HDRP中体积雾的实现分享

HDRP中体积雾的实现分享

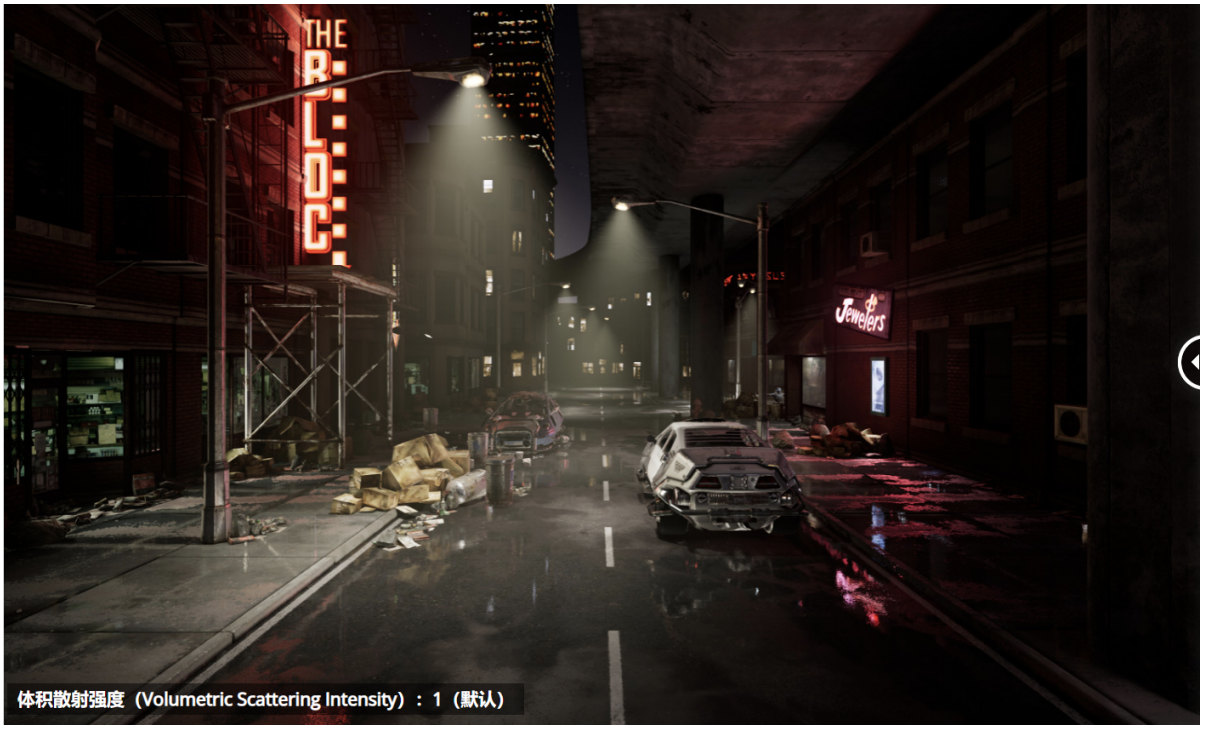
- 总览
- 理论部分
  - 参数部分
    - 吸收系数
    - 外散射系数
    - 总衰减系数 （单位：/m）
    - 平均自由程/mean free path
    - 反照率albedo
    - Beer–Lambert law
    - 相函数
      - 米氏散射
      - 瑞利散射
      - Henyey-Greenstein相函数
      - Cornette-Shanks相函数
    - 内散射
    - 单次散射
    - 多次散射
- 工程部分
  - 市面上体积雾展示
  - 参数
    - 艺术家参数
    - 管线参数
  - HDRP中体积雾的实现的基本原理
    - 算法过程
      - 视锥体素化
      - 高度雾计算
      - 体积雾计算
        - 切片数据准备
        - 绘制切片
        - 填充体积雾数据
      - 体积光着色
        - 抖动
        - 重投影
        - 高斯滤波
    - 管线
    - Pass
      - Clear and Height Fog Voxelization
      - Fog Volume Voxelization
        - Compute VolumetricMaterial
        - DrawProcedural Indexed Indirect
      - Volumetric Lighting
        - VolumetricLighting
        - Denoising
      - OpaqueAtmosphericScattering

总览

在现实世界中，空气中充满了微小的粒子（灰尘，水蒸气等），这些粒子都会跟光产生交互（吸收，散射等），从而产生一些不一样的光学现象。

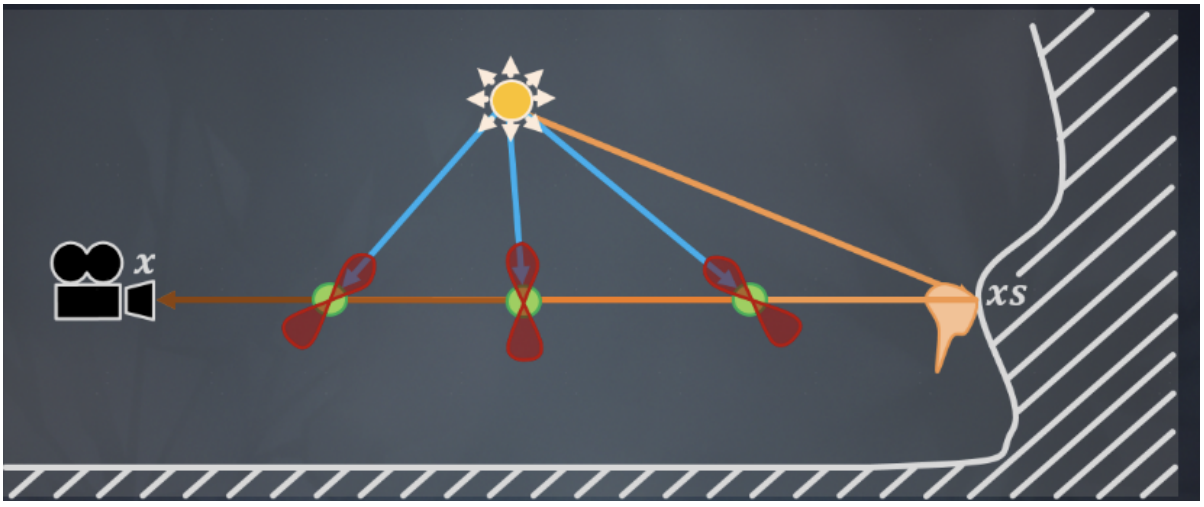
而在游戏渲染中，如果没有特殊处理，则我们看到的渲染结果，其实是相当于把空间当成真空，没有任何介质干扰关的传播而得到的结果，这个结果只考虑了物体表面着色。

当考虑了空气介质对于关传播的影响后，效果如下



我们在这里先考虑某一个着色点

如下图，实际上考虑空气介质的影响，我们需要考虑到光线传播过程中和空气中粒子的“反应”



然而由于算力的限制，实际渲染无法做到真实的粒子模拟，因此常规的做法是在传输路径上进行微分。如上图，假如只考虑路径上三个点，则最终摄像机接收到的着色结果为xs着色点的表面着色结果，依次计算经过每个点的结果，进行累加。

对于每个点，需要同时考虑外部光源（环境光，间接光，直接光等）的影响，如上图蓝色射线所示，对于入射的光，有一定概率会因为散射正好散射到观察方向，下文我们将通过一个描述函数来描述这个概率。

当然，如果点位划分更多，则结果会更精确，当划分无限多时，则近似于现实中的结果。

在渲染中，常见的做法是RayMarching和体素化的方式去模拟对光线传播路径的积分。

理论部分

参考： https://zhuanlan.zhihu.com/p/348973932

参数部分

吸收系数

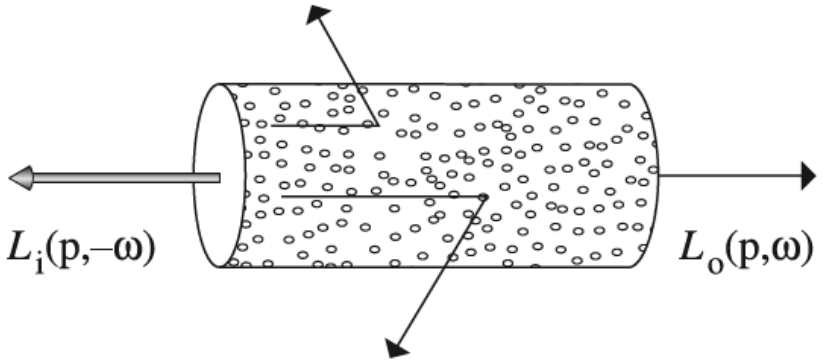
$$\sigma_a$$

光穿过单位距离时被吸收的概率

外散射系数

$$\sigma_s$$

光穿过单位距离时外散射的概率



总衰减系数 （单位：/m）

$$\sigma_t = \sigma_a + \sigma_s$$

平均自由程/mean free path

$$\rho = \frac{1}{\sigma_t}$$

反照率albedo

$$\rho = \frac{\sigma_s}{\sigma_t}$$

表现为介质外观颜色属性

Beer–Lambert law

描述两点之间光线的透光率

当介质内部完全均匀时

$$T_r(p \to p^{'}) = e^{-\sigma_t d}$$

当介质内部不均匀时

$$T_r(p \to p^{'}) = e^{-\int_0^d \sigma_t(p+t\omega,\omega)dt}$$

相函数

描述发生散射之后，散射方向的分布情况。

在各向同性完全均匀的介质中，散射方向分布也是均匀的，为单位球体总立体角的倒数

$$p(\omega_i, \omega_o) = \frac{1}{4\pi}$$

米氏散射

当微粒半径的大小**接近于**或者**大于**入射光线的波长λ的时候，大部分的入射光线会沿着**前进**的方向进行散射，这种现象被称为**米氏散射**。这种大微粒包括灰尘，水滴，来自污染物的颗粒物质，如烟雾等。

瑞利散射

瑞利散射是一种对应于颗粒尺寸远小于光波长的散射现象。波长较短的蓝光比波长较长的红光更易产生瑞利散射。

$$P(cos\theta) = \frac{3}{4}\frac{(1+cos^2\theta)}{\lambda^4}$$

Henyey-Greenstein相函数

当各项异性时，常用Henyey-Greenstein相函数来描述

$$P_{HG}(cos\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2+2g(cos\theta))^{\frac{3}{2}}}$$

其中g为非对称参数，取值范围 [-1, 1] , 当g<0,代表光更倾向于发生背散射，当g>0,则代表光更倾向于发生前散射。

Cornette-Shanks相函数

CS相函数是对于HG相函数的修正，在云的渲染上提供了更加符合物理的描述。

基础版公式为

$$P_{cs}(cos\theta) = \frac{3}{2}\frac{(1-g^2)}{(2+g^2)}\frac{(1+cos^2\theta)}{(1+g^2-2gcos\theta)^{\frac{3}{2}}}$$

对于云雾的渲染，主要关注前向散射部分，因此可以对公式进行简化

改进版公式为

$$P(cos\theta) = \frac{3}{2}\frac{(1-g^2)}{(2+g^2)}\frac{(1+cos^2\theta)}{(1+g^2-2gcos\theta)} + gcos\theta$$

下图是关于相函数的相关介绍

3.2 Common Phase Functions

Rayleigh phase function. As shown in formula (9), Rayleigh phase function is applicable to the case of Rayleigh scattering, which requires particle size is much smaller than the wavelength λ of incident light. Rayleigh phase function is simple and fast, but not close to the cloud optical properties. It was used in [19] and yielded good results. But there were some bright spots in some clouds shown as Fig. 5(a).

$$P(\phi) = \frac{3}{4}\frac{(1+\cos^2\theta)}{\lambda^4} \quad (9)$$

Henyey-Greenstein phase function. It described the scattering of radiation in the galaxy firstly [27]. It is useful in scattering calculation of biological organs, water, clouds and many other natural materials, and is an approximation of Mie scattering, expressed as formula (10).

$$P_{HG}(\phi) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g\cos\phi)^{3/2}} \quad (10)$$

where g is asymmetric factor, controlling the redistribution shapes of the scattered lights.

Cornette-Shanks phase function. Cornette and Shanks amended Henyey-Greenstein phase function and gave a more physically reasonable representation for clouds illumination expressed as formula (11). It was used in [28] while drawing cloud scenes.

$$P_{CS}(\phi) = \frac{3}{2}\frac{(1-g^2)}{(2+g^2)}\frac{(1+\cos^2\theta)}{(1+g^2-2g\cos\theta)^{3/2}} \quad (11)$$

3.3 Improved Phase Function

It can be seen that exponential computations are needed in Henyey-Greenstein and Cornette-Shanks phase functions, which make the computation time greatly improved. Rayleigh phase function is simple, but it is not suitable for the illumination computation of clouds. To solve these problems, improved phase function is proposed based on Cornette-Shanks phase function.

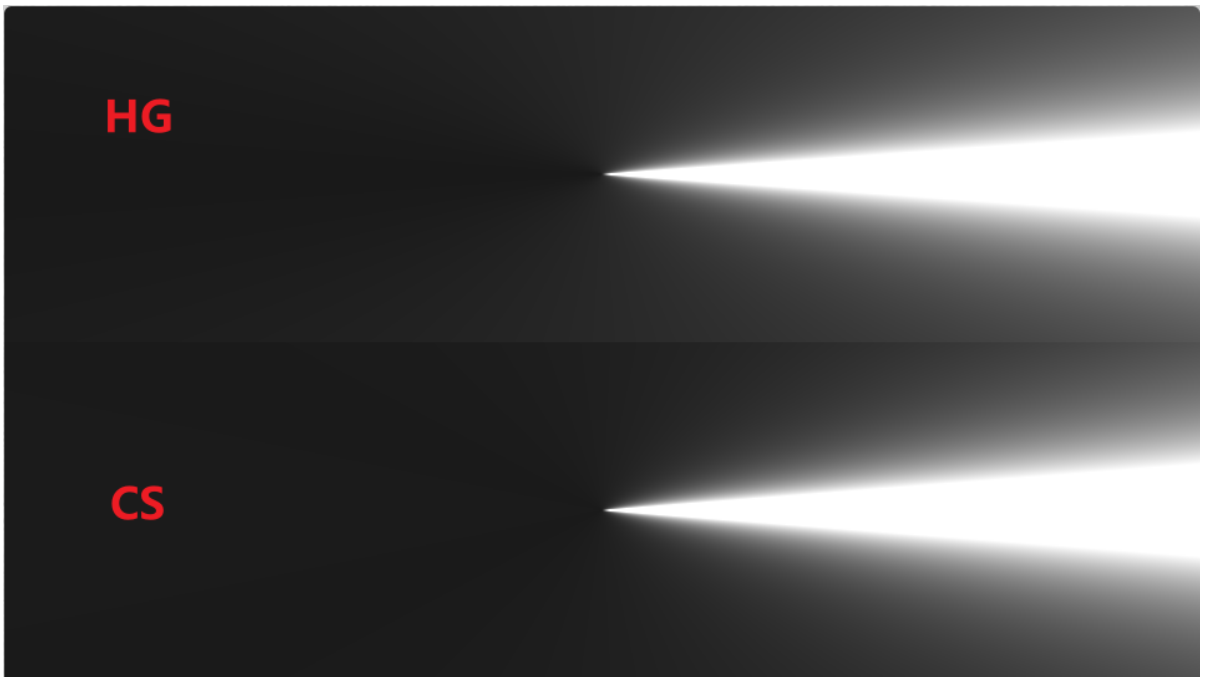
As the scattering of cloud particles focuses on the forward scattering, the exponential item in formula (11) is simplified, and another adjustment item is added. We can get the following simplified phase function defined as formula (12).

$$P(\phi) = \frac{3}{2}\frac{(1-g^2)}{(2+g^2)}\frac{(1+\cos^2\theta)}{(1+g^2-2g\cos\theta)} + g\cos\theta \quad (12)$$

