to be understood. And I will also supplement to some uncertain explanation from the author with my personal thoughts. Finally, I'll include a conclusion section in the end of each chapter to elaborate my opinions to the author's approach.

# 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)

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.

Figure 2.4: Volume Rendering Clouds

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

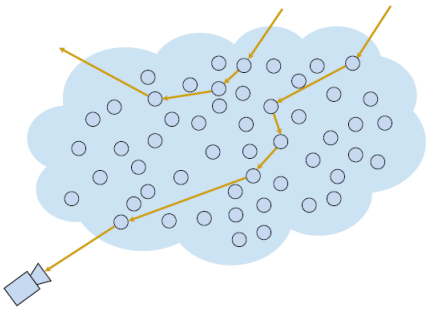

Figure 2.5: The scattering behaviour of lights through a cloud

in different amount of scattering, absorption and emission. We denote the scattering coefficient by $\beta_{Sc}$, the absorption coefficient by $\beta_{Ab}$, the emission coefficient by $\beta_{Em}$. Now we'll introduce a new coefficient called the extinction coefficient $\beta_{Ex}$, which is the measurement of how strongly

the light is absorbed in a medium. For any type of 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 denote the extinction coefficient simply by $\beta$ 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.

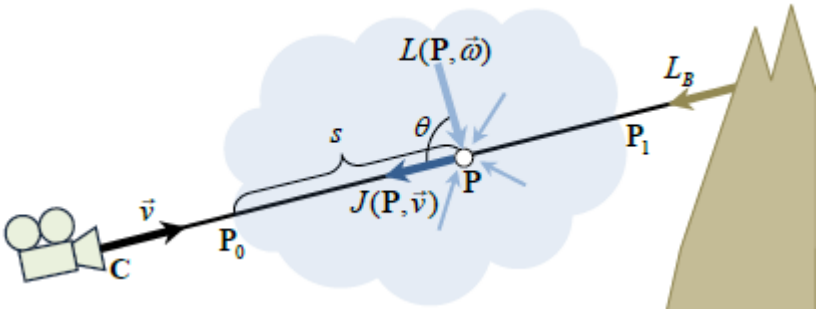


Figure 2.6: Light scattering in the cloud

Now let's say, we have a point **A** and point **B**, the normal direction from point **A** to **B** is $\vec{r} = \frac{B-A}{|B-A|}$. We denote the current integration point, which has one step size from point **A** to **B**, by **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 $\vec{v}$. The author proposed that the final light $L_{In}$ getting into the camera can be calculated in the following equation:

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

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. $J(P, \vec{v})$ is the total radiance of light at point **P** towards the camera:

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

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 $\vec{v}$, where $\theta$ is the angle between two.

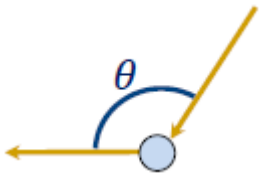


Figure 2.7: The angle $\theta$ between the incident direction and outgoing direction of the light

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

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

If we represent $L_{In}$ as the infinite sum of the intensities of the light scattered $n$ times:

$$L_{In} = \sum_{n=0}^{\infty} L_{In}^{(n)}, \tag{2.6}$$

where

$$L_{In}^{(n)}(C, \vec{v}) = \int_{P_0}^{P_1} e^{-\tau(P,P_0)} \cdot J(P, \vec{v}) \cdot \beta(P) \cdot ds \tag{2.7}$$

and

$$J^{(n)}(P, \vec{v}) = \int_{\Omega} L_I n(P, \vec{\omega}) \cdot P(\theta) \cdot d\omega \tag{2.8}$$

The author noted that the initial scattered light $L_{In}^{(0)}$ is the radiance of light scattered 0 times, which is the external radiance attenuated by the cloud.