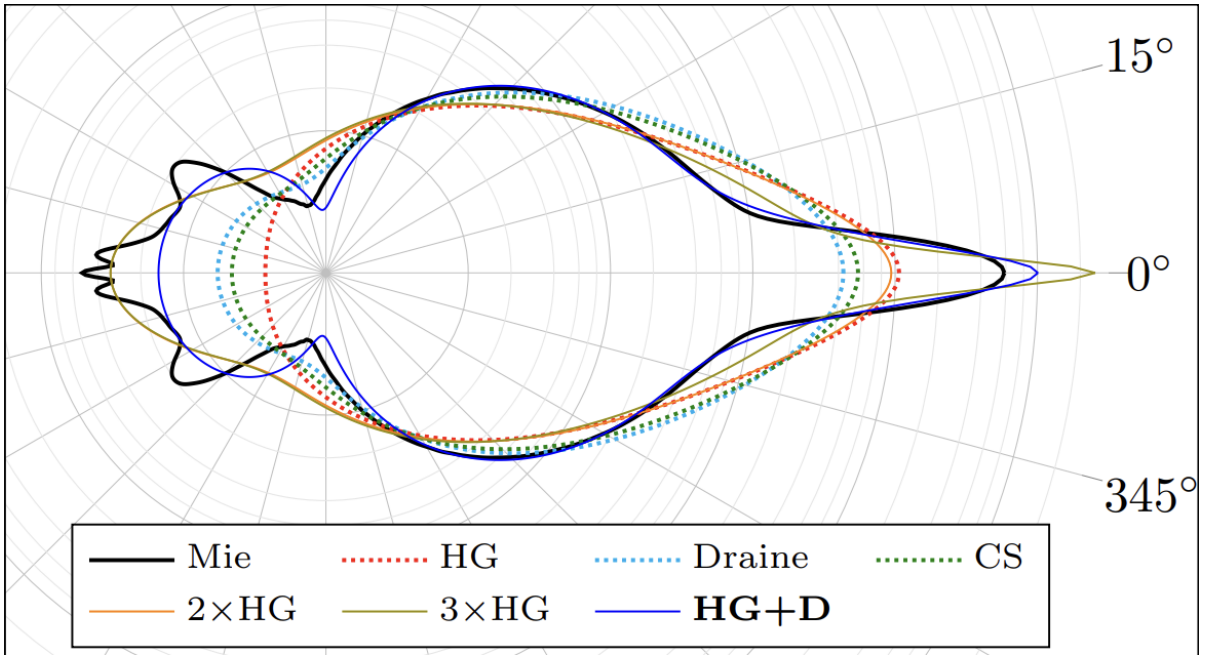
15

参考：http://www.csroc.org.tw/journal/JOC25-3/JOC25-3-2.pdf

以下为HG相函数跟CS相函数的效果对比

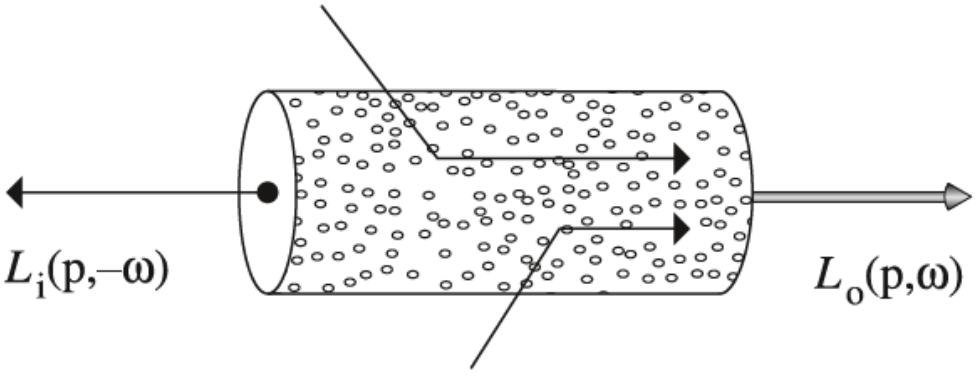


下图为对于米氏散射，各个相位方程的拟合情况



内散射

描述介质中的一个点接收到四面八方来的光线，正好散射到观察方向。



对于出射光，可以由微元的LightSource进行积分得出

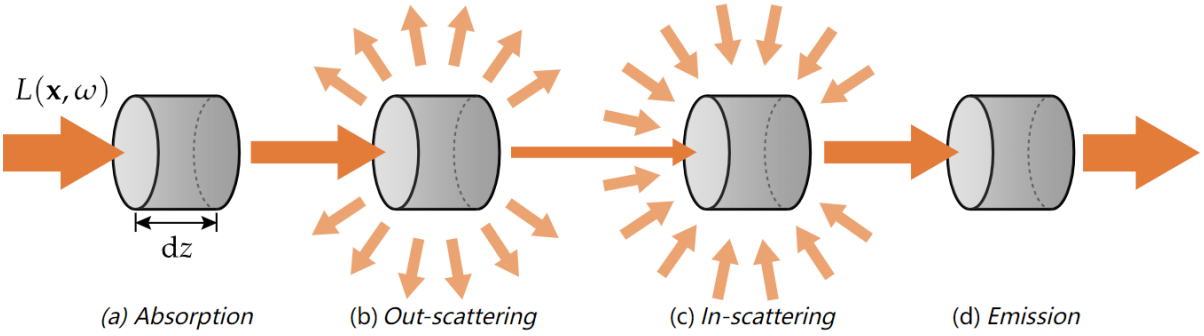
$$dL_o = L_s dt$$

其中，Ls考虑自发光的情况可以由下式得出

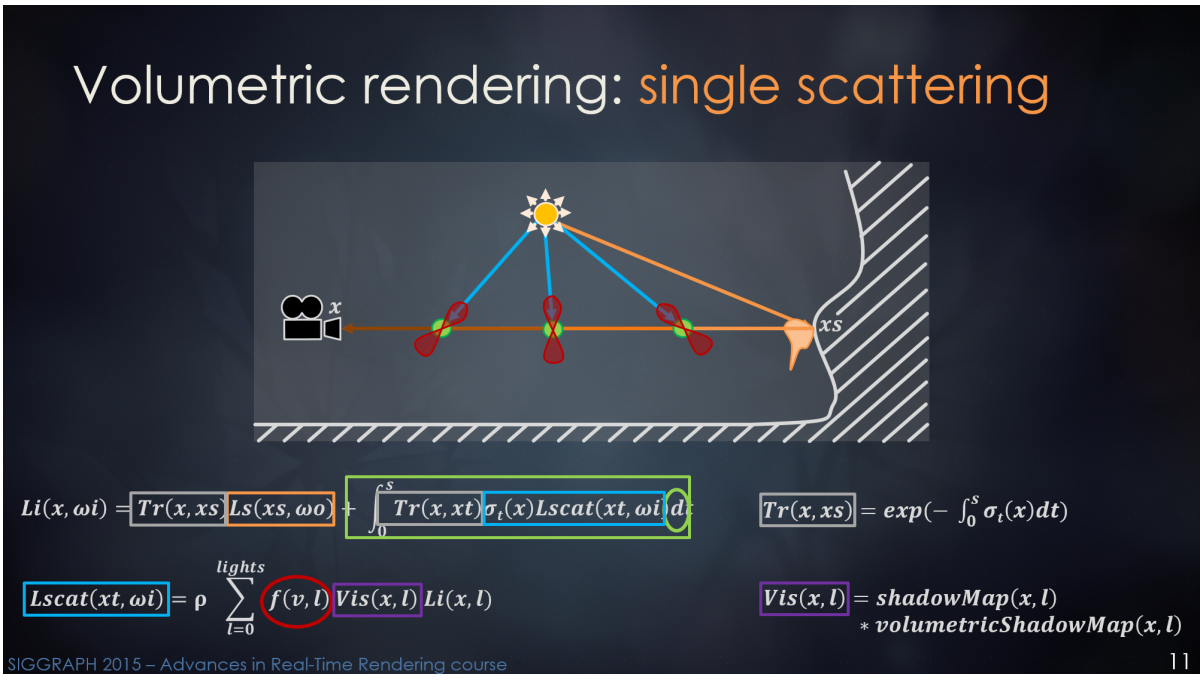
$$L_s = L_e + \sigma_s \int_{S^2} p(\omega_i, \omega_o) L_i d\omega$$

Le为自发光，p(wi, wo)为相函数，描述由wi摄入且朝wo方向散射的概率分布，然后对整个单位球进行积分。

单次散射



对于体积雾，多次散射效果很弱，所以一般只考虑单次散射的情况。下图为从xs着色点的着色结果到观察方向的传输过程，并且考虑了传输过程中的介质影响，也就是非真空状态下。



Li(x, wi): 观察到的radiance

Tr(x, xs): 为Beer-Lambert定律公式

Ls(xs, wo): 为xs这个着色点的着色结果

Lscat(xt, wi): 为所有影响传输介质中每一个微元的光源且散射向观察方向的radiance

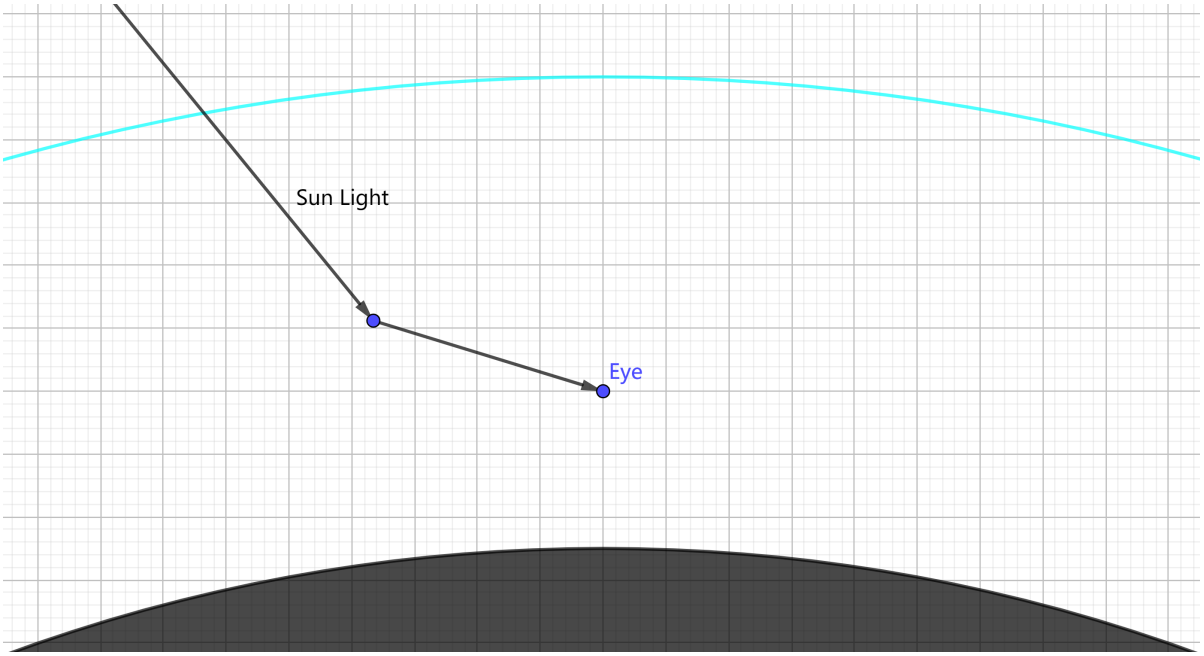
因此，上图绿色的积分，可以理解为在介质传输过程中，对每个微元同一个方向（观察方向）radiance的累加。

而计算**Lscat**需要对所有影响的光源进行累加，同时考虑可见度，即**Vis**项，因为只考虑观察方向，因此需要乘相位函数**f(v,l)**,类似brdf函数。

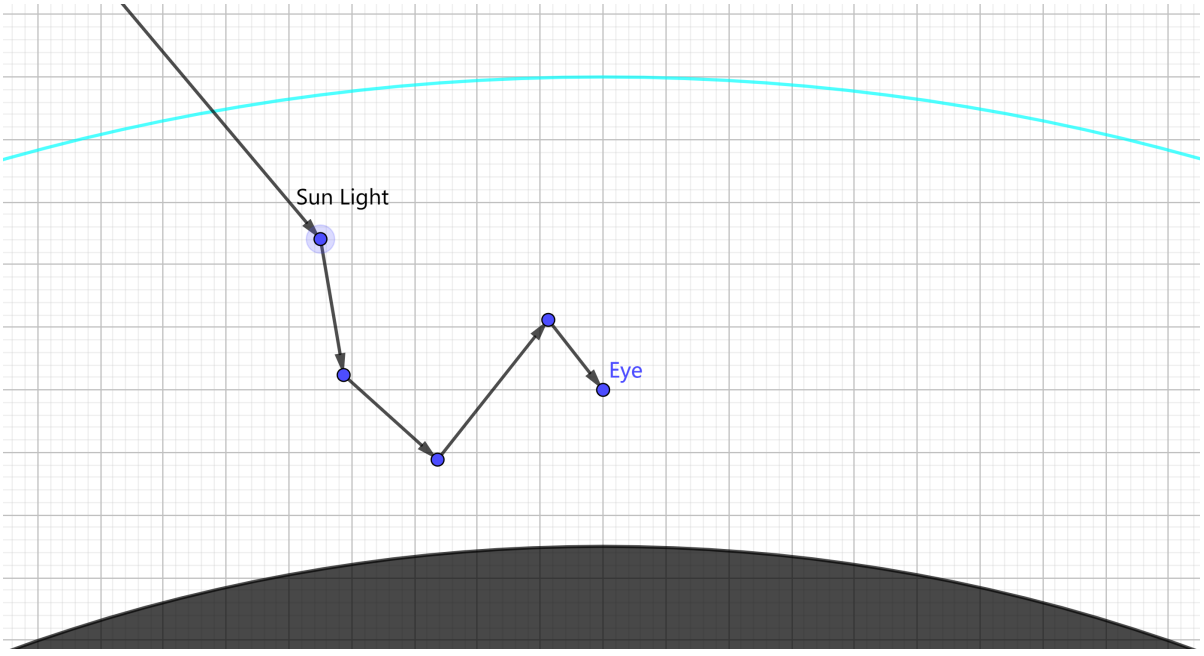
多次散射

对于大气渲染，多次散射跟单次散射的区别如下：

单次散射



多次散射



工程部分

市面上体积雾展示

都市天际线

https://www.bilibili.com/video/BV1wH4y1y7gk

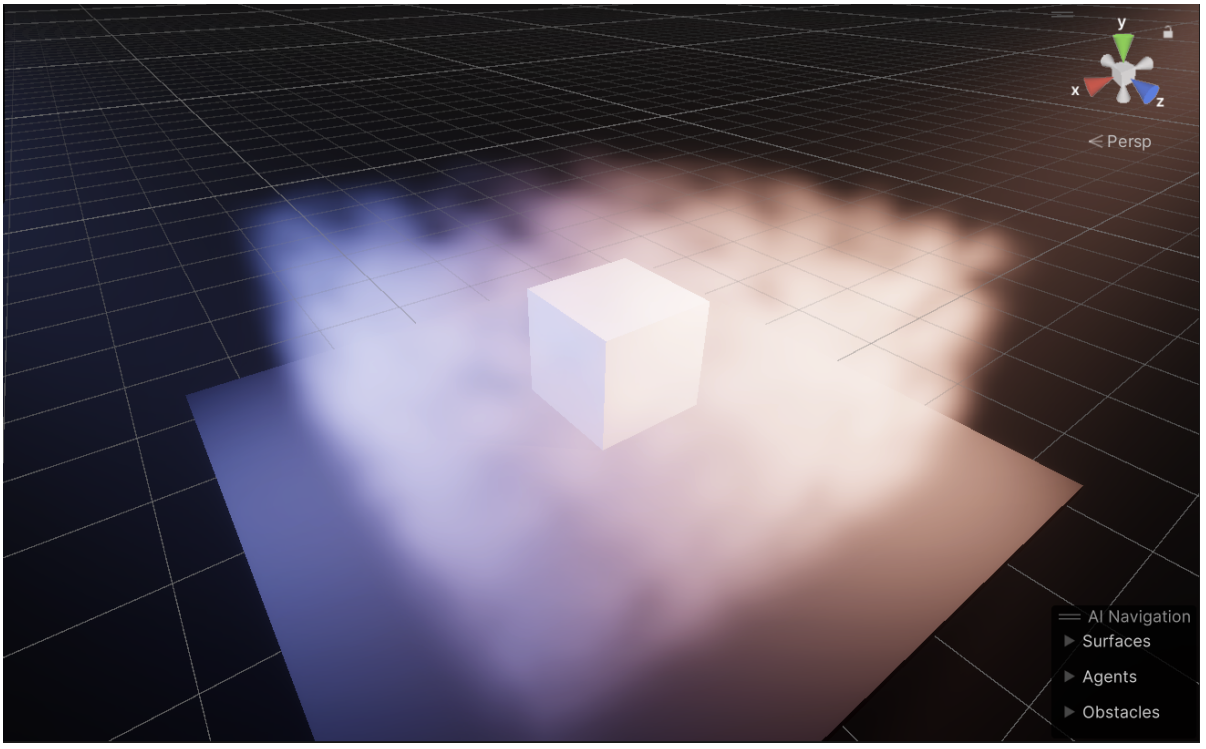
UE5体积雾插件

https://www.bilibili.com/video/BV1WV411S7zr

大镖客2

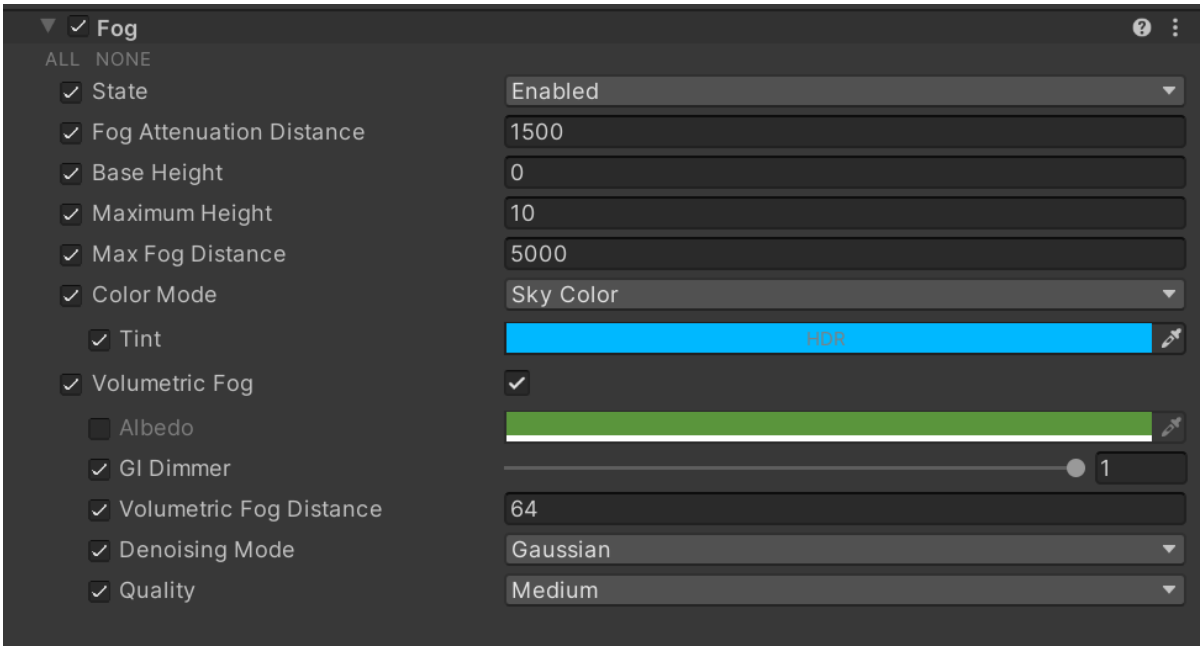
https://www.bilibili.com/video/BV1kb411N7Mj

HDRP体积雾效果展示

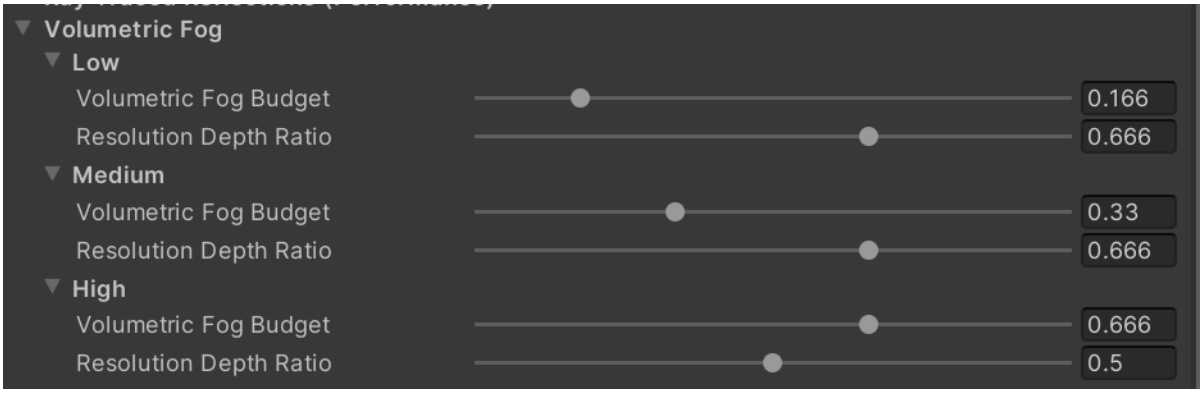

y
x
z
Persp
AI Navigation
Surfaces
Agents
Obstacles

参数

艺术家参数



管线参数

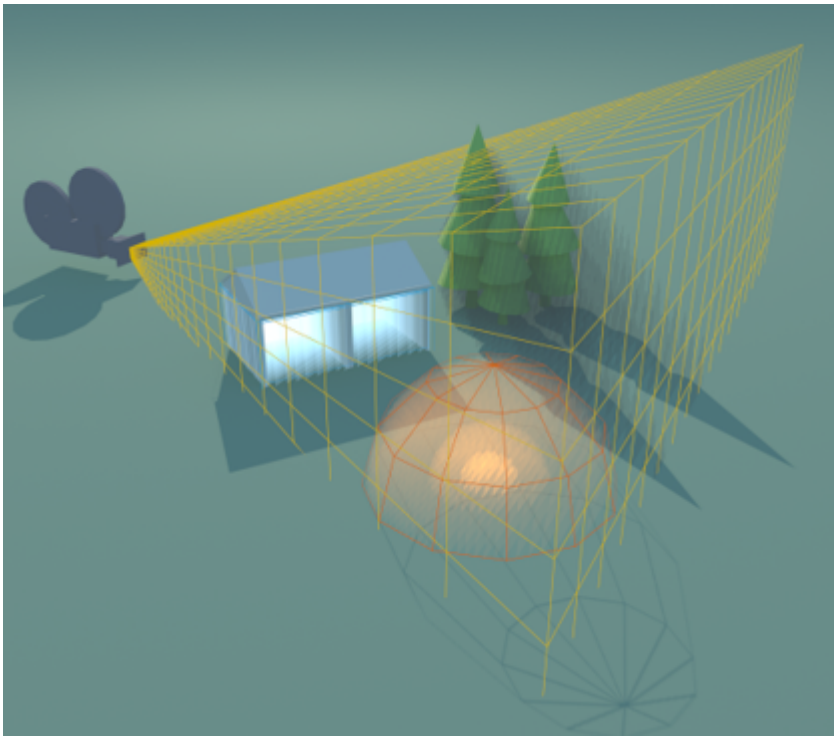


TODO: URP制作一个Demo场景，展示有无体积雾的效果区别，并打包试跑性能

HDRP中体积雾的实现的基本原理

算法过程

视锥体素化

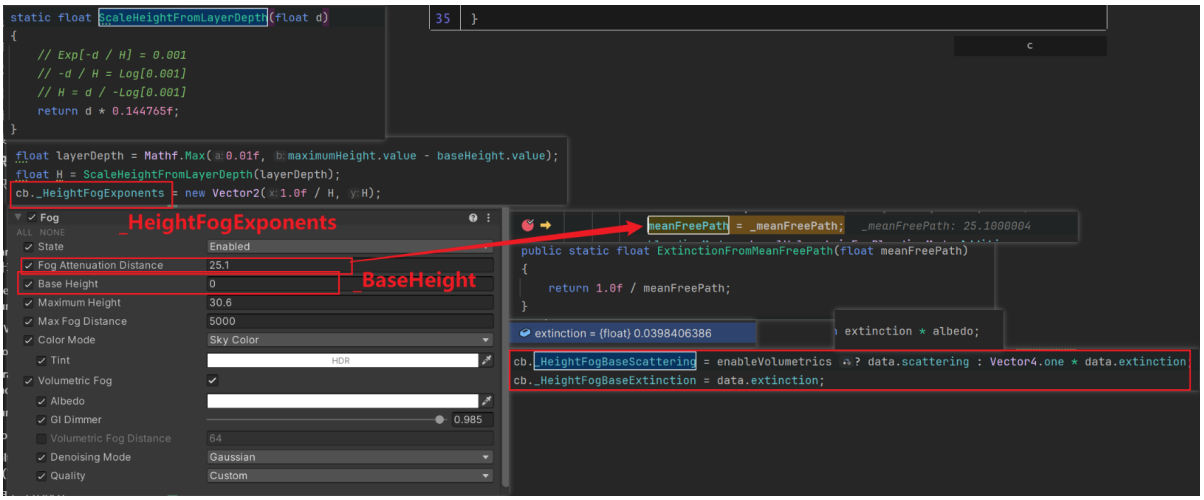


```
struct Ray
{
    float3 origrinWS;
    float3 centerDirWS;
};

for(int i = 0; i < bufferWidth; ++i)
{
    for(int j = 0; j < bufferHeight; ++j)
    {
        uint2 voxelCoord = uint2(i, j);
        uint2 voxelCoordCenter = voxelCoord + uint2(0.5, 0.5);
        // 构造射线
        Ray ray = Ray(
            CameraPosition,
            VoxelCoordToRayDir(voxelCoordCenter, near, fov, aspectRatio));
        // 填充高度雾
        FillHeightFogBuffer(voxelCoord, ray);
    }
}
```