The author also indicated that the final radiance at the camera position is the sum of in-scattered light $L_{In}$ and the attenuated background radiance $L_B$:

$$L(C, \vec{v}) = L_{In}(C, \vec{v}) + e^{-\tau(P_0,P_1)} \cdot L_B. \tag{2.9}$$

# Chapter 3

# Algorithm and Implementation

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

## 3.1 Generating Particles

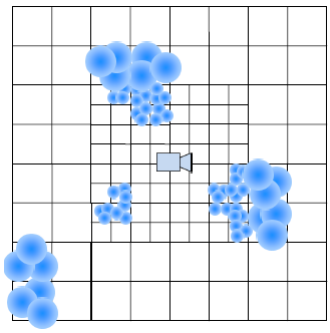

Figure 3.1: Camera-centered particle grid

The author proposed with an interesting fully procedural method to generate the particles using a camera-centered particle grid shown above.

### 3.1.1 The Design

This grid has several cells, each one containing several particle layers. The number of the layers, the particle size and the density are determined by 2D noise texture. Usually, the particles in each next ring have twice the size of the inner ring.

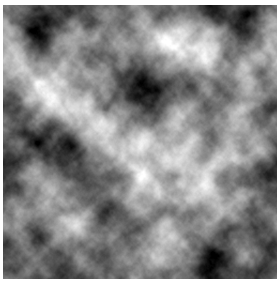

Figure 3.2: A typical noise texture

### 3.1.2 The Implementation

The generation process is performed on GPU by the following steps:

1. Process each cell, compute the cloud density of this cell, and create a list of valid non-empty cells according to the 2D noise texture. In this step, one shader thread process one cell, and a buffer is used to store the indices of valid cells.

2. Process the list of valid cells from the previous step. Generate one or more particles for each cell depending on the number of the layers and cloud density in the cell. In this step, one shader thread process one valid cell and generates particles.

3. Generate an ordered list of particles because they must be rendered in back to front order for the blending purpose. However, sorting on GPU is quite expensive, so the author provided a solution that using streaming output the particles only for valid cells to preserve the order, and then process 32 particles by one GS thread.

## 3.2 Pre-Computing 4D Look-up Tables

The current graphics hardware does not support 4D textures, so the author used 3D textures with manual interpolation for the forth coordinate to emulate the 4D texture.

### 3.2.1 The Design

The optical depth integral is stored in a $64 \times 32 \times 64 \times 32$ 8-bit look-up table which occupies 4 MB storage space. Single and multiple scattering are stored in two $32 \times 64 \times 16 \times 8$ 16-bit float look-up tables. Each table occupies 0.5 MB storage space.

### 3.2.2 The Implementation

```
#define SAMPLE_4D_LUT(tex3DLUT, LUT_DIM, f4LUTCoords, fLOD, Result)
{
    float3 f3UVW;
    f3UVW.xy = f4LUTCoords.xy;
    float fQSlice = f4LUTCoords.w * LUT_DIM.w - 0.5;
```

```
    float fQ0Slice = floor(fQSlice);
    float fQWeight = fQSlice - fQ0Slice;

    f3UVW.z = (fQ0Slice + f4LUTCoords.z) / LUT_DIM.w;
    /* frac() assures wraparound filtering of w coordinate*/
    Result = lerp(tex3DLUT.SampleLevel(samLinearWrap, f3UVW, fLOD),
                                                    tex3DLUT.SampleLevel(
        samLinearWrap, frac(f3UVW + float3(0,0,1LUT_DIM.w)),
                        fLOD), fQWeight);
}
```

As we can see in the code snippet above, **SAMPLE_4D_LUT** is a utility function which returns an interpolated result into the 5th parameter from the first 4 parameters, which are 3D texture, dimension, coordinates and LOD index.

## 3.3 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.

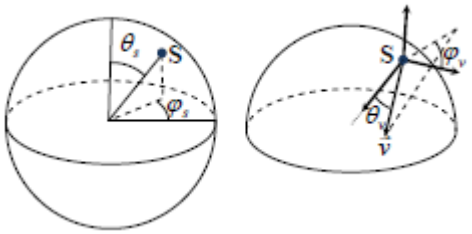


Figure 3.3: Four angels to describe a ray

### 3.3.1 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.

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.