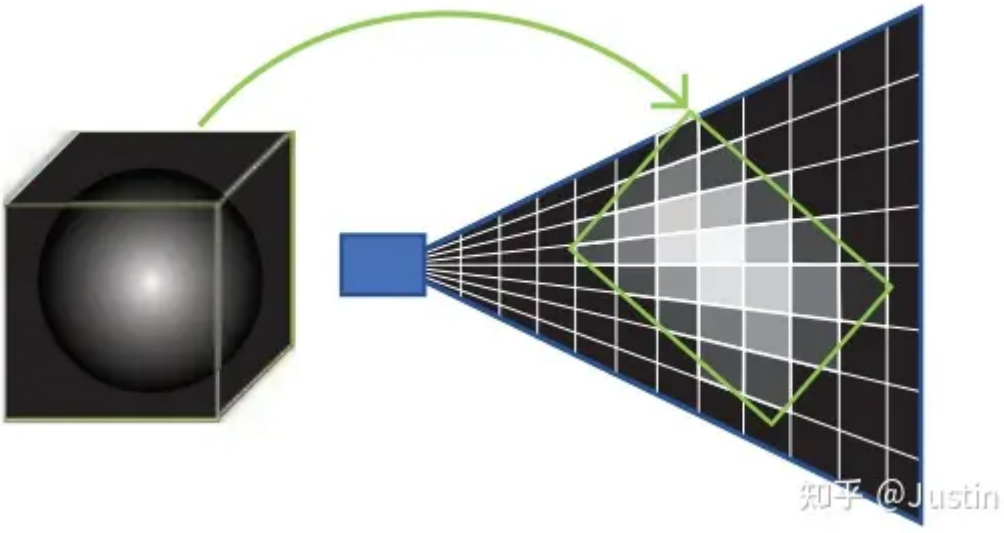
$$Ray.CurrentPosition = Ray.OriginPosition + t * Ray.Direction$$

高度雾计算

```
FillHeightFogBuffer(uint2 voxelCoord2D, Ray ray)
{
    // 获取近平面，也是光线步进的起点t
    float t0 = ConvertDepthToLinearLengthInLogSpace(0);
    // 总的深度范围是[0-1]，切了TotalSlice片，计算出每一片占据的深度
    float thicknessPerSlice = 1 / TotalSlice;

    for (int slice = 0; slice < TotalSlice; ++slice)
    {
        // 这里算3DTexture的写入坐标
        uint3 voxelCoord = uint3(voxelCoord2D, slice + thicknessPerSlice);

        // 计算当前切片方块的后切片，这里用mad指令，算出来是一个[0-1]的深度值
        float sliceBackwardPlaneDepth = slice * thicknessPerSlice +
thicknessPerSlice;
        // 将深度值转成线性值[near, far]
        float t1 =
ConvertDepthToLinearLengthInLogSpace(sliceBackwardPlaneDepth);

        float dt = t1 - t0;
        // 取格子中心点
        float t  = t0 + 0.5 * dt;
        // 计算出当前格子中心点的世界坐标，用于后面计算高度雾信息
        float3 voxelCenterWS = ray.originWS + t * ray.centerDirWS;
        float fragmentHeight   = voxelCenterWS.y;

        // 计算高度衰减系数，
        // _HeightFogBaseHeight， _HeightFogExponents 都是通过艺术家参数传进来的
        float heightMultipler =
            exp(-max(fragmentHeight - _HeightFogBaseHeight, 0) *
_HeightFogExponents.x);

        _VBufferDensity[voxelCoord] =
            float4(_HeightFogBaseScattering.xyz * heightMultiplier,
                   _HeightFogBaseExtinction * heightMultiplier);

        // to the next step
        t0 = t1;

    }
}
```



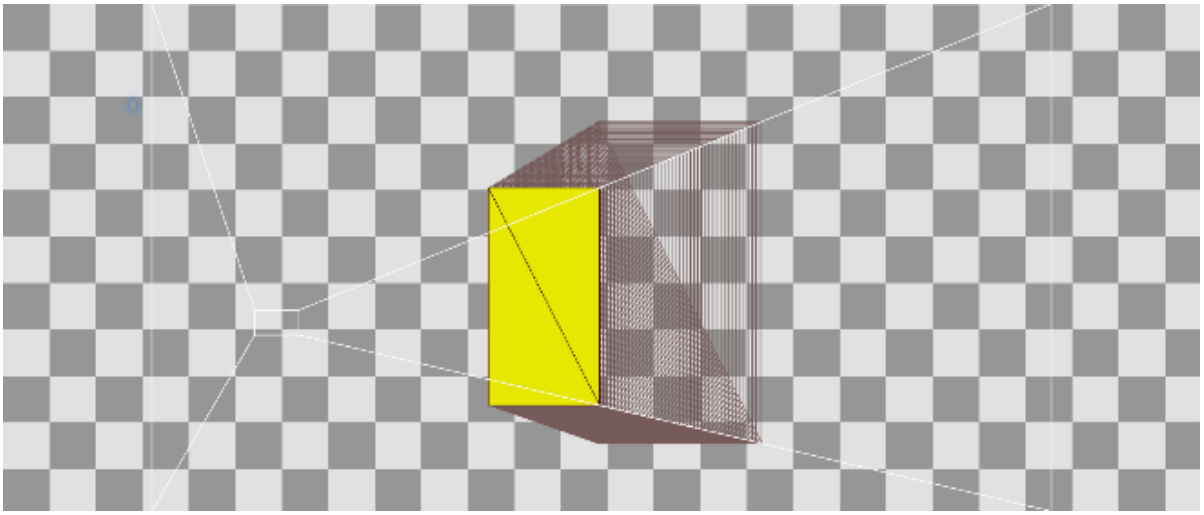
体积雾计算



如图23所示，在场景中放入一个LocalFogVolume组件，需要在体素中去填充VolumeBox包围的体积雾信息。

切片数据准备

这一步需要计算VolumeBox在视锥体体素中的范围（xy范围，和z切片范围），然后为每一个切片生成一个Quad Instance。



```
SliceDataPrepare()
{
    foreach(var volume : volumeList)
    {
        var tStart, tEnd = GetVolumeRange(volume);
        var startSlice = DistanceToSlice(tStart);
        var stopSlice = DistanceToSlice(tEnd);
        var sliceCount = stopSlice - startSlice;

        var bounds = CalculateViewBounds(volume);

        // fill parameters
        IndexCountPerQuad = 6;
        QuadInstanceCount[volumeIndex] = sliceCount;
        QuadInstanceStartSlice[volumeIndex] = startSlice;
        QuadInstanceBounds[volumeIndex] = bounds;
    }
}
```

绘制切片

```
IndexArray =
[
    0, 1, 2,
    0, 2, 3
]
/*
    0 --------------- 3
    |  .              |
    |       .         |
    |             .   |
    1 --------------- 2
*/
DrawSliceQuad()
{
    for(int instance; instance < QuadInstanceCount; ++instance)
    {
        float4 positionCS;
        // vertexID : [0 - 3]
        positionCS = VertexIDToVertex(vertexID);
        // 把坐标映射到Box的范围
        positionCS = ClampToVolumeBounds(positionCS);
        positionCS.z = SliceToDepth(QuadStartSliceIndexs[instance]);
        positionCS.w = 1.0;

        Draw(positionCS);
    }
}
```

填充体积雾数据

```
FragForEachSlice() : Target->3DTexture
{
    if (currentPixel out of VolumeBox)
    {
        discard;
    }

    albedo = sampler MaskTexture * fogAlbedo.rgb;
    extinction *= 1 / meanFreePath;
    pixelColor = float4(saturate(albedo * extinction), extinction);
}
```

体积光着色

对于每一条传播路径，对每一次切片计算Radiance结果，并累加（积分）。

对于每一个点（切片单元），需要考虑该处的散射率scattering，和BeerLambert定律。

每一个点的radiance来源有直接光照的入射和环境光，而入射进来的Irradiance，只有一部分会射向观察方向，这部分比例由相位方程来描述。

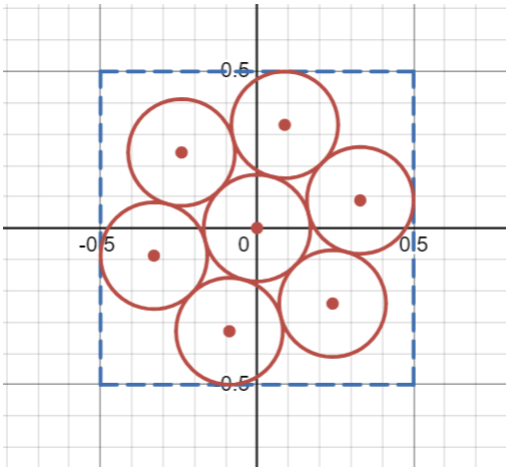
```
Shading()
{
    var totalRadiance = 0;

    for(int slice = 0; slice < TotalSlice; ++slice)
    {
        foreach(var light : VisibleLights)
        {   // GIProbeRadiance 已经是考虑了相位方程的结果
```

```
            totalRadiance += (light.color * light.shadowAtten *
light.distanceAtten* CSPhase + GIProbeRadiance) * scattering *
BeerLambertFactor;
        }
    }

    out totalRadiance;
}
```

抖动

对于每个体素，如果只用一根光线去计算，会导致结果噪点比较多。因此可以通过抖动和结合历史帧数据的方式，实现TAA的效果来进行优化。

对于每一次采样，按照下图所示去选择采样点



参考：https://en.wikipedia.org/wiki/Close-packing_of_equal_spheres

由于该分布具有一定的随机性以及均匀分布性，所以可以实现类似蓝噪声的抖动。

在管线中，通过两个Buffer交替，来实现对于历史帧的融合。

如图26所示，总共由7个采样点，因此需要7帧，每帧选用一个采样点。

```
static uint VolumetricFrameIndex(HDCamera hdCamera)
{
    // Here we do modulo 14 because we need the enable to detect a change every frame, but the accumulation is done on 7 frames (7x2=14)
    return hdCamera.GetCameraFrameCount() % 14;
}
```

之所以选择对14取余数，是因为完整一次叠加需要7帧，两个Buffer交替。

```
// [0 - 13]
int frameIndex = (int)VolumetricFrameIndex(hdCamera);
var currIdx :int = (frameIndex + 0) & 1;
var prevIdx :int = (frameIndex + 1) & 1;
```

偶数帧：

current： 0

preview：1

奇数帧：

current： 1

preview：0

FrameIndex	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Current	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
History	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

重投影

在拿历史帧数据的时候，需要用当前Voxel的世界坐标，通过上一帧VP矩阵以及上一帧相机的世界坐标重新还原uvw，然后去采样HistroyBuffer。

这里考虑Volume是静态的，因此Voxel的世界坐标不会改变，因此可以算出上一帧的深度，推算出Slice。

最后将历史帧与当前帧结果进行融合。

$$CurrentFrame = Lerp(CurrentFrame, HistroyFrame, 6/7)$$

高斯滤波

对于2D的Texture，进行3X3高斯滤波就是对九宫格进行加权（高斯函数计算权重）平均。

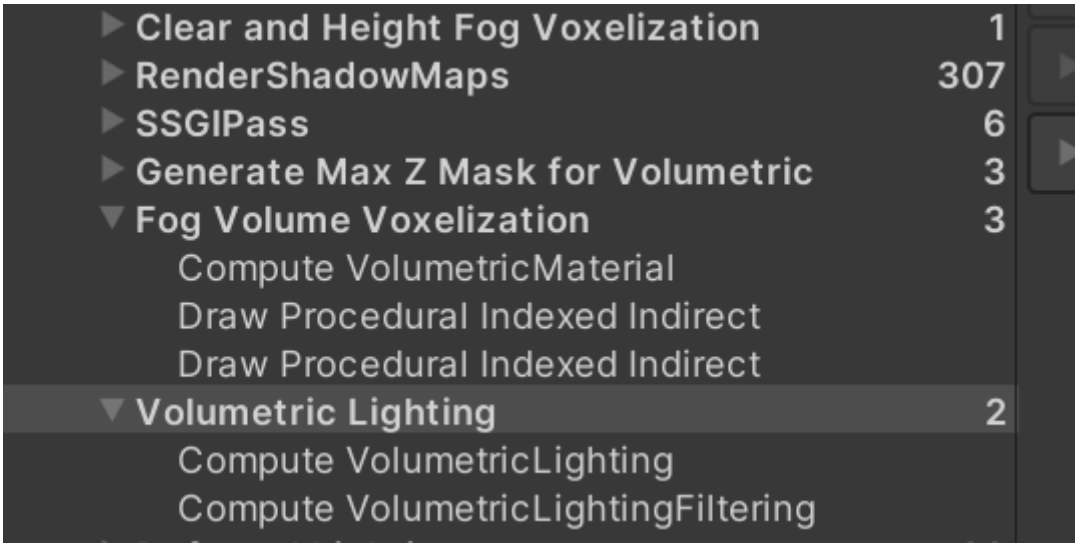
```
GussianFiltering()
{
    const int radius = 1;
    // 对每个切片做3x3高斯模糊
```

```
    for (int idx = -radius; idx <= radius; ++idx)
    {
        for (int idx2 = -radius; idx2 <= radius; ++idx2)
        {
            float4 currentValue = GetSample(voxelCoord.xy + int2(idx,
idx2));

            // Compute the weight for this tap
            float weight = Gaussian(length(int2(idx, idx2)),
GAUSSIAN_SIGMA);

            // Accumulate the value and weight
            value += currentValue * weight;
            sumW += weight;
        }

    }

    output value / sumW

}
```

管线

基于RenderGraph管线，RenderGraph是一种基于新一代的图形API去设计的管线架构，每一帧渲染的Pass都可以根据资源的依赖关系抽象出一个有向无环图。RenderGraph可以通过依赖关系以及生命周期，通过资源别名系统实现资源的复用，以及自动算好AsyncCompute的同步点。

RenderGraph主要分为三个阶段：Setup，Compile，Execute。Setup阶段可以比较乐观的去添加各种Pass，并且标记好资源的读写，Compile阶段会根据资源的依赖关系，剔除不必要的Pass和资源。Execute会最终去执行各个Pass。



在HDRP中，体积雾是通过LocalVolumetricFog组件来添加的，在管线的RecordRenderGraph方法中

```
// 添加第一个Pass，并且返回3DTexture的handle
var volumetricDensityBuffer = ClearAndHeightFogVoxelizationPass(m_RenderGraph,
hdCamera);
```

```
// 添加第二个Pass，将第一个Pass计算出来高度雾3DTexture作为输入，同时也作为输出
volumetricDensityBuffer = FogVolumeVoxelizationPass(m_RenderGraph, hdCamera,
volumetricDensityBuffer, m_VisibleVolumeBoundsBuffer);

// 添加着色Pass
var volumetricLighting = VolumetricLightingPass(m_RenderGraph, hdCamera,
prepassOutput.depthPyramidTexture, volumetricDensityBuffer,
    maxZMask, gpuLightListOutput.bigTileLightList, shadowResult);
```

总共有三个Pass，第一个Pass负责体素化并填充高度衰减散射率和总衰减系数，第二个Pass负责计算体积雾的密度数据，第三个Pass根据密度Buffer进行体积光着色。

Pass

Clear and Height Fog Voxelization

以下是在ComputeShader中 要写的buffer

```
RW_TEXTURE3D(float4, _VBufferDensity); // RGB = sqrt(scattering), A =
sqrt(extinction)
```

以下代码是3DTexture的声明

```
// 在第一个Pass添加时创建3DTexture并标记为Write
passData.densityBuffer = builder.WriteTexture(renderGraph.CreateTexture(
    new TextureDesc(s_CurrentVolumetricBufferSize.x,
s_CurrentVolumetricBufferSize.y, false, false)
{ slices = s_CurrentVolumetricBufferSize.z,
    colorFormat = GraphicsFormat.R16G16B16A16_SFloat,
    dimension = TextureDimension.Tex3D, enableRandomWrite = true, name =
"VBufferDensity" }));
```

以下代码计算3DTexture的尺寸，通过配置的屏幕划分比例和深度占比来计算

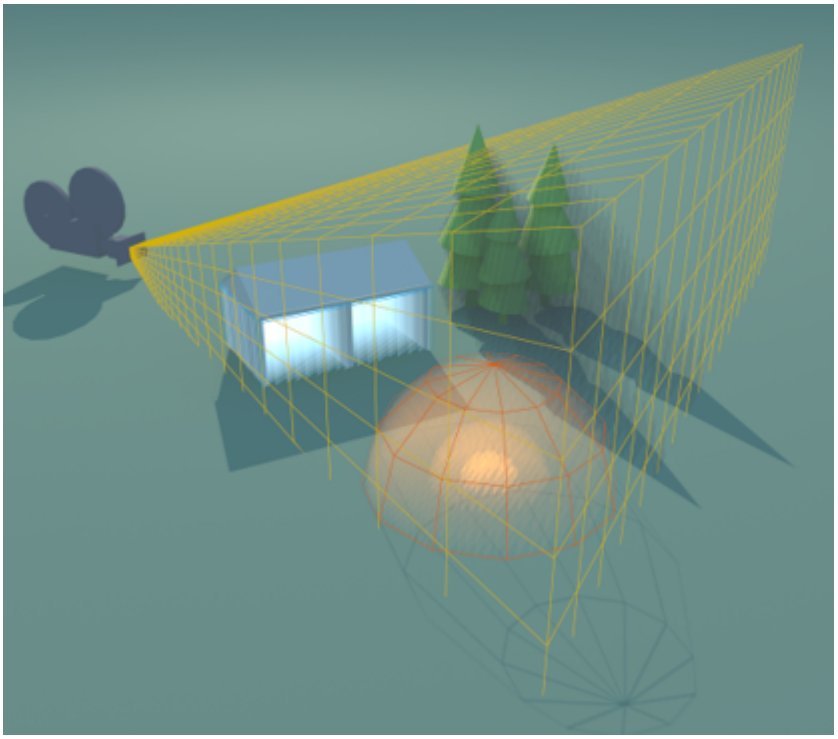
```
// Update size used to create volumetric buffers.
// 3DTexture的尺寸根据屏幕分块数和深度分块数来确定
s_CurrentVolumetricBufferSize = new
Vector3Int(Math.Max(s_CurrentVolumetricBufferSize.x,
currentParams.viewportSize.x),
    Math.Max(s_CurrentVolumetricBufferSize.y, currentParams.viewportSize.y),
    Math.Max(s_CurrentVolumetricBufferSize.z, currentParams.viewportSize.z));
```