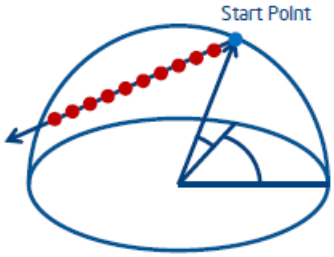


Figure 3.4: Numerically integrate the optical depth of a ray

### 3.3.2 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.

## 3.4 Pre-Computing Scattering

In this step, the author proposed with two types of scattering: single scattering and multiple scattering. In nature, a photon inside a cloud is scattered multiple times before it leaves the cloud. Hence the author concentrated on the multiple scattering in order to render a realistic cloud appearance. The goal is to calculate multiple scattering for every possible light position, camera position and orientation.

### 3.4.1 Single Scattering

#### The Design

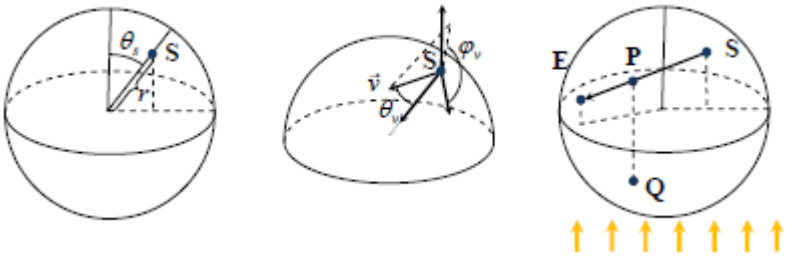


Figure 3.5: Pre-computing single scattering inside the spherical particle

First of all, the author assumed that the particle density only depends on the distance to the center. As the particle is a perfect sphere, which has the nature of symmetry, the author indicated that an arbitrary light direction coinciding with positive $z$ axis can be chosen to examine one of the rays that intersect the sphere. Since the light field is also symmetrical, only one angle $\theta_S$ is needed to describe the start point of a ray. As discussed in the previous section, another 2 angles $\phi_v$ and $\theta_v$ will still be needed to describe the ray direction. As the light field inside the volumes is important to solve the scattering equation, the author introduced a 4D look-up table with 3 angles($\theta_S$, $\phi_v$ and $\theta_v$) and the fraction of distance $r$ from the center of the sphere. We denote the table by

$$\overline{L}_{In}^{(1)}(\theta_S, \phi_v, \theta_v, r), \theta_S \in [0, \pi], \phi_v \in [0, \pi], \theta_v \in [0, \pi], r \in [0, 1]. \tag{3.1}$$

The author then proposed with a solution to single scattering inside the spherical particle:

$$\overline{L}_{In}^{(1)} = \int_S^E e^{-\tau(P,S)} e^{-\tau(P,Q)} \cdot \beta(P) \cdot ds, \tag{3.2}$$

where **S** is the start point of the ray and **E** is the exit point of the ray. **Q** is the point on the surface of the sphere through which the light reaches the current integration point **P**. The author finally implied that the phase function $P(\theta)$ is only used at run time to avoid precision issues. Just like the optical depth calculation, the extinction coefficient $\beta(P)$ can be evaluated in different methods as long as the distance to the center is the only influencing factor. The author tried different methods and finally decided to make the density constant.

### The Implementation

Let's have a look at the implementation detail in code.

```
float PrecomputeSingleSctrPS(SScreenSizeQuadVSOutput In) : SV_Target
{
    float4 f4LUTCoords = float4(ProjToUV(In.m_f2PosPS), g_GlobalCloudAttribs.f4Parameter.
        xy);
    float3 f3EntryPointUSSpace, f3ViewRayUSSpace, f3LightDirUSSpace;
    float fDensityScale;
    ParticleScatteringLUTToWorldParams(f4LUTCoords, g_GlobalCloudAttribs.f4Parameter.z,
        f3EntryPointUSSpace, f3ViewRayUSSpace, f3LightDirUSSpace, false, fDensityScale);
    // Intersect view ray with the unit sphere:
    float2 f2RayIsecs;
    // f3NormalizedStartPos  is located exactly on the surface; slightly move the start
        pos inside the sphere
    // to avoid precision issues
    float3 f3BiasedEntryPoint = f3EntryPointUSSpace + f3ViewRayUSSpace*1e-4;
    GetRaySphereIntersection(f3BiasedEntryPoint, f3ViewRayUSSpace, 0, 1.f, f2RayIsecs);
    if( f2RayIsecs.y < f2RayIsecs.x ) return 0;
    float3 f3EndPos = f3BiasedEntryPoint + f3ViewRayUSSpace * f2RayIsecs.y;
    float fNumSteps = NUM_INTEGRATION_STEPS;
    float3 f3Step = (f3EndPos - f3EntryPointUSSpace) / fNumSteps;
    float fStepLen = length(f3Step);
    float fCloudMassToCamera = 0;
```

```
    float fParticleRadius = GetParticleSize(GetCloudRingWorldStep(0, g_GlobalCloudAttribs
        ));
    float fInscattering = 0;
    for(float fStepNum=0.5; fStepNum < fNumSteps; ++fStepNum)
    {
        float3 f3CurrPos = f3EntryPointUSSpace + f3Step * fStepNum;
        float fDensity = 1;
        fDensity *= fDensityScale;
        float fCloudMassToLight = 0;
        GetRaySphereIntersection(f3CurrPos, f3LightDirUSSpace, 0, 1.f, f2RayIsecs);
        if( f2RayIsecs.y > f2RayIsecs.x )
        {
            fCloudMassToLight = abs(f2RayIsecs.x) * fParticleRadius;
        }
        float fTotalLightAttenuation = exp( -g_GlobalCloudAttribs.fAttenuationCoeff * (
            fCloudMassToLight + fCloudMassToCamera) );
        fInscattering += fTotalLightAttenuation * fDensity * g_GlobalCloudAttribs.
            fScatteringCoeff;
        fCloudMassToCamera += fDensity * fStepLen * fParticleRadius;
    }
    return fInscattering * fStepLen * fParticleRadius;
}
```

Similar to the function calculating optical depth, this function **PrecomputeSingleSctrPS** firstly decomposes the given projection position and the 4 parameters from the 4D look-up table into 3 vec3 positions: the first one **f3EntryPointUSSpace** describes the start position of the ray, the second one **f3ViewRayUSSpace** describes the ray direction, the last one **f3LightDirUSSpace** describes the light direction. Using these 3 components, the density scale is calculated. Similarly, the intersection view ray is evaluated carefully avoiding the precision issues by moving the starting point slightly into the sphere by the amount of **f3ViewRayUSSpace*1e-4**. Then the integration is performed in the **for loop**. In each step, we calculate the distance **fCloudMassToLight** from center of the cloud to the light, and the distance **fCloudMassToCamera** from the center of the cloud to the camera. Then we can evaluate the total light attenuation **fTotalLightAttenuation**. Finally, we accumulate the single scattering result **fInscattering** and **fCloudMassToCamera** because we are moving forward. Why didn't the author accumulate **fCloudMassToLight**? That's because the light source is treated as directional light, which means the light source is in an infinite distance.

### 3.4.2 Multiple Scattering

Multiple scattering is based single scattering using Equation 3.2 $\overline{L}_{In}^{(1)} = \int_S^E e^{-\tau(P,S)} e^{-\tau(P,Q)} \cdot \beta(P) \cdot ds$. However, this time, we need to calculate $\overline{L}_{In}^{(n)}$.

#### The Design

The author proposed with 3 steps:

1. Evaluate $\overline{J}^{(n)}(\theta_S, \phi_v, \theta_v, r)$ for every point and direction inside he sphere by solving Equation 2.8: $J^{(n)}(P, \vec{v}) = \int_\Omega L_I n(P, \vec{\omega}) \cdot P(\theta) \cdot d\omega$.

2. Evaluate $\overline{L}_{In}^{(n)}(\theta_S, \phi_v, \theta_v, r)$ as the current order in scattering by solving Equation 2.7: $L_{In}^{(n)}(C, \vec{v}) = \int_{P_0}^{P_1} e^{-\tau(P,P_0)} \cdot J(P, \vec{v}) \cdot \beta(P) \cdot ds$.

3. Accumulate current scattering order to the total look-up table: $\overline{L}_{In}^{M} = \overline{L}_{In}^{M} + \overline{L}_{In}^{(n)}$.

As a result, the figure below shows single, multiple and final lighting for the sphere.