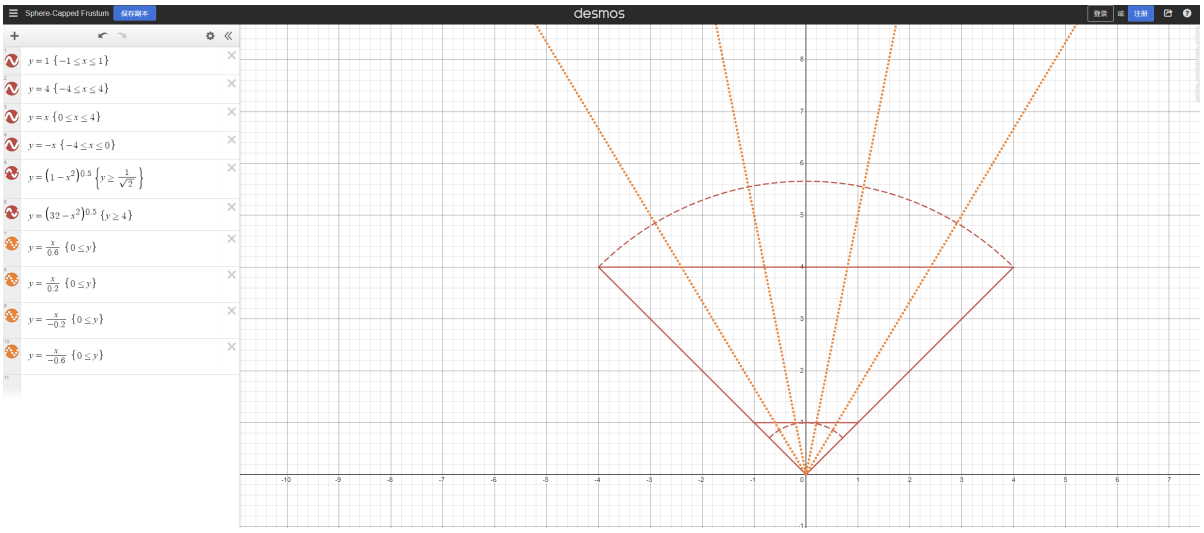
以下代码是VBuffer在shader中计算需要用的相关参数

```
struct VBufferParameters
{
    public Vector3Int viewportSize;
    public float voxelSize;
    public Vector4 depthEncodingParams;
    public Vector4 depthDecodingParams;
}
```

```
public VBufferParameters(Vector3Int viewportSize, float depthExtent, float camNear, float camFar, float camVFoV,
                         float sliceDistributionUniformity, float voxelSize)
{
    this.viewportSize = viewportSize;
    this.voxelSize = voxelSize;

    // The V-Buffer is sphere-capped, while the camera frustum is not.
    // We always start from the near plane of the camera.

    float aspectRatio = viewportSize.x / (float)viewportSize.y;
    float farPlaneHeight = 2.0f * Mathf.Tan(0.5f * camVFoV) * camFar;
    float farPlaneWidth = farPlaneHeight * aspectRatio;
    float farPlaneMaxDim = Mathf.Max(farPlaneWidth, farPlaneHeight);
    // 求出最远的距离，也就是斜边的长度，根据勾股定理
    float farPlaneDist = Mathf.Sqrt(camFar * camFar + 0.25f * farPlaneMaxDim * farPlaneMaxDim);
    // 求出最近的距离，也就是垂直的camNear
    float nearDist = camNear;
    // 通过配置的depthExtent求出深度范围
    float farDist = Math.Min(nearDist + depthExtent, farPlaneDist);
    // 将分布参数重映射
    float c = 2 - 2 * sliceDistributionUniformity; // remap [0, 1] -> [2, 0]
    c = Mathf.Max(c, 0.001f);                      // Avoid NaNs

    depthEncodingParams = ComputeLogarithmicDepthEncodingParams(nearDist, farDist, c);
    depthDecodingParams = ComputeLogarithmicDepthDecodingParams(nearDist, farDist, c);
}
```

下面的代码是用于计算变换矩阵（**是第一个Pass最关键的部分，用于视锥体体素化**）





screenSize为RT的size，x: width, y: height, z: 1/width, w: 1/height

```
aspectRatio = aspectRatio < 0 ? screenSize.x * screenSize.w : aspectRatio;
float tanHalfVertFoV = Mathf.Tan(0.5f * verticalFoV);

// Compose the matrix.

float m00 = -2.0f * screenSize.z * tanHalfVertFoV * aspectRatio;
float m11 = -2.0f * screenSize.w * tanHalfVertFoV;

// 镜头平移，  一般用于建筑摄影
float m21 = (1.0f - 2.0f * lensShift.y) * tanHalfVertFoV;
float m20 = (1.0f - 2.0f * lensShift.x) * tanHalfVertFoV * aspectRatio;

```

```
viewSpaceRasterTransform = new Matrix4x4(
    new Vector4(m00, 0.0f, 0.0f, 0.0f),
    new Vector4(0.0f, m11, 0.0f, 0.0f),
    new Vector4(m20, m21, -1.0f, 0.0f),
    new Vector4(0.0f, 0.0f, 0.0f, 1.0f));
```

```
// Remove the translation component.
var homogeneousZero = new Vector4(0, 0, 0, 1);
worldToViewMatrix.SetColumn(3, homogeneousZero);

// Flip the Z to make the coordinate system left-handed.
worldToViewMatrix.SetRow(2, -worldToViewMatrix.GetRow(2));

// Transpose for HLSL.
return Matrix4x4.Transpose(worldToViewMatrix.transpose *
viewSpaceRasterTransform);
```

以下是ComputeShader中的代码

```
// 相机的 -z
float3 F = GetViewForwardDir();
// 相机的 y
float3 U = GetViewUpDir();
// 假设线程数为512x512 则 dispatchThreadId.xy 范围[0-511]
```

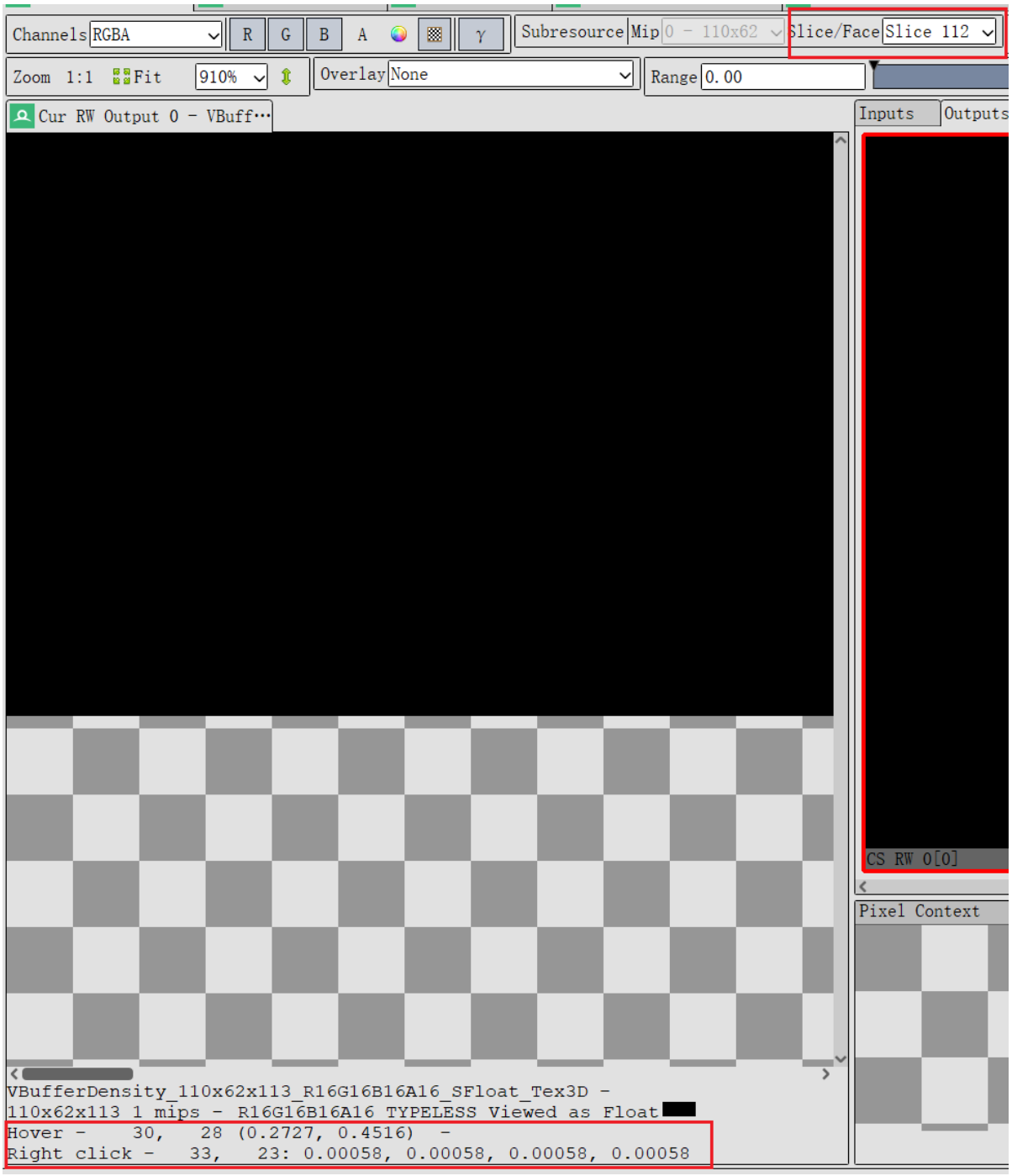
```
uint2 voxelCoord = dispatchThreadId.xy;
float2 centerCoord = voxelCoord + float2(0.5, 0.5);

// Compute a ray direction s.t. ViewSpace(rayDirWS).z = 1.
/**
 *                 [ m00, m01, m02, m03 ]
 *                 [ m10, m11, m12, m13 ]
 *    [x,y,z,w] *  [ m20, m21, m22, m23 ]
 *                 [ m30, m31, m32, m33 ]
 *
 *
 **/
float3 rayDirWS       = mul(-float4(centerCoord, 1, 1), _VBufferCoordToViewDirWS[unity_StereoEyeIndex]).xyz;
float3 rightDirWS     = cross(rayDirWS, U);
float  rcpLenRayDir   = rsqrt(dot(rayDirWS, rayDirWS));
float  rcpLenRightDir = rsqrt(dot(rightDirWS, rightDirWS));

JitteredRay ray;
ray.originWS    = GetCurrentViewPosition();
ray.centerDirWS = rayDirWS * rcpLenRayDir; // Normalize

//计算出 forward跟rayDir的cos值
float FdotD = dot(F, ray.centerDirWS);
/**
 * _VBufferUnitDepthTexelSpacing : 远平面每个纹素占据多大单位
 * FdotD * rcpLenRayDir :
 * 因为单位格子下，越靠近屏幕中心，格子对应的立体角越大。
 **/
float unitDistFaceSize = _VBufferUnitDepthTexelSpacing * FdotD * rcpLenRayDir;
// 越靠近视野中心，导数越大
ray.xDirDerivWS = rightDirWS * (rcpLenRightDir * unitDistFaceSize); // Normalize & rescale
ray.yDirDerivWS = cross(ray.xDirDerivWS, ray.centerDirWS); // Will have the length of 'unitDistFaceSize' by construction
ray.jitterDirWS = ray.centerDirWS; // TODO ???

// 处理 Camera-Relative Rendering
ApplyCameraRelativeXR(ray.originWS);

FillVolumetricDensityBuffer(voxelCoord, ray);
```

以下是填充高度雾的shader代码

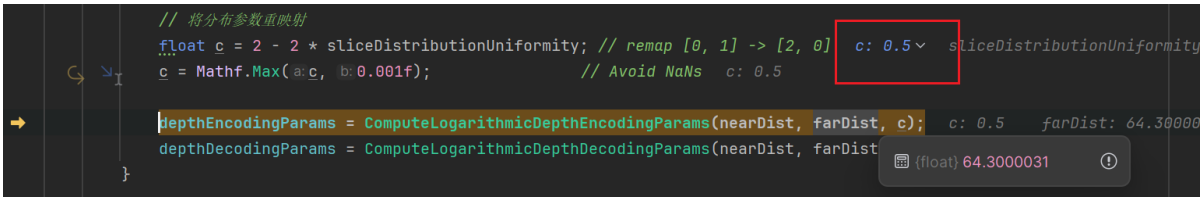
```
// 将深度0映射到近平面
float t0 = DecodeLogarithmicDepthGeneralized(0, _VBufferDistanceDecodingParams);
// ？？？？ 这里貌似没有进行Log-encoded啊 ？？？？？
float de = _VBufferRcpSliceCount; // Log-encoded distance between slices

// _VBufferSliceCount 在RecordAndExecute之前，已经通过UpdateGlobalConstant传进来了
for (uint slice = 0; slice < _VBufferSliceCount; slice++)
{
```

```
    uint3 voxelCoord = uint3(voxelCoord2D, slice + _VBufferSliceCount * unity_StereoEyeIndex);

    float e1 = slice * de + de; // (slice + 1) / sliceCount
    float t1 = DecodeLogarithmicDepthGeneralized(e1, _VBufferDistanceDecodingParams);
    float dt = t1 - t0;
    float t  = t0 + 0.5 * dt;

    float3 voxelCenterWS = ray.originWS + t * ray.centerDirWS;

    // TODO: the fog value at the center is likely different from the average value across the voxel.
    // Compute the average value.
    float fragmentHeight   = voxelCenterWS.y;
    float heightMultiplier = ComputeHeightFogMultiplier(fragmentHeight, _HeightFogBaseHeight, _HeightFogExponents);

    // Start by sampling the height fog.
    float3 voxelScattering = _HeightFogBaseScattering.xyz * heightMultiplier;
    float  voxelExtinction = _HeightFogBaseExtinction * heightMultiplier;

    _VBufferDensity[voxelCoord] = float4(voxelScattering, voxelExtinction);

    t0 = t1;
}
```

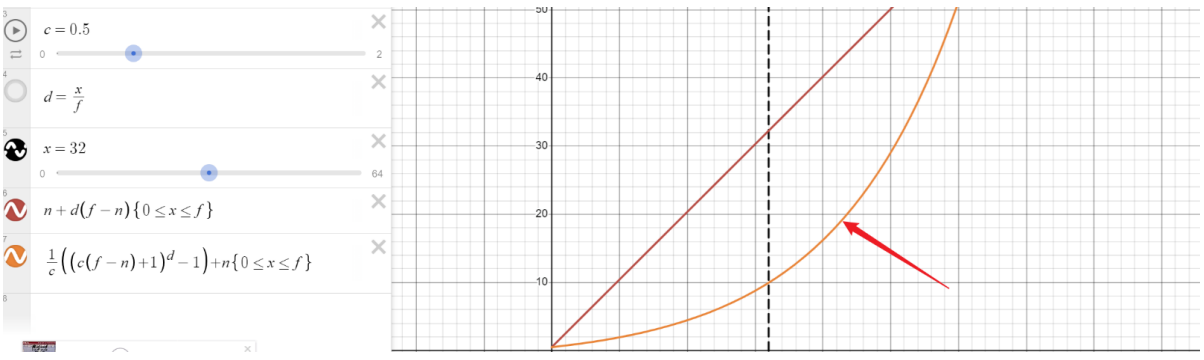
计算结果如下



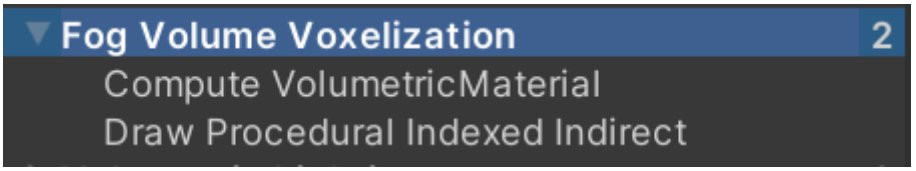
以下是深度切片分布系数



深度按Log分布比例是0.5，图像如下



Fog Volume Voxelization



Compute VolumetricMaterial

```
Runtime/Material/VolumetricMaterial/VolumetricMaterial.compute
```

以下是ComputeShader中的代码

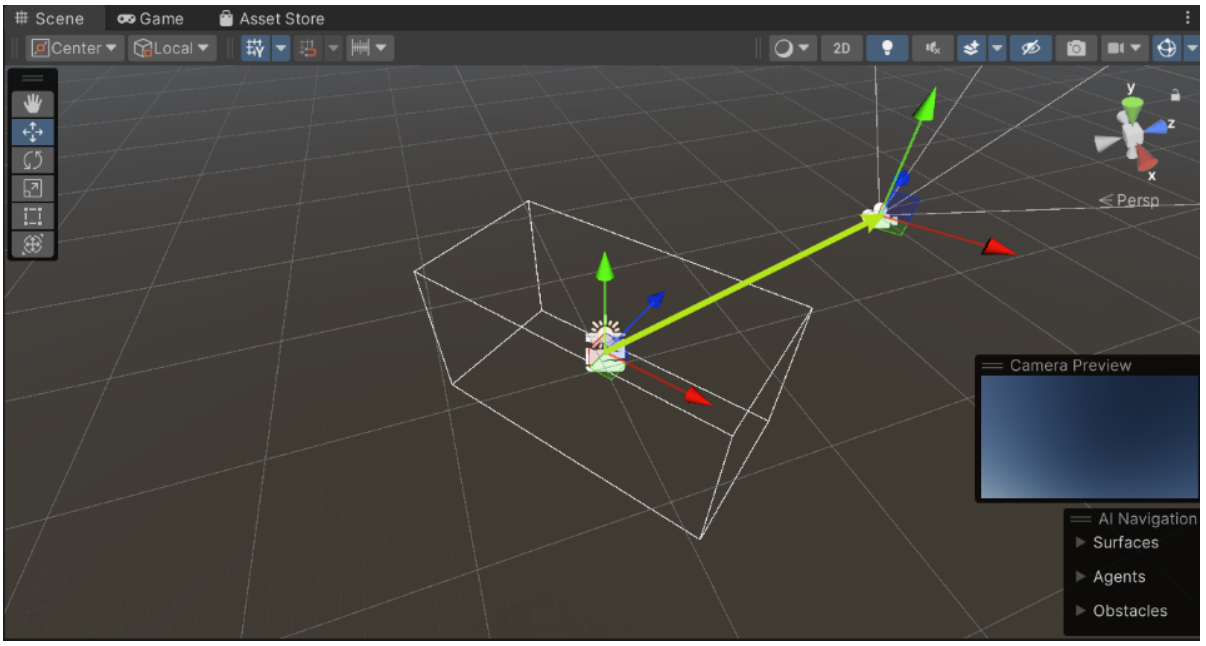
```
// Generate the compute buffer to dispatch the indirect draw
// 每个线程组分配32个线程，线程组数量为32个volume共用一个线程组，相当于一个线程对应一个Volume
[numthreads(32, 1, 1)]
void ComputeVolumetricMaterialRenderingParameters(uint3 dispatchThreadId : SV_DispatchThreadID)
{
    // _VolumeCount 为Volume的总个数，_ViewCount 跟XR相关
    if (dispatchThreadId.x >= _VolumeCount * _ViewCount)
        return;
    // 计算volume下标
    uint volumeIndex = dispatchThreadId.x % _VolumeCount;
    // 因为有考虑到ViewCount>1的情况，因此volumeIndex跟writeIndex分开计算
    uint volumeWriteIndex = dispatchThreadId.x;
    uint viewIndex = dispatchThreadId.x / _VolumeCount;

#if defined(UNITY_STEREO_INSTANCING_ENABLED)
    unity_StereoEyeIndex = viewIndex;
#endif

    // Sort cube vertices for slicing in vertex
#if USE_VERTEX_CUBE_SLICING
    ComputeCubeVerticesOrder(volumeIndex);
#endif

    OrientedBBox obb = _VolumeBounds[volumeIndex]; // the OBB is in world space
    float3 obbExtents = float3(obb.extentX, obb.extentY, obb.extentZ);

    float2 minPositionVS = 1;
    float2 maxPositionVS = -1;
    // DistanceDistanceToOBB 计算过程见下一个代码段
    // 计算出相机到包围盒的最短距离
    float cameraDistanceToOBB = DistanceDistanceToOBB(GetCameraPositionWS(), obb);

    // 小于0说明相机在Volume内部
    if (cameraDistanceToOBB <= 0)
    {
        minPositionVS = -1;
```

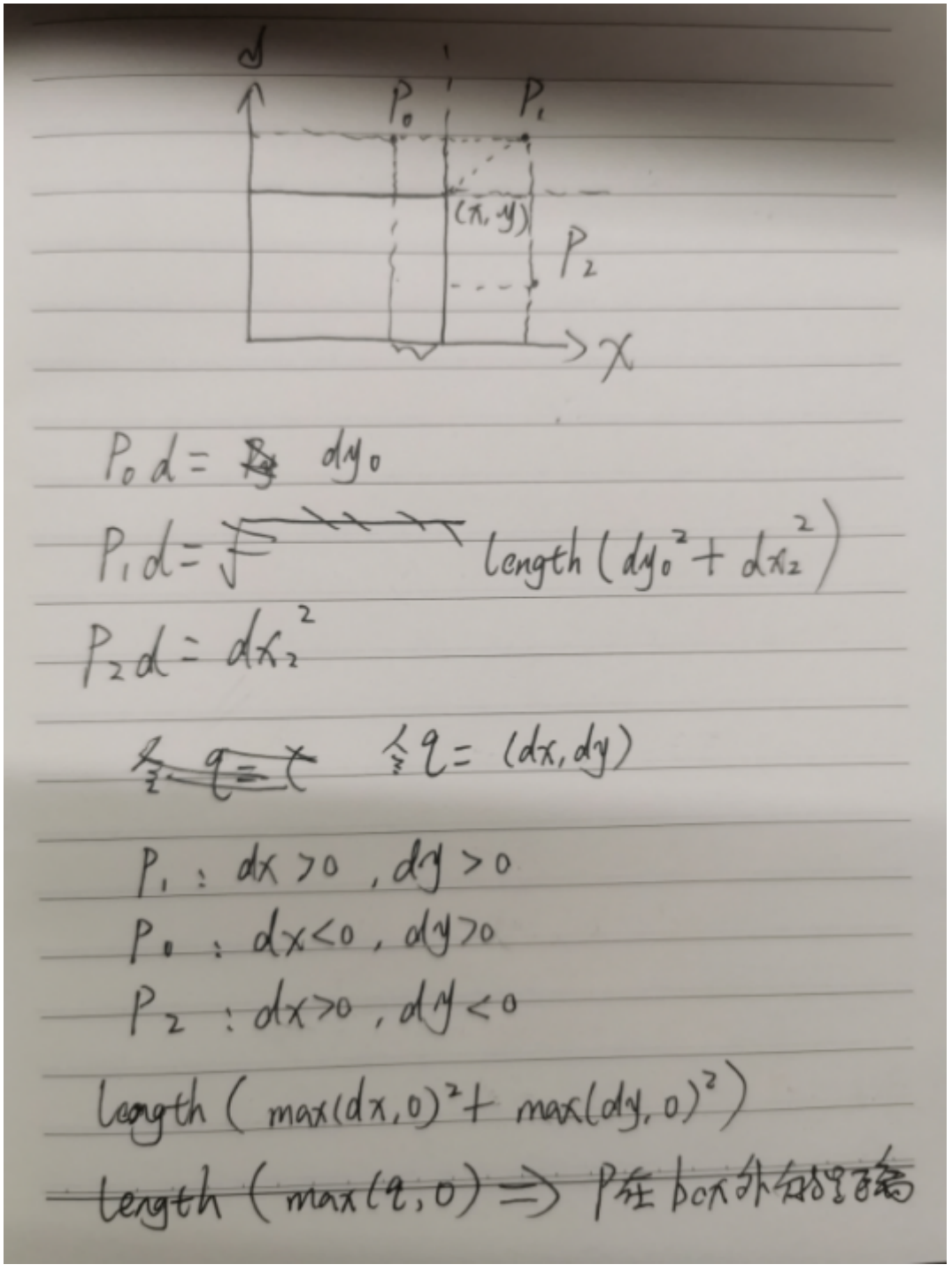
```
        maxPositionVS = 1;
    }

    // Find the min and max distance value from vertices
    // 这个计算Volume在世界空间中的最小顶点， 但是没用到 = =!
    float3 minPositionRWS = obb.center - obbExtents;
    float maxVertexDepth;
    int i;
    for (i = 0; i < 8; i++)
    {   // ComputeCubeVertexPositionRWS方法的计算见下面代码段
        // 得出由世界原点指向Volume各个顶点的向量（世界空间下）
        float3 position = ComputeCubeVertexPositionRWS(obb, minPositionRWS,
vertexMask[i]);
        // 算出投影到CameraForward的距离
        float depth = DepthDistance(position);
        float distance = length(position);
        maxVertexDepth = max(maxVertexDepth, distance);
        // 如果是相机在Volume外面
        if (cameraDistanceToOBB > 0)
        {
            float4 positionCS = TransformWorldToHClip(position);

            // clamp positionCS inside the view in case the point is behind
the camera
            if (positionCS.w < 0)
            {
                minPositionVS = -1;
                maxPositionVS = 1;
            }
            else
            {
                positionCS.xy /= positionCS.w;
                minPositionVS = min(positionCS.xy, minPositionVS);
                maxPositionVS = max(positionCS.xy, maxPositionVS);
            }
        }
    }

    /// 算出Volume投影到屏幕的NDC坐标范围
    minPositionVS = clamp(minPositionVS, -1, 1);
    maxPositionVS = clamp(maxPositionVS, -1, 1);

    // Compute min distance using the
    float vBufferNearplane = DecodeLogarithmicDepthGeneralized(0,
_VBufferDistanceDecodingParams);
    float minBoxDistance = max(vBufferNearplane, cameraDistanceToOBB);
    int startSliceIndex = clamp(DistanceToSlice(minBoxDistance), 0,
int(_MaxSliceCount));
    int stopSliceIndex = DistanceToSlice(maxVertexDepth);
    uint sliceCount = clamp(stopSliceIndex - startSliceIndex, 0,
int(_MaxSliceCount) - startSliceIndex);

    _IndirectBufferArguments[volumeWriteIndex * 5 + 0] = 6; //
IndexCountPerInstance
    _IndirectBufferArguments[volumeWriteIndex * 5 + 1] = sliceCount; //
InstanceCount
```

```
    _IndirectBufferArguments[volumeWriteIndex * 5 + 2] = 0; // StartIndexLocation
    _IndirectBufferArguments[volumeWriteIndex * 5 + 3] = 0; // BaseVertexLocation
    _IndirectBufferArguments[volumeWriteIndex * 5 + 4] = 0; // StartInstanceLocation

    _VolumetricMaterialData[volumeWriteIndex].sliceCount = sliceCount;
    _VolumetricMaterialData[volumeWriteIndex].startSliceIndex = startSliceIndex;
    _VolumetricMaterialData[volumeWriteIndex].viewSpaceBounds = float4(minPositionVS, maxPositionVS - minPositionVS);
}
```

以下代码是DistanceDistanceToOBB的计算过程

```
float DistanceDistanceToOBB(float3 p, OrientedBBox obb)
{   // offset为由volume中心指向相机原点的向量， p为相机的世界坐标
    float3 offset = p - obb.center;
    // volume的forward向量
    float3 boxForward = normalize(cross(obb.right, obb.up));
    // 把offset投影到volume的正交基，相当于转到了VolumeOBB的空间坐标系，extent则变成了Volume空间下的坐标
    // 因此得到的axisAlignedPoint相当于相机在VolumeOBB坐标系下的位置
    float3 axisAlignedPoint = float3(dot(offset, normalize(obb.right)), dot(offset, normalize(obb.up)), dot(offset, boxForward));
    // 最后计算点到AABB的距离，就可以得出相机跟Volume的距离了
    return DistanceToAABB(axisAlignedPoint, float3(obb.extentX, obb.extentY, obb.extentZ));
}

// 这里其实算的是有向距离，也就是在box内是负的
float DistanceToAABB(float3 p, float3 b)
{   // 对于b，描述的是box的size，而p可能位于8个象限的任意一个，而这里算距离，只需要考虑第一象限，因此取绝对值
    // 减去b，可以得出q的分量存的是各个轴上的距离
    float3 q = abs(p) - b;
    // 这一步的计算原理见下图，
    // 当p在box外，则结果为length(max(q, 0.0))
    // 当p在box内，则length(max(q, 0.0)) = 0， 结果是min(max(q.x, max(q.y, q.z)), 0.0)
    return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);
}
```





$P_0 d = dy_0$

$P_1 d = \sqrt{\quad}$ length $(dy_0^2 + dx_2^2)$

$P_2 d = dx_2^2$

~~$q = t$~~ $\Leftrightarrow q = (dx, dy)$

P_1 : $dx > 0$, $dy > 0$

P_0 : $dx < 0$, $dy > 0$

P_2 : $dx > 0$, $dy < 0$

length $(\max(dx, 0)^2 + \max(dy, 0)^2)$

~~length $(\max(q, 0)$~~ $\Rightarrow$ ~~P在box外的距离~~

以下代码是ComputeCubeVertexPositionRWS的计算过程

```
float3 ComputeCubeVertexPositionRWS(OrientedBBox obb, float3 minPositionRWS,
float3 vertexMask)
{   // 构造正交基
    float3x3 obbFrame   = float3x3(obb.right, obb.up, cross(obb.right,
obb.up));
    float3   obbExtents = float3(obb.extentX, obb.extentY, obb.extentZ);
    //// (vertexMask * 2 - 1) * obbExtents 为volume空间下，各个顶点的向量
    //// mul((vertexMask * 2 - 1) * obbExtents, (obbFrame))  把向量转换到世界空
间，类似TBN
    //// 得出由世界原点指向Volume各个顶点的向量（世界空间下）
    return mul((vertexMask * 2 - 1) * obbExtents, (obbFrame)) + obb.center;
}
```

最终这个Pass的输出是

```
_IndirectBufferArguments[volumeWriteIndex * 5 + 0] = 6; //
IndexCountPerInstance
_IndirectBufferArguments[volumeWriteIndex * 5 + 1] = sliceCount; //
InstanceCount
_IndirectBufferArguments[volumeWriteIndex * 5 + 2] = 0; // StartIndexLocation
_IndirectBufferArguments[volumeWriteIndex * 5 + 3] = 0; // BaseVertexLocation
_IndirectBufferArguments[volumeWriteIndex * 5 + 4] = 0; //
StartInstanceLocation
// 当前Volume在视锥体体素中占据的切片数
_VolumetricMaterialData[volumeWriteIndex].sliceCount = sliceCount;
_VolumetricMaterialData[volumeWriteIndex].startSliceIndex = startSliceIndex;
_VolumetricMaterialData[volumeWriteIndex].viewSpaceBounds =
float4(minPositionVS, maxPositionVS - minPositionVS);
```

下图是DX关于IndirectBufferArgs的内存布局规定

Bytes	Description
0:3	IndexCountPerInstance
4:7	InstanceCount
8:11	StartIndexLocation
12:15	BaseVertexLocation
16:19	StartInstanceLocation
20:35	Padding

DrawProcedural Indexed Indirect

通过DrawProceduralIndirect绘制，有2种着色模式，以及5种Blend模式，不同的Blend模式选择不同的PassIndex

Blend

Overwrite

```
Blend One Zero, One Zero
BlendOp Add, Add
```

Additive

```
Blend One One, One One
BlendOp Add, Add
```

Multiply

```
Blend DstColor Zero, DstAlpha Zero
BlendOp Add, Add
```

（该模式下shader会做特殊处理）

```
outColor = max(0, lerp(float4(1.0, 1.0, 1.0, 1.0), float4(saturate(albedo * extinction), extinction), fade.xxxx));
```

Min

```
Blend One One, One One
BlendOp Min, Min
```

Max

```
Blend One One, One One
BlendOp Max, Max
```

以下代码是DrawProceduralIndirect的顶点索引定义

```
m_VolumetricMaterialIndexBuffer = new GraphicsBuffer(GraphicsBuff
m_VolumetricFogSortKeys = new NativeArray<uint>(asset.currentPlat
m_VolumetricFogSortKeysTemp = new NativeArray<uint>(asset.current
// Index buffer for triangle fan with max 6 vertices
m_VolumetricMaterialIndexBuffer.SetData(new List<uint>{
    0, 1, 2,
    0, 2, 3,
    0, 3, 4,
    0, 4, 5
});
```

以下是绘制的代码

```
///  triangleFanIndexBuffer
```

```
///  0, 1, 2,
///  0, 2, 3,
///  0, 3, 4,
///  0, 4, 5
/*
    0 --------------- 3
    |  .              |
    |       .         |
    |             .   |
    1 --------------- 2
*/

ctx.cmd.DrawProceduralIndirect(
    // 绘制的顶点Index
    data.triangleFanIndexBuffer,
    // M矩阵
    volume.transform.localToWorldMatrix,
    // 材质及Pass
    material, passIndex,
    // 拓扑
    MeshTopology.Triangles,
    // 绘制参数，  用于控制顶点索引数，Mesh示例数，以及数据偏移
    data.indirectArgumentBuffer,
    k_VolumetricMaterialIndirectArgumentByteSize * volumeIndex + viewOffset,
    props
);
```

以下是Shader中Vertext部分

Packages/com.unity.render-pipelines.high-definition@14.0.9/Editor/Material/FogVolume/ShaderGraph/ShaderPassVoxelize.hlsl

```
struct VertexToFragment
{
    float4 positionCS : SV_POSITION;
    float3 viewDirectionWS : TEXCOORD0;
    float3 positionOS : TEXCOORD1;
    uint depthSlice : SV_RenderTargetArrayIndex;
};

// instanceId 根据 IndirectArgs的 4:7 Bytes得到
// vertexId 根据 IndirectArgs的 0:3 Bytes得到
VertexToFragment Vert(uint instanceId : INSTANCEID_SEMANTIC, uint vertexId : VERTEXID_SEMANTIC)
{
    VertexToFragment output;

#if defined(UNITY_STEREO_INSTANCING_ENABLED)
    unity_StereoEyeIndex = _ViewIndex;
#endif
    /*
        _VolumetricMaterialData[volumeWriteIndex].sliceCount = sliceCount;
        _VolumetricMaterialData[volumeWriteIndex].startSliceIndex = startSliceIndex;
        _VolumetricMaterialData[volumeWriteIndex].viewSpaceBounds =
```

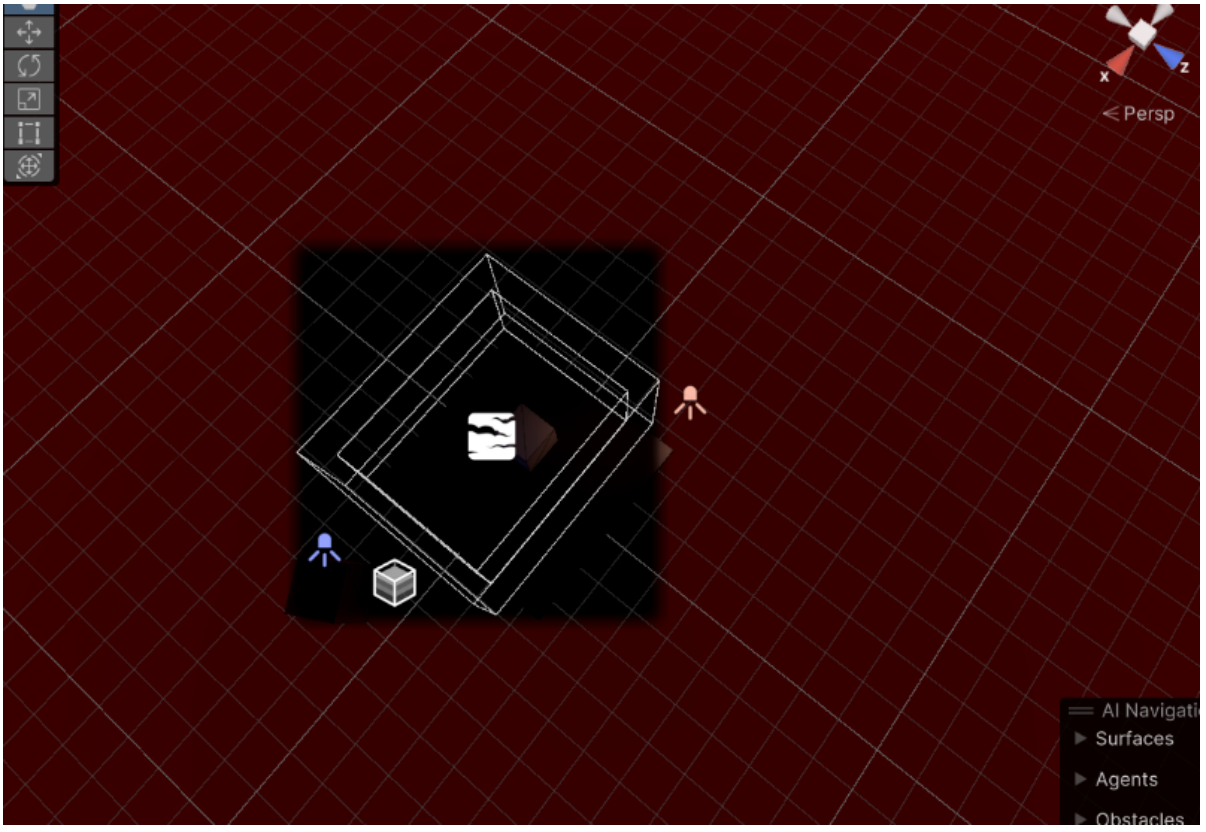
```
                float4(minPositionVS, maxPositionVS - minPositionVS);
    */
    // _VolumeMaterialDataIndex :  volumeIndex
    // 这里拿到的sliceCount就是Volume的Box占据视锥体中切片的个数
    uint sliceCount =
_VolumetricMaterialData[_VolumeMaterialDataIndex].sliceCount;
    // 拿到起始的Slice下标
    uint sliceStartIndex =
_VolumetricMaterialData[_VolumeMaterialDataIndex].startSliceIndex;
    // 每一个切片是一个instance，这里通过instanceId算出当前切片的index
    uint sliceIndex = sliceStartIndex + (instanceId % sliceCount);
    // 算出RT的z坐标（切片坐标），如果是Double Views，则RT的Slice也是Double
    output.depthSlice = sliceIndex + _ViewIndex * _VBufferSliceCount;
    // 算出当前slice的深度值
    float sliceDepth = VBufferDistanceToSliceIndex(sliceIndex);

#if USE_VERTEX_CUBE_SLICING

    float3 cameraForward = -UNITY_MATRIX_V[2].xyz;
    float3 sliceCubeVertexPosition =
ComputeCubeSliceVertexPositionRWS(cameraForward, sliceDepth, vertexId);
    output.positionCS =
TransformWorldToHClip(float4(sliceCubeVertexPosition, 1.0));
    output.viewDirectionWS = GetWorldSpaceViewDir(sliceCubeVertexPosition);
    output.positionOS = mul(UNITY_MATRIX_I_M, sliceCubeVertexPosition);

#else
    // 这一步是根据vertexId去构造顶点，具体代码见下
    // 最终得到 (0, 0),(0, 1),(1, 0),(1, 1)
    output.positionCS = GetQuadVertexPosition(vertexId);
    // 上面的得到的是一个占据屏幕四分之一的Quad（DX会显示在右下角），下面会根据Volume占
据屏幕的范围做Clamp
    output.positionCS.xy = output.positionCS.xy *
        // viewSpaceBounds: float4(minPositionVS, maxPositionVS -
minPositionVS);
        // x: NDC下x的起点
        // y: NDC下y的起点
        // z: NDC下x方向的长度
        // w: NDC下y方向的长度
        _VolumetricMaterialData[_VolumeMaterialDataIndex].viewSpaceBounds.zw
    + _VolumetricMaterialData[_VolumeMaterialDataIndex].viewSpaceBounds.xy;
    output.positionCS.z = EyeDepthToLinear(sliceDepth, _ZBufferParams);
    output.positionCS.w = 1;

    float3 positionWS = ComputeWorldSpacePosition(output.positionCS,
_IsObliqueProjectionMatrix ? _CameraInverseViewProjection_NO :
UNITY_MATRIX_I_VP);
    output.viewDirectionWS = GetWorldSpaceViewDir(positionWS);

    // Calculate object space position
    output.positionOS = mul(UNITY_MATRIX_I_M, float4(positionWS, 1)).xyz;
#endif // USE_VERTEX_CUBE_SLICING

    return output;
}
```