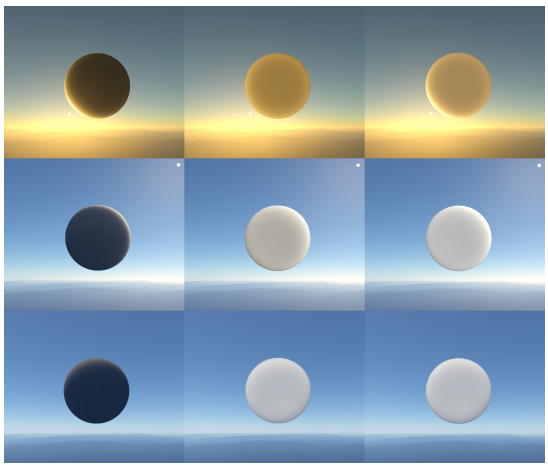

Figure 3.6: Pre-computed scattering for different light orientations. From left to right: single scattering only, 2 + scattering only, all terms (including ambient). The particle in the bottom row is illuminated from above

### The Implementation

We'll look into the implementation details of the 3 steps in this section.

1. Compute $\overline{J}^{(n)}$ for each point and direction inside the sphere.

```
float GatherScatteringPS(SScreenSizeQuadVSOutput In) : SV_Target
{
    float4 f4LUTCoords = float4(ProjToUV(In.m_f2PosPS), g_GlobalCloudAttribs.
        f4Parameter.xy);

    float3 f3PosUSSpace, f3ViewRayUSSpace, f3LightDirUSSpace;
    ParticleScatteringLUTToWorldParams(f4LUTCoords, f3PosUSSpace, f3ViewRayUSSpace,
         f3LightDirUSSpace, false);

    float3 f3LocalX, f3LocalY, f3LocalZ;
    ConstructLocalFrameXYZ(-normalize(f3PosUSSpace), f3LightDirUSSpace, f3LocalX,
        f3LocalY, f3LocalZ);

    float fGatheredScattering = 0;
    float fTotalSolidAngle = 0;
    const float fNumZenithAngles = VOL_SCATTERING_IN_PARTICLE_LUT_DIM.z;
    const float fNumAzimuthAngles = VOL_SCATTERING_IN_PARTICLE_LUT_DIM.y;
    const float fZenithSpan = PI;
    const float fAzimuthSpan = 2*PI;
    for(float ZenithAngleNum = 0.5; ZenithAngleNum < fNumZenithAngles; ++
        ZenithAngleNum)
```

```
        for(float AzimuthAngleNum = 0.5; AzimuthAngleNum < fNumAzimuthAngles; ++
            AzimuthAngleNum)
        {
            float ZenithAngle = ZenithAngleNum/fNumZenithAngles * fZenithSpan;
            float AzimuthAngle = (AzimuthAngleNum/fNumAzimuthAngles - 0.5) *
                fAzimuthSpan;
            float3 f3CurrDir = GetDirectionInLocalFrameXYZ(f3LocalX, f3LocalY,
                f3LocalZ, ZenithAngle, AzimuthAngle);
            float4 f4CurrDirLUTCoords = WorldParamsToParticleScatteringLUT(
                f3PosUSSpace, f3CurrDir, f3LightDirUSSpace, false);
            float fCurrDirScattering = 0;
            SAMPLE_4D_LUT(g_tex3DPrevSctrOrder, VOL_SCATTERING_IN_PARTICLE_LUT_DIM,
                 f4CurrDirLUTCoords, 0, fCurrDirScattering);
            if( g_GlobalCloudAttribs.f4Parameter.w == 1 )
            {
                fCurrDirScattering *= HGPhaseFunc( dot(-f3CurrDir,
                    f3LightDirUSSpace) );
            }
            fCurrDirScattering *= HGPhaseFunc( dot(f3CurrDir, f3ViewRayUSSpace),
                0.7 );

            float fdZenithAngle = fZenithSpan / fNumZenithAngles;
            float fdAzimuthAngle = fAzimuthSpan / fNumAzimuthAngles * sin(
                ZenithAngle);
            float fDiffSolidAngle = fdZenithAngle * fdAzimuthAngle;
            fTotalSolidAngle += fDiffSolidAngle;
            fGatheredScattering += fCurrDirScattering * fDiffSolidAngle;
        }

    // Total solid angle should be 4*PI. Renormalize to fix discretization issues
    fGatheredScattering *= 4*PI / fTotalSolidAngle;

    return fGatheredScattering;
}
```

Similarly, in this function **GatherScatteringPS**, the given projection position and the 4 parameters from the 4D look-up table is decomposed to 3 vec3 positions: the first one **f3PosUSSpace** describes the start position of the ray, the second one **f3ViewRayUSSpace** describes the ray direction, the last one **f3LightDirUSSpace** describes the light direction. Then we use the **zenith angles** $\varphi_v$ and **azimuth angles** $\theta_v$ count as the total steps, and we do the integration inside the two nested for loops. In each step, $\varphi_v$ and $\theta_v$ are increased by a step size. According to $\varphi_v$, $\theta_v$ and the start point, we calculate the ray direction **f3CurrDir**. After this, we transform the ray from world space back to the particle space. Then we check if the distance of the 4D table is 1 in order to calculate the current scattering direction **fCurrDirScattering** correctly. Finally, we accumulate the total scattering in each step by the amount of multiplying the current scattering direction **fCurrDirScattering** to the delta angle **fDiffSolidAngle**.

2. Compute $\overline{L}_{In}^{(n)}$ as the current order.

```
float ComputeScatteringOrderPS(SScreenSizeQuadVSOutput In) : SV_Target
{
```

```
    float4 f4StartPointLUTCoords = float4(ProjToUV(In.m_f2PosPS),
        g_GlobalCloudAttribs.f4Parameter.xy);

    float3 f3PosUSSpace, f3ViewRayUSSpace, f3LightDirUSSpace;
    ParticleScatteringLUTToWorldParams(f4StartPointLUTCoords, f3PosUSSpace,
        f3ViewRayUSSpace, f3LightDirUSSpace, false);

    // 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
    float3 f3BiasedPos = f3PosUSSpace + f3ViewRayUSSpace*1e-4;
    GetRaySphereIntersection(f3BiasedPos, f3ViewRayUSSpace, 0, 1.f, f2RayIsecs);
    if( f2RayIsecs.y < f2RayIsecs.x )
        return 0;

    float3 f3EndPos = f3BiasedPos + f3ViewRayUSSpace * f2RayIsecs.y;
    float fNumSteps = max(VOL_SCATTERING_IN_PARTICLE_LUT_DIM.w*2,
        NUM_INTEGRATION_STEPS)*2;
    float3 f3Step = (f3EndPos - f3PosUSSpace) / fNumSteps;
    float fStepLen = length(f3Step);
    float fCloudMassToCamera = 0;
    float fParticleRadius = g_GlobalCloudAttribs.fReferenceParticleRadius;
    float fInscattering = 0;

    float fPrevGatheredSctr = 0;
    SAMPLE_4D_LUT(g_tex3DGatheredScattering, VOL_SCATTERING_IN_PARTICLE_LUT_DIM,
        f4StartPointLUTCoords, 0, fPrevGatheredSctr);
        // Light attenuation == 1
    for(float fStepNum=1; fStepNum <= fNumSteps; ++fStepNum)
    {
        float3 f3CurrPos = f3PosUSSpace + f3Step * fStepNum;

        fCloudMassToCamera += fStepLen * fParticleRadius;
        float fAttenuationToCamera = exp( -g_GlobalCloudAttribs.fAttenuationCoeff *
             fCloudMassToCamera );

        float4 f4CurrDirLUTCoords = WorldParamsToParticleScatteringLUT(f3CurrPos,
            f3ViewRayUSSpace, f3LightDirUSSpace, false);
        float fGatheredScattering = 0;
        SAMPLE_4D_LUT(g_tex3DGatheredScattering, VOL_SCATTERING_IN_PARTICLE_LUT_DIM
            , f4CurrDirLUTCoords, 0, fGatheredScattering);
        fGatheredScattering *= fAttenuationToCamera;

        fInscattering += (fGatheredScattering + fPrevGatheredSctr) /2;
        fPrevGatheredSctr = fGatheredScattering;
    }

    return fInscattering * fStepLen * fParticleRadius * g_GlobalCloudAttribs.
        fScatteringCoeff;
}
```