以下代码是通过vertexId构造顶点

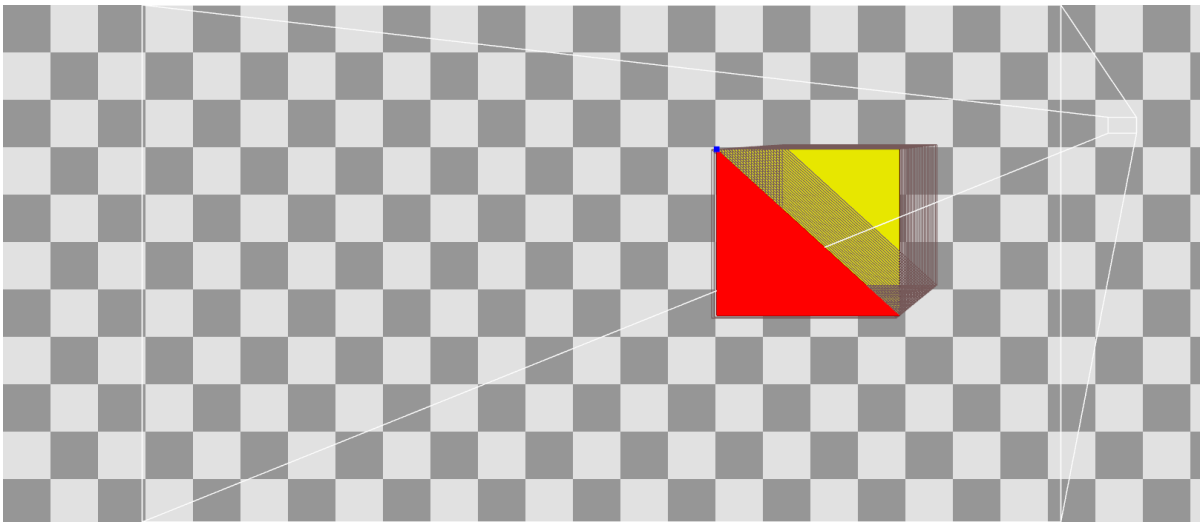
```
// 0 - 0,1
// 1 - 0,0
// 2 - 1,0
// 3 - 1,1
// index 是
///  0, 1, 2,
///  0, 2, 3,
// 因此vertexID的范围是 0 ~ 3
float4 GetQuadVertexPosition(uint vertexID, float z = UNITY_NEAR_CLIP_VALUE)
{   // 0 - 0
    // 1 - 0
    // 2 - 1
    // 3 - 1
    uint topBit = vertexID >> 1;
    // 0 - 0,
    // 1 - 1,
    // 2 - 0,
    // 3 - 1
    uint botBit = (vertexID & 1);
    float x = topBit;
    float y = 1 - (topBit + botBit) & 1; // produces 1 for indices 0,3 and 0 for 1,2
    float4 pos = float4(x, y, z, 1.0);
#ifdef UNITY_PRETRANSFORM_TO_DISPLAY_ORIENTATION
    pos = ApplyPretransformRotation(pos);
#endif
    return pos;
}
```

下图是GetQuadVertexPosition得出来的结果展示

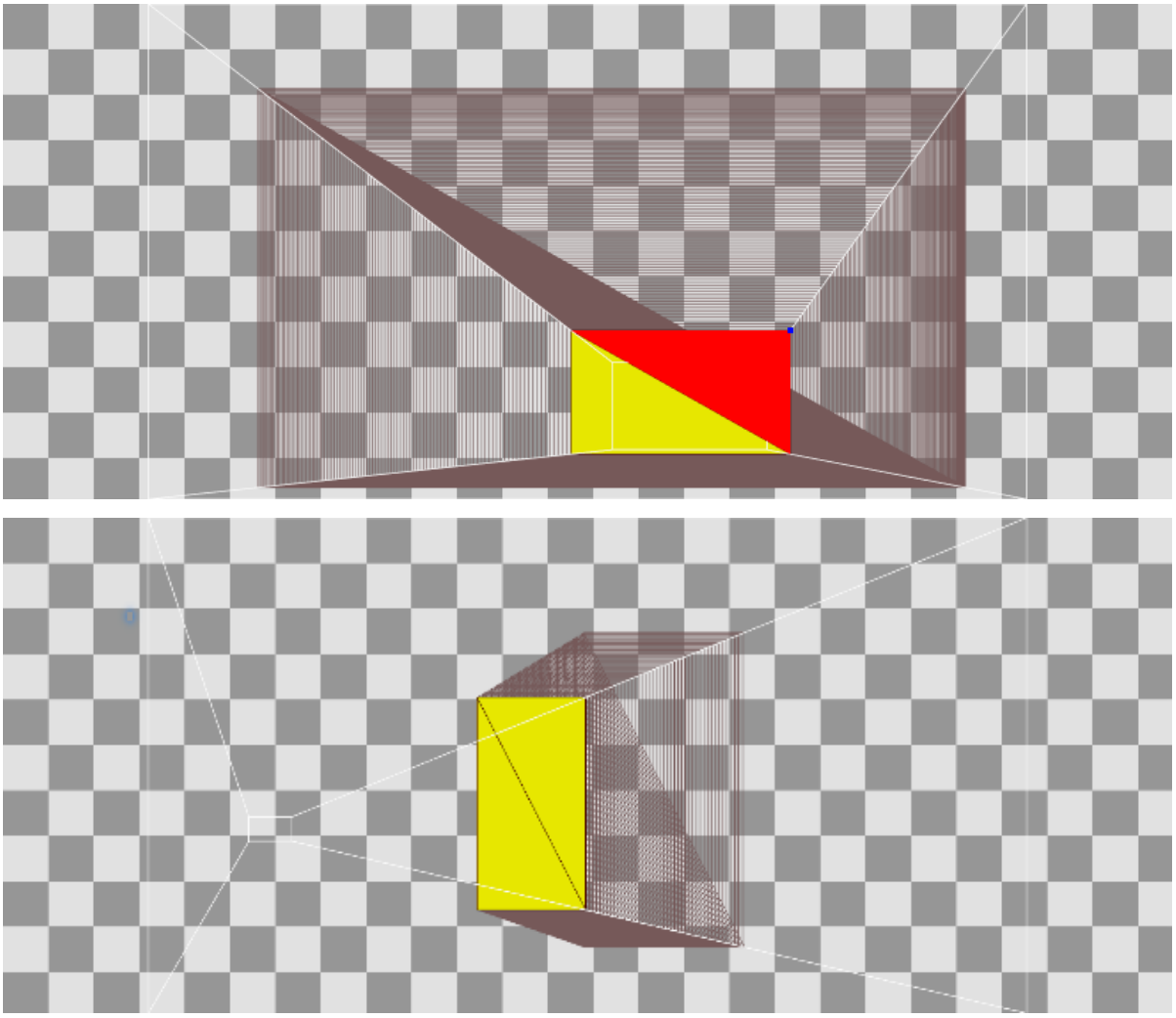
经过Clamp，Vert着色器最终输出的CS坐标正好为Volume在屏幕的Box范围，且每一个切片一个 Quad Instance 如下图



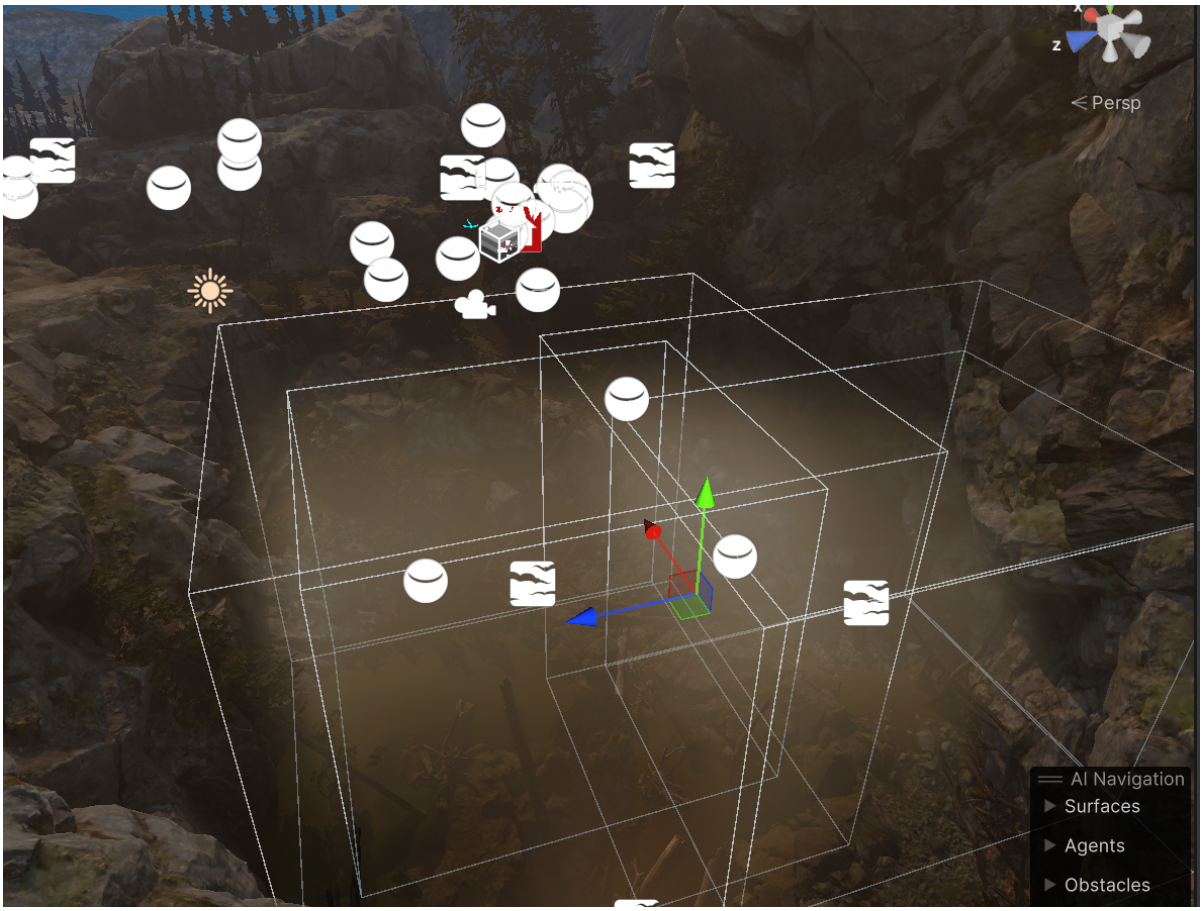
下图展示实际绘制的切片（相机拉远时）



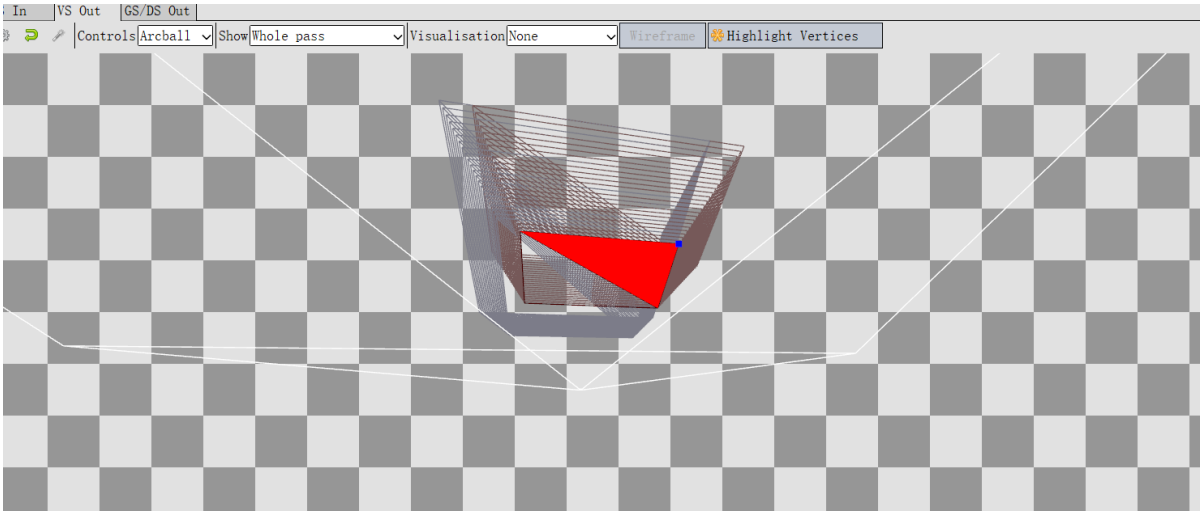
相机拉近时



如果屏幕中可见的Volume有两个，如下图



则需要调用两次cmd.DrawProceduralIndirect



下一步开始进行着色

TextureMask

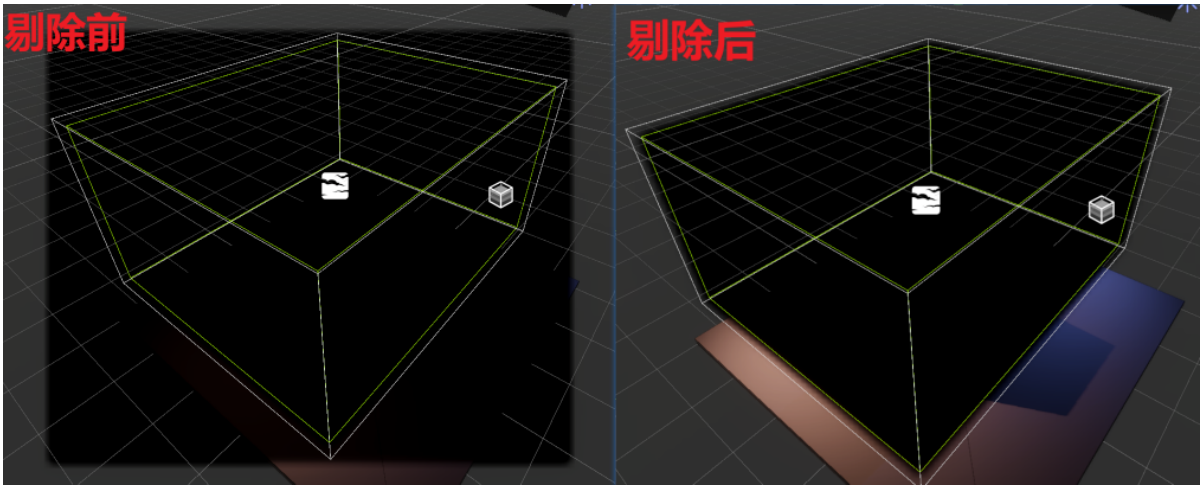
Runtime/RenderPipelineResources/ShaderGraph/DefaultFogVolume.shadergraph

```
void Frag(VertexToFragment v2f, out float4 outColor : SV_Target0)
{

```

```
    // Setup VR storeo eye index manually because we use the SV_RenderTargetArrayIndex semantic which conflicts with XR macros
#if defined(UNITY_SINGLE_PASS_STEREO)
    unity_StereoEyeIndex = _ViewIndex;
#endif

    float3 albedo;
    float extinction;
    // 把3DTexture的z坐标转成切片深度值
    float sliceDepth = VBufferDistanceToSliceIndex(v2f.depthSlice % _VBufferSliceCount);
    float3 cameraForward = -UNITY_MATRIX_V[2].xyz;
    float sliceDistance = sliceDepth;// / dot(-v2f.viewDirectionWS, cameraForward);

    // Compute voxel center position and test against volume OBB
    float3 raycenterDirWS = normalize(-v2f.viewDirectionWS); // Normalize
    float3 rayoriginWS    = GetCurrentViewPosition();
    // 计算出当前切片的体素中心点世界坐标
    float3 voxelCenterWS = rayoriginWS + sliceDistance * raycenterDirWS;
    // 构造正交基
    float3x3 obbFrame = float3x3(_VolumetricMaterialObbRight.xyz, _VolumetricMaterialObbUp.xyz, cross(_VolumetricMaterialObbRight.xyz, _VolumetricMaterialObbUp.xyz));
    // voxelCenterWS - _VolumetricMaterialObbCenter.xyz 相当于由box中心指向顶点的世界向量
    // 下一步运算要把世界向量转成Box空间下的向量，由于obbFrame是世界空间下的正交基，因此需要求逆矩阵
    // 又因为obbFrame是正交矩阵，逆等于它的转置
    float3 voxelCenterBS = mul(voxelCenterWS - _VolumetricMaterialObbCenter.xyz, transpose(obbFrame));
    // 进行归一化
    float3 voxelCenterCS = (voxelCenterBS * rcp(_VolumetricMaterialObbExtents.xyz));

    // Still need to clip pixels outside of the box because of the froxel buffer shape
    // 如果不进行剔除，则渲染出来的Quad会超出VolumeBox的范围
    bool overlap = Max3(abs(voxelCenterCS.x), abs(voxelCenterCS.y), abs(voxelCenterCS.z)) <= 1;
    if (!overlap)
        clip(-1);

    FragInputs fragInputs = BuildFragInputs(v2f, voxelCenterBS, voxelCenterCS);
    // 采样Mask贴图
    GetVolumeData(fragInputs, v2f.viewDirectionWS, albedo, extinction);

    // Accumulate volume parameters
    // 根据LocalVolumeFog的衰减距离累积衰减值
    extinction *= _VolumetricMaterialExtinction;
    // LocalVolumeFog的反照率
    albedo *= _VolumetricMaterialAlbedo.rgb;

    float3 voxelCenterNDC = saturate(voxelCenterCS * 0.5 + 0.5);
    // 计算衰减
```

```
    float fade = ComputeFadeFactor(voxelCenterNDC, sliceDistance);

    // When multiplying fog, we need to handle specifically the blend area to avoid creating gaps in the fog
#if defined FOG_VOLUME_BLENDING_MULTIPLY
    outColor = max(0, lerp(float4(1.0, 1.0, 1.0, 1.0), float4(saturate(albedo * extinction), extinction), fade.xxxx));
#else
    extinction *= fade;
    outColor = max(0, float4(saturate(albedo * extinction), extinction));
#endif
}
```