As always, firstly the function **ComputeScatteringOrderPS** computes the ray's start position, the direction, and the light direction. Again, moving the start point slightly inside the sphere to avoid precision issues. Then we calculate the intersected ray result,

and step up some basic variables for the integration. We also calculate the interpolated result from the 3D texture of gathered scattering texture and store it into a variable called **fPrevGatheredSctr** as the initial scattering. In each integration step, we multiply the gathered scattering by the attenuation to the camera **fAttenuationToCamera**. Finally we accumulate the final scattering light into the camera **fInscattering** with the average amount of the scattering in this step **fGatheredScattering** and the scattering from the previous step **fPrevGatheredSctr**.

3. Accumulate current scattering order to the total look-up table: $\overline{L}_{In}^{M} = \overline{L}_{In}^{M} + \overline{L}_{In}^{(n)}$.

```
float AccumulateMultipleScattering(SScreenSizeQuadVSOutput In) : SV_Target
{
    float3 f3LUTCoords = float3(ProjToUV(In.m_f2PosPS), g_GlobalCloudAttribs.
        f4Parameter.x);
    float fMultipleSctr = g_tex3DPrevSctrOrder.SampleLevel(samPointWrap,
        f3LUTCoords, 0);
    return fMultipleSctr;
}
```

This function **AccumulateMultipleScattering** is quite straightforward, it accumulates the multiple scattering.

## 3.5 Estimating Occlusion

We already have the scattering data, by this point, the cloud is still semi-transparent no matter how thick it is.

### 3.5.1 The Design

The occlusion describes the attenuation by other particles as the light travels to the current particle. Thus, computing the light occlusion is necessary.

### 3.5.2 The Implementation

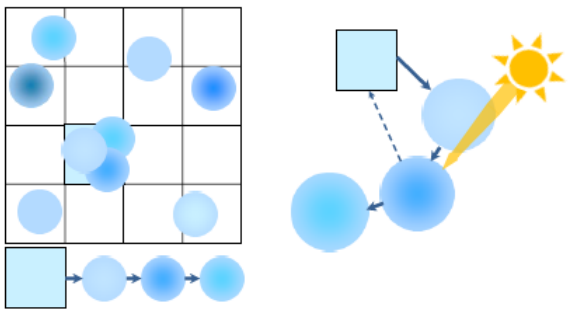

Figure 3.7: Light-space tile grid (left) and computing light occlusion by traversing the tiles list (right)

The occlusion is estimated in the following 3 steps:

1. Perform light-space tiling and construct lists of particles covering each tile. In this step, one tile is actually one pixel, and each particle is assigned to one tile. A buffer of the screen size is used to store the index of the first particle in the list and another buffer is used to store the lists elements. As discussed in the first section of this chapter, it is important to preserve the original particle order for blending purpose. Hence, the Pixel Shader Ordering is used.

2. Traverse the particle lists from the previous step to estimate the occlusion. This step is carried out by the compute shader. Since each particle is assigned to one tile, the shader traverse through the list of the tile and calculate the opcity of particles on the light path. As the particles are in back-to-front order, the traversing loop for the compute shader can be terminated immediately when the current particle is reached. Or the programmer can set a threshold for the total transparency. The threshold makes sure that the cloud will never be totally semi-transparent at some certain transparency level. And it is encouraged to set a threshold for faster performance.

3. Smooth the occlusion and simulates the diffusion for the whole cloud. A compute shader is responsible for processing each valid particle and performing low-pass filtering of the light occlusion. The author also mentioned that using particle density as weights is a good way to avoid incorrect smoothing.

## 3.6 Generating Real-time Shading Model

In this section, the author proposed with an approximated solution to Equation 2.9: $L(C, \vec{v}) = L_{In}(C, \vec{v}) + e^{-\tau(P_0,P_1)} \cdot L_B$ using the results from the previous two steps, which are the **optical depth** $\overline{T}$, the **single scattering** $\overline{L}_{in}^{(1)}$ and the multiple scattering $\overline{L}_{in}^{(M)}$.