下图展示做Box剔除前后的区别， 剔除后可以减少一些不必要片元的计算

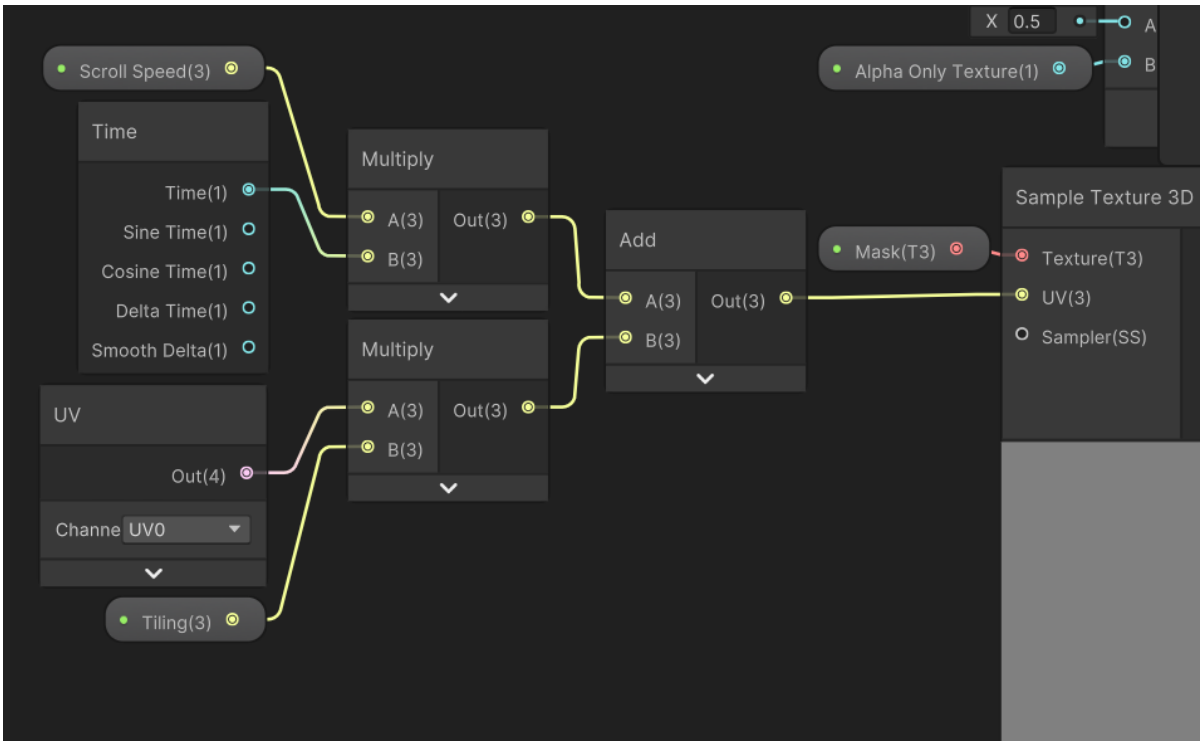



```
void GetVolumeData(FragInputs fragInputs, float3 V, out float3 scatteringColor, out float density)
{
    SurfaceDescriptionInputs surfaceDescriptionInputs = FragInputsToSurfaceDescriptionInputs(fragInputs, V);
    SurfaceDescription surfaceDescription = SurfaceDescriptionFunction(surfaceDescriptionInputs);

    scatteringColor = surfaceDescription.BaseColor;
    density = surfaceDescription.Alpha;
}
```

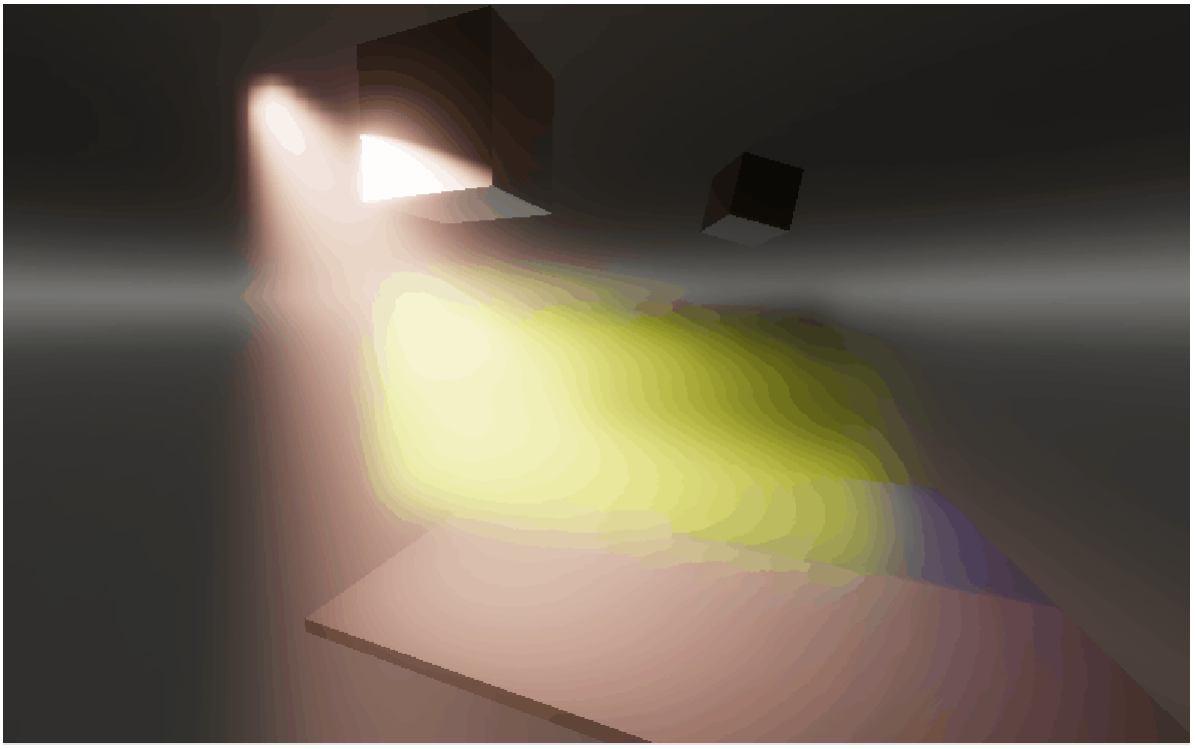
```
FragInputs BuildFragInputs(VertexToFragment v2f, float3 voxelPositionOS, float3 voxelClipSpace)
{
    FragInputs output;
    ZERO_INITIALIZE(FragInputs, output);

    float3 positionWS = mul(UNITY_MATRIX_M, float4(voxelPositionOS, 1)).xyz;
    output.positionSS = v2f.positionCS;
    output.positionRWS = output.positionPredisplacementRWS = positionWS;
    output.positionPixel = uint2(v2f.positionCS.xy);
    output.texCoord0 = float4(saturate(voxelClipSpace * 0.5 + 0.5), 0);
    output.tangentToWorld = k_identity3x3;
```

```
12
13      return output;
14  }
```

以下代码为计算UV



Volumetric Lighting



着色部分主要有两个步骤：

1. 体积光照
2. 过滤（可选）

VolumetricLighting

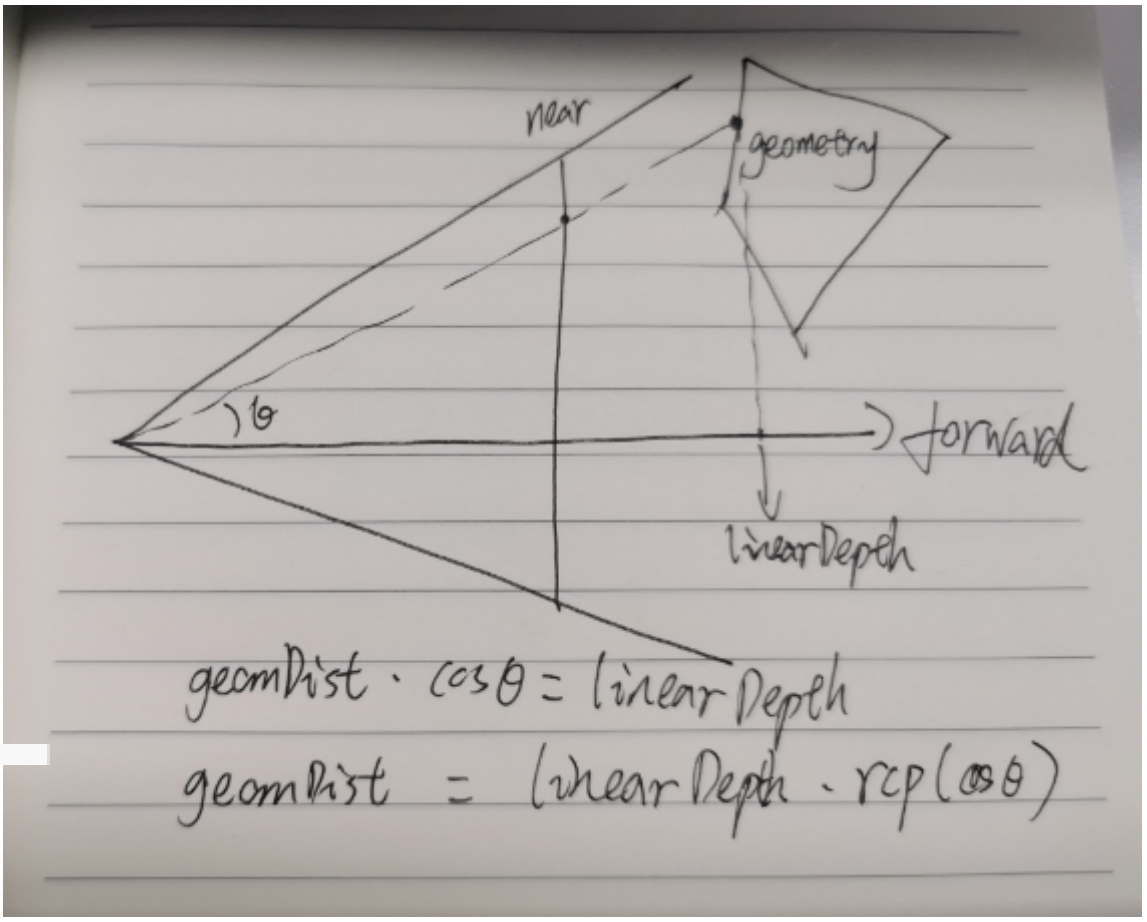
```
Runtime/Lighting/VolumetricLighting/VolumetricLighting.compute
```

下图是 VolumetricLighting Kernel的核心代码片段， 对于构造Ray以及计算导数部分，代码跟之前第一个Pass是一样的

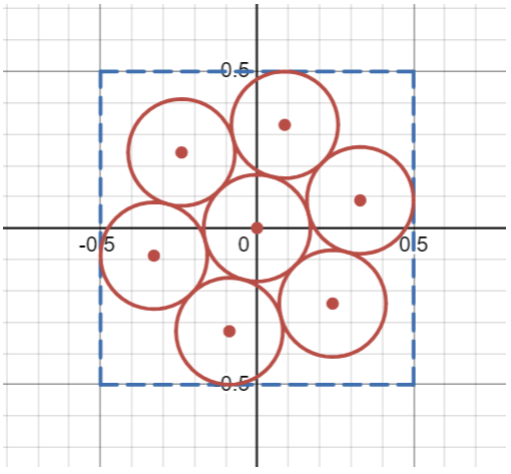
```
#ifdef ENABLE_REPROJECTION
    float2 sampleOffset = _VBufferSampleOffset.xy;
#else
    float2 sampleOffset = 0;
#endif
    // 重投影后的光线方向
    ray.jitterDirWS = normalize(ray.centerDirWS + sampleOffset.x * ray.xDirDerivWS
                                                + sampleOffset.y * ray.yDirDerivWS);
    // dot(ray.jitterDirWS, F) 为 cosine
    // cosine = Near / sStart,
    // sStart = Near / cosine;
    // NOTE: F是单位矩阵
    /*                      Near
                             .
                        .    |
                   .         . p
               .     .       |
           .   .   theta      |
       O ------------------------------> forward
           .                  |
               .              |
                   .          |
                        .     |
                              .
       cos(theta) = dot(normalize(op), forward) ;  ps: forward 都为单位向量
       cos(theta) = near / length(op);
    */
    float tStart = g_fNearPlane / dot(ray.jitterDirWS, F);

    // We would like to determine the screen pixel (at the full resolution) which
    // the jittered ray corresponds to. The exact solution can be obtained by intersecting
    // the ray with the screen plane, e.i. (ViewSpace(jitterDirWS).z = 1). That's a little expensive.
    // So, as an approximation, we ignore the curvature of the frustum.
    // 计算对应的全屏幕坐标
    uint2 pixelCoord = (uint2)((voxelCoord + 0.5 + sampleOffset) * _VBufferVoxelSize);

#ifdef VL_PRESET_OPTIMAL
    // The entire thread group is within the same light tile.
    // 正好每个线程组负责一条射线，且slice为8
    uint2 tileCoord = groupOffset * VBUFFER_VOXEL_SIZE / TILE_SIZE_BIG_TILE;
#else
    // No compile-time optimizations, no scalarization.
```

```
    // 计算出tile的坐标
    uint2 tileCoord = pixelCoord / TILE_SIZE_BIG_TILE;
#endif
    // 计算出Grid的Index
    uint  tileIndex = tileCoord.x + _NumTileBigTileX * tileCoord.y;
    // This clamp is important as _VBufferVoxelSize can have float value which can cause en overflow (Crash on Vulkan and Metal)
    tileIndex = min(tileIndex, _NumTileBigTileX * _NumTileBigTileY);

    // Do not jitter 'voxelCoord' else. It's expected to correspond to the center of the voxel.
    // 此方法注释见下
    PositionInputs posInput = GetPositionInput(voxelCoord, _VBufferViewportSize.zw, tileCoord);

    ray.geomDist = FLT_INF;
    ray.maxDist = FLT_INF;
#if USE_DEPTH_BUFFER
    // 采样_CameraDepthTexture
    float deviceDepth = LoadCameraDepth(pixelCoord);

    if (deviceDepth > 0) // Skip the skybox
    {
        // Convert it to distance along the ray. Doesn't work with tilt shift, etc.
        // 转成线性深度 [Near, Far]
        float linearDepth = LinearEyeDepth(deviceDepth, _ZBufferParams);
        // 同样的余弦定理应用计算出到光到几何体的线性距离
        // 详情见下图
        ray.geomDist = linearDepth * rcp(dot(ray.jitterDirWS, F));

        float2 UV = posInput.positionNDC * _RTHandleScale.xy;

        // This should really be using a max sampler here. This is a bit overdilating given that it is already dilated.
        // Better to be safer though.
        float4 d = GATHER_RED_TEXTURE2D_X(_MaxZMaskTexture, s_point_clamp_sampler, UV) * rcp(dot(ray.jitterDirWS, F));
        ray.maxDist = max(Max3(d.x, d.y, d.z), d.w);
    }
#endif

    // TODO
    LightLoopContext context;
    context.shadowContext = InitShadowContext();
    uint featureFlags     = 0xFFFFFFFF;

    ApplyCameraRelativeXR(ray.originWS);

    FillVolumetricLightingBuffer(context, featureFlags, posInput, tileIndex, groupIndex, ray, tStart);
```

下图是计算geomDist的示意图



下图是进行重投影的Offset选择规则示意图，总共7个采样点，分7帧进行，每一帧选择1个采样点。



下面是填充LightingBuffer的核心代码

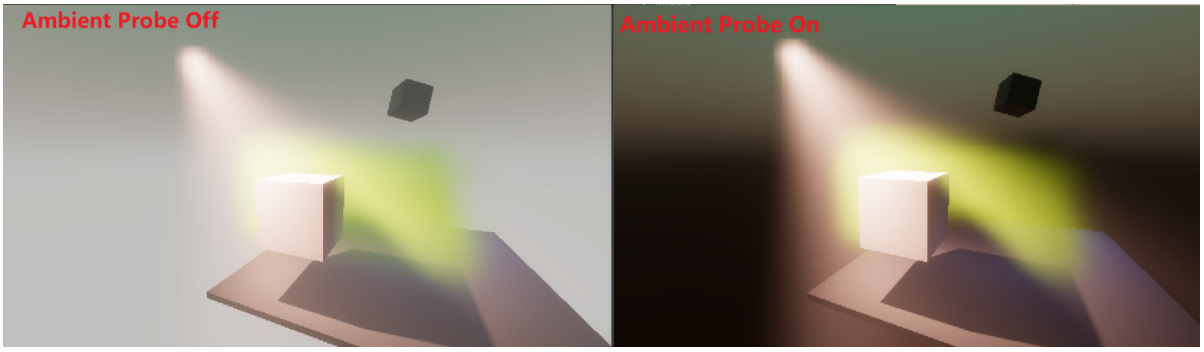
1. 数据准备部分： 准备灯光数据

```
// Computes the in-scattered radiance along the ray.
void FillVolumetricLightingBuffer(LightLoopContext context, uint
featureFlags,
                                  PositionInputs posInput, uint tileIndex,
int groupIdx, JitteredRay ray, float tStart)
{
    uint lightCount, lightStart;

#ifdef USE_BIG_TILE_LIGHTLIST
```

```
    // Offset for stereo rendering
    tileIndex += unity_StereoEyeIndex * _NumTileBigTileX * _NumTileBigTileY;

    // The "big tile" list contains the number of objects contained within the tile followed by the
    // list of object indices. Note that while objects are already sorted by type, we don't know the
    // number of each type of objects (e.g. lights), so we should remember to break out of the loop.
    lightCount = g_vBigTileLightList[MAX_NR_BIG_TILE_LIGHTS_PLUS_ONE * tileIndex];
    // On Metal for unknow reasons it seems we have bad value sometimes causing freeze / crash. This min here prevent it.
    lightCount = min(lightCount, MAX_NR_BIG_TILE_LIGHTS_PLUS_ONE);
    lightStart = MAX_NR_BIG_TILE_LIGHTS_PLUS_ONE * tileIndex + 1;

    // For now, iterate through all the objects to determine the correct range.
    // TODO: precompute this, of course.
    {
        uint offset = 0;

        int validLightCount = 0;

        for (; offset < lightCount; offset++)
        {
            uint objectIndex = FetchIndex(lightStart, offset);
#if PRE_FILTER_LIGHT_LIST
            LightData light = _LightDatas[objectIndex];
            if (light.volumetricLightDimmer > 0 && validLightCount < MAX_SUPPORTED_LIGHTS && objectIndex < _EnvLightIndexShift)
            {
                // 提前按group进行index的存储
                gs_localLightList[groupIdx][validLightCount++] = objectIndex;
            }
#else
            validLightCount++;
#endif
            if (objectIndex >= _EnvLightIndexShift)
            {
                // We have found the last local analytical light.
                break;
            }
        }

        lightCount = validLightCount;
    }

#else  // USE_BIG_TILE_LIGHTLIST

    lightCount = _PunctualLightCount;
    lightStart = 0;

#endif // USE_BIG_TILE_LIGHTLIST