To generate a shading model, the first step is getting the optical depth $\overline{\tau}$ for the given ray from $\overline{T}$:

$$\overline{\tau} = \overline{T}(\varphi_S, \theta_S, \varphi_v, \theta_v). \tag{3.3}$$

### 3.6.1 The Design

The lighting model has three components: single scattering, multiple scattering and ambient.

1. The author assumed that the single scattering $\bar{l}^{(1)}$ in the light model is equal to single scattering in the homogeneous sphere:

$$\bar{l}^{(1)} = P(\theta) \cdot \overline{L}^{(1)}(\theta_S, \phi_v, \theta_v, \overline{\rho} \cdot \overline{\tau}), \tag{3.4}$$

   where $\overline{\rho}$ is the density scale of each particle and $P(\theta)$ is this the phase function discussed in Chapter 2.

2. However, the multiple scattering only depends on the total density scale of the whole particle, the individual optical depth and phase function do not have any impact. Hence, the multiple scattering can be evaluated using the equation below:

$$\bar{l}^{(M)} = \overline{L}^{(M)}(\theta_S, \phi_v, \theta_v, \overline{\rho}) \tag{3.5}$$

3. The ambient light $\bar{l}^A$ is used to make the creases of the cloud slightly brighter than the edges influenced by the distance to the first cloud. And this distance is stored in the 4D look-up table $\overline{T}$. It is used at run time for the ambient light intensity of the sky.

### 3.6.2 The Implementation

```
void GetSunLightExtinctionAndSkyLight(in float3 f3PosWS,
                                      out float3 f3Extinction,
                                      out float3 f3AmbientSkyLight,
                                      Texture2D<float2> tex2DOccludedNetDensityToAtmTop,
                                      Texture2D<float3> tex2DAmbientSkylight )
{
    float3 f3EarthCentre = float3(0, -g_MediaParams.fEarthRadius, 0);
    float3 f3DirFromEarthCentre = f3PosWS - f3EarthCentre;
    float fDistToCentre = length(f3DirFromEarthCentre);
    f3DirFromEarthCentre /= fDistToCentre;
    float fHeightAboveSurface = fDistToCentre - g_MediaParams.fEarthRadius;
    float fCosZenithAngle = dot(f3DirFromEarthCentre, g_LightAttribs.f4DirOnLight.xyz);

    float fRelativeHeightAboveSurface = fHeightAboveSurface / g_MediaParams.fAtmTopHeight
        ;
    float2 f2ParticleDensityToAtmTop = g_tex2DOccludedNetDensityToAtmTop.SampleLevel(
        samLinearClamp, float2(fRelativeHeightAboveSurface, fCosZenithAngle*0.5+0.5), 0).
        xy;

    float3 f3RlghOpticalDepth = g_MediaParams.f4RayleighExtinctionCoeff.rgb *
        f2ParticleDensityToAtmTop.x;
    float3 f3MieOpticalDepth  = g_MediaParams.f4MieExtinctionCoeff.rgb      *
        f2ParticleDensityToAtmTop.y;

    // And total extinction for the current integration point:
    f3Extinction = exp( -(f3RlghOpticalDepth + f3MieOpticalDepth) );

    f3AmbientSkyLight = tex2DAmbientSkylight.SampleLevel(samLinearClamp, float2(
        fCosZenithAngle*0.5+0.5, 0.5), 0);
}
```

This function **GetSunLightExtinctionAndSkyLight** basically calculate the ambient sky light **f3AmbientSkyLight** from the look-up table $\overline{T}$. As we can see, we get the earth center position **f3EarthCentre** first, and use it to calculate the distance between the earth center to the particle **fDistToCentre**. Then, the altitude of the particle **fHeightAboveSurface** and the cos value of the zenith angle **fCosZenithAngle** are evaluated. After this, we retrieve the optical depth from the pre-computed 4D look-up tale. Finally, we calculate the ambient sky light using the data from above and sample it with a 2D texture describing the general ambient pattern.

# Appendix A

# Abbreviations

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

# Bibliography


[1] Egor Yusov (2014)
High-Performance Rendering of Realistic Cumulus Clouds Using Pre-computed Lighting

[2] Cloud Sky
https://github.com/GameTechDev/CloudySky

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

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

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

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

[7] Hufnagel R., Held M. (2012)
A survey of cloud lighting and rendering techniques

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

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