```

2. 采样环境光

```
// 获取起点t和每一次步进的深度
    float t0 = max(tStart, DecodeLogarithmicDepthGeneralized(0, _VBufferDistanceDecodingParams));
    float de = _VBufferRcpSliceCount; // Log-encoded distance between slices

    // The contribution of the ambient probe does not depend on the position,
    // only on the direction and the length of the interval.
    // SampleSH9() evaluates the 3-band SH in a given direction.
    // The probe is already pre-convolved with the phase function.
    // Note: anisotropic, no jittering.
    // 对环境光进行球谐采样
    float3 probeInScatteredRadiance = EvaluateVolumetricAmbientProbe(ray.centerDirWS);


```



3. 根据slice进行步进，同时确定抖动offset

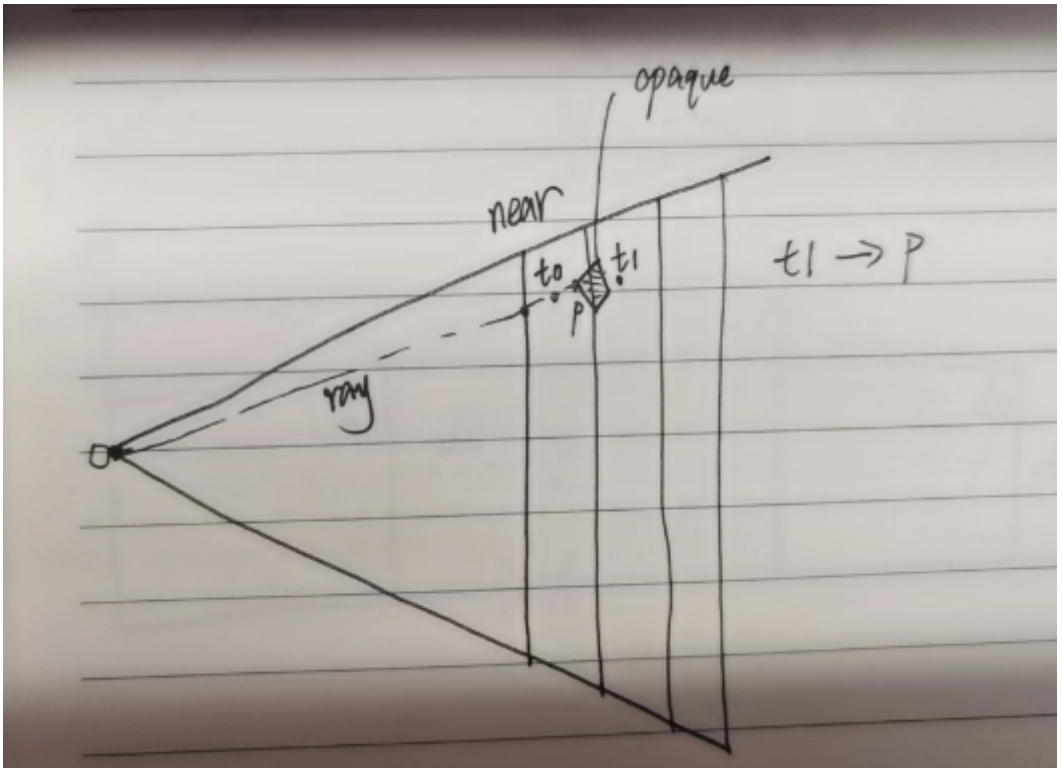
```
    float3 totalRadiance = 0;
    float  opticalDepth  = 0;
    uint slice = 0;
    for (; slice < _VBufferSliceCount; slice++)
    {   // positionSS 就是 voxelCoord
        uint3 voxelCoord = uint3(posInput.positionSS, slice + _VBufferSliceCount * unity_StereoEyeIndex);

        float e1 = slice * de + de; // (slice + 1) / sliceCount
        // 当前的起点
        float t1 = max(tStart, DecodeLogarithmicDepthGeneralized(e1, _VBufferDistanceDecodingParams));
        float tNext = t1;

    #if USE_DEPTH_BUFFER
        bool containsOpaqueGeometry = IsInRange(ray.geomDist, float2(t0, t1));
        // 如果射线穿透了opaque物体，则需要对t1做修正
        if (containsOpaqueGeometry)
```

```
        {
            // Only integrate up to the opaque surface (make the voxel shorter, but not completely flat).
            // Note that we can NOT completely stop integrating when the ray reaches geometry, since
            // otherwise we get flickering at geometric discontinuities if reprojection is enabled.
            // In this case, a temporally stable light leak is better than flickering.
            // 对t1进行修正，详见下图
            t1 = max(t0 * 1.0001, ray.geomDist);
        }
    #endif
        float dt = t1 - t0; // Is geometry-aware
        if(dt <= 0.0) // 积分微元非正数，则无需积分
        {
            _VBufferLighting[voxelCoord] = 0;
#ifdef ENABLE_REPROJECTION
            _VBufferFeedback[voxelCoord] = 0;
#endif
            t0 = t1;
            continue;
        }

        // Accurately compute the center of the voxel in the log space. It's important to perform
        // the inversion exactly, since the accumulated value of the integral is stored at the center.
        // We will use it for participating media sampling, asymmetric scattering and reprojection.
        // 求出当前Slice中心点的t
        float  t = DecodeLogarithmicDepthGeneralized(e1 - 0.5 * de, _VBufferDistanceDecodingParams);
        float3 centerWS = ray.originWS + t * ray.centerDirWS;

        // Sample the participating medium at the center of the voxel.
        // We consider it to be constant along the interval [t0, t1] (within the voxel).
        // 采样
        float4 density = LOAD_TEXTURE3D(_VBufferDensity, voxelCoord);

        float3 scattering = density.rgb;
        float  extinction = density.a;
        // 用于控制相函数的方向，[-1, 1]
        float  anisotropy = _GlobalFogAnisotropy;

        // Perform per-pixel randomization by adding an offset and then sampling uniformly
        // (in the log space) in a vein similar to Stochastic Universal Sampling:
        // https://en.wikipedia.org/wiki/Stochastic_universal_sampling
        // 哈希
        float perPixelRandomOffset = GenerateHashedRandomFloat(posInput.positionSS);

    #ifdef ENABLE_REPROJECTION
```

```
        // This is a time-based sequence of 7 equidistant numbers from 1/14 to 13/14.
        // Each of them is the centroid of the interval of length 2/14.
        float rndVal = frac(perPixelRandomOffset + _VBufferSampleOffset.z);
    #else
        float rndVal = frac(perPixelRandomOffset + 0.5);
    #endif
        /*
          struct VoxelLighting
       {
           float3 radianceComplete;
           float3 radianceNoPhase;
       };
        */
        VoxelLighting aggregateLighting;
        ZERO_INITIALIZE(VoxelLighting, aggregateLighting);

        // Prevent division by 0.
        extinction = max(extinction, FLT_MIN);

        {
            // 着色部分代码段（见下一段）
            ...
                ...
                    ...
        }

        // Compute the optical depth up to the center of the interval.
        // opticalDepth可以理解为当前光子的深度位置，用来计算transmit
        opticalDepth += 0.5 * blendValue.a;

        // Store the voxel data.
        // Note: for correct filtering, the data has to be stored in the perceptual space.
        // This means storing the tone mapped radiance and transmittance instead of optical depth.
        // See "A Fresh Look at Generalized Sampling", p. 51.
        // TODO: re-enable tone mapping after implementing pre-exposure.
        // 保存着色结果
        _VBufferLighting[voxelCoord] = LinearizeRGBD(float4(/*FastTonemap*/(totalRadiance), opticalDepth)) * float4(GetCurrentExposureMultiplier().xxx, 1);

        // Compute the optical depth up to the end of the interval.
        opticalDepth += 0.5 * blendValue.a;

        // 到达最远距离，直接退出步进
        if (t0 * 0.99 > ray.maxDist)
        {
            break;
        }

        t0 = tNext;
    } // end of loop

   // 结束上面的步进后，对于剩余的slice，需要把数据清零
```

```
    for (; slice < _VBufferSliceCount; slice++)
    {
        uint3 voxelCoord = uint3(posInput.positionSS, slice +
_VBufferSliceCount * unity_StereoEyeIndex);
        _VBufferLighting[voxelCoord] = 0;
#ifdef ENABLE_REPROJECTION
        _VBufferFeedback[voxelCoord] = 0;
#endif

    } // end of loop
}
```

下图是对t1进行修正的示意图



4.光的着色

下面是着色部分的代码，主要分为**平行光**，**局部光**，**全局光照**三部分

```
    if (featureFlags & LIGHTFEATUREFLAGS_DIRECTIONAL)
       {
           // 这里计算平行光的着色结果
           VoxelLighting lighting =
EvaluateVoxelLightingDirectional(context, featureFlags, posInput,

centerWS, ray, t0, t1, dt, rndVal,

extinction, anisotropy);

           aggregateLighting.radianceNoPhase  += lighting.radianceNoPhase;
           aggregateLighting.radianceComplete += lighting.radianceComplete;
       }

       #ifdef SUPPORT_LOCAL_LIGHTS
```

```
        {   // 这里计算局部光（点光，聚光...）的着色结果
            VoxelLighting lighting = EvaluateVoxelLightingLocal(context, groupIdx, featureFlags, posInput,
                                                                lightCount, lightStart,
                                                                centerWS, ray, t0, t1, dt, rndVal,
                                                                extinction, anisotropy);

            aggregateLighting.radianceNoPhase  += lighting.radianceNoPhase;
            aggregateLighting.radianceComplete += lighting.radianceComplete;
        }
        #endif

        #if SAMPLE_PROBE_VOLUMES
        {
            // 计算低频的全局光照
            float3 apvDiffuseGI = EvaluateVoxelDiffuseGI(posInput, ray, t0, dt, rndVal, extinction);
            aggregateLighting.radianceNoPhase += apvDiffuseGI;
            aggregateLighting.radianceComplete += apvDiffuseGI;

        }
        #endif

```

4.重投影

下面的代码主要是做**重投影**，以及着色结果的**合并**

重投影时从_**VBufferHistory**读取上一帧数据，

并且把这一帧结果写进_**VBufferFeedback**.

在HDCamera会创建一个双buffer用于重投影

```
internal RTHandle[] volumetricHistoryBuffers; // Double-buffered; only used for reprojection

```

```
if (passData.enableReprojection)
{
    passData.feedbackBuffer = builder.WriteTexture( input: renderGraph.ImportTexture(hdCamera.volumetricHistoryBuffers[currIdx]));
    passData.historyBuffer = builder.ReadTexture( input: renderGraph.ImportTexture(hdCamera.volumetricHistoryBuffers[prevIdx]));
}
```

其中feedbackBuffer用于存当前帧的结果

对于历史帧的混合计算如下

```
float numFrames     = 7;
float frameWeight   = 1 / numFrames;
float historyWeight = 1 - frameWeight;

return historyWeight;
```

```
// Perform temporal blending in the log space ("Pixar blend").
normalizedBlendValue = lerp(normalizedVoxelValue, reprojValue,
ComputeHistoryWeight());
```

以下代码为重投影的计算过程

```
#ifdef ENABLE_REPROJECTION
     // Clamp here to prevent generation of NaNs.
     float4 voxelValue           = float4(aggregateLighting.radianceNoPhase,
extinction * dt);
     float4 linearizedVoxelValue = LinearizeRGBD(voxelValue);
    // 进行单位化，可以确保不会受不同长度dt的影响
     float4 normalizedVoxelValue = linearizedVoxelValue * rcp(dt);
     float4 normalizedBlendValue = normalizedVoxelValue;

 #if (SHADEROPTIONS_CAMERA_RELATIVE_RENDERING != 0) &&
defined(USING_STEREO_MATRICES)
     // With XR single-pass, remove the camera-relative offset for the
reprojected sample
     centerWS -= _WorldSpaceCameraPosViewOffset;
 #endif

     // Reproject the history at 'centerWS'.
     float4 reprojValue = SampleVBuffer(TEXTURE3D_ARGS(_VBufferHistory,
s_linear_clamp_sampler),
                                        centerWS,
                                        _PrevCamPosRWS.xyz,
                                        UNITY_MATRIX_PREV_VP,
                                        _VBufferPrevViewportSize,
                                        _VBufferHistoryViewportScale.xyz,
                                        _VBufferHistoryViewportLimit.xyz,
                                        _VBufferPrevDistanceEncodingParams,
                                        _VBufferPrevDistanceDecodingParams,

                                        // 历史帧中存的是加了曝光系数的，因此需要除
以历史曝光系数
                                        false, false, true) *
float4(GetInversePreviousExposureMultiplier().xxx, 1);

     bool reprojSuccess = (_VBufferHistoryIsValid != 0) && (reprojValue.a !=
0);

     if (reprojSuccess)
     {
         // Perform temporal blending in the log space ("Pixar blend").
         // ComputeHistoryWeight() = 6/7
         normalizedBlendValue = lerp(normalizedVoxelValue, reprojValue,
ComputeHistoryWeight());
     }

     // Store the feedback for the voxel.
     // TODO: dynamic lights (which update their position, rotation, cookie
or shadow at runtime)
```

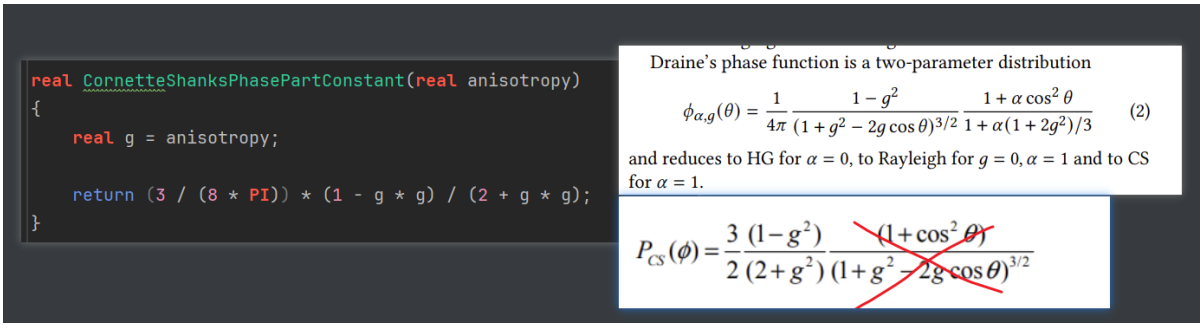
```
38      // do not support reprojection and should neither read nor write to the
    history buffer.
39      // This will cause them to alias, but it is the only way to prevent
    ghosting.
40      _VBufferFeedback[voxelCoord] = normalizedBlendValue *
    float4(GetCurrentExposureMultiplier().xxx, 1);
41
42      float4 linearizedBlendValue = normalizedBlendValue * dt;
43      float4 blendValue = DelinearizeRGBD(linearizedBlendValue);
44
45   #ifdef ENABLE_ANISOTROPY
46      // Estimate the influence of the phase function on the results of the
    current frame.
47      float3 phaseCurrFrame;
48
49      phaseCurrFrame.r = SafeDiv(aggregateLighting.radianceComplete.r,
    aggregateLighting.radianceNoPhase.r);
50      phaseCurrFrame.g = SafeDiv(aggregateLighting.radianceComplete.g,
    aggregateLighting.radianceNoPhase.g);
51      phaseCurrFrame.b = SafeDiv(aggregateLighting.radianceComplete.b,
    aggregateLighting.radianceNoPhase.b);
52
53      // Warning: in general, this does not work!
54      // For a voxel with a single light, 'phaseCurrFrame' is monochromatic,
    and since
55      // we don't jitter anisotropy, its value does not change from frame to
    frame
56      // for a static camera/scene. This is fine.
57      // If you have two lights per voxel, we compute:
58      // phaseCurrFrame = (phaseA * lightingA + phaseB * lightingB) /
    (lightingA + lightingB).
59      // 'phaseA' and 'phaseB' are still (different) constants for a static
    camera/scene.
60      // 'lightingA' and 'lightingB' are jittered, so they change from frame
    to frame.
61      // Therefore, 'phaseCurrFrame' becomes temporarily unstable and can
    cause flickering in practice. :-(
62      blendValue.rgb *= phaseCurrFrame;
63   #endif // ENABLE_ANISOTROPY
64
65   #else // NO REPROJECTION
66
67   #ifdef ENABLE_ANISOTROPY
68      float4 blendValue = float4(aggregateLighting.radianceComplete,
    extinction * dt);
69   #else
70      float4 blendValue = float4(aggregateLighting.radianceNoPhase,
     extinction * dt);
71   #endif // ENABLE_ANISOTROPY
72
73   #endif // ENABLE_REPROJECTION
74
75      // Compute the transmittance from the camera to 't0'.
76      // 这里的计算依据Beer-Lambert定律
77      float transmittance = TransmittanceFromOpticalDepth(opticalDepth);
78
```

```
#ifdef ENABLE_ANISOTROPY
    float phase = _CornetteShanksConstant;
#else
    float phase = IsotropicPhaseFunction();
#endif // ENABLE_ANISOTROPY

    // Integrate the contribution of the probe over the interval.
    // Integral{a, b}{Transmittance(0, t) * L_s(t) dt} = Transmittance(0, a) * Integral{a, b}{Transmittance(0, t - a) * L_s(t) dt}.
    float3 probeRadiance = probeInScatteredRadiance * TransmittanceIntegralHomogeneousMedium(extinction, dt);

    // Accumulate radiance along the ray.
    // blendValue前面已经乘了一次CS相函数非常数项的部分，这里phase是常数项部分
    totalRadiance += transmittance * scattering * (phase * blendValue.rgb + probeRadiance);
```

HDRP相位函数分析

在HDRP中，采用的CS模型的相函数，主要拆分为了三个部分：

1.常量部分



参考：https://research.nvidia.com/labs/rtr/approximate-mie/publications/approximate-mie.pdf

HDRP上面公式中常数项多乘了 1/4pi，可能是考虑到了均匀介质下，对于每个单位立体角散射的概率就是 1/4pi。

2.变量部分

```
real CornetteShanksPhasePartVarying(real anisotropy, real cosTheta)
{
    return CornetteShanksPhasePartSymmetrical(cosTheta) *
           CornetteShanksPhasePartAsymmetrical(anisotropy, cosTheta); // h * x^(-3/2)
}
```

在公式的变量部分，又拆分成了**各向同性**和**各向异性**两部分。

2.1 各项同性

2.2 各向异性

```
real CornetteShanksPhasePartAsymmetrical(real anisotropy, real cosTheta)
{
    real g = anisotropy;
    real x = 1 + g * g - 2 * g * cosTheta;
    real f = rsqrt(max(x, REAL_EPS)); // x^(-1/2)
    return f * f * f;                 // x^(-3/2)
}
```

$$P_{CS}(\phi) = \frac{3}{2}\frac{(1-g^2)}{(2+g^2)}\frac{(1+\cos^2\theta)}{(1+g^2-2g\cos\theta)^{3/2}}$$

光的着色

以下代码是对平行光内散射的积分计算

```
// Computes the light integral (in-scattered radiance) within the voxel.
// Multiplication by the scattering coefficient and the phase function is performed outside.
VoxelLighting EvaluateVoxelLightingDirectional(LightLoopContext context, uint featureFlags, PositionInputs posInput, float3 centerWS,
                                               JitteredRay ray, float t0, float t1, float dt, float rndVal, float extinction, float anisotropy)
{
    VoxelLighting lighting;
    ZERO_INITIALIZE(VoxelLighting, lighting);

    BuiltinData unused; // Unused for now, so define once
    ZERO_INITIALIZE(BuiltinData, unused);

    const float NdotL = 1;

    float tOffset, weight;
    //
    ImportanceSampleHomogeneousMedium(rndVal, extinction, dt, tOffset, weight);

    float t = t0 + tOffset;
    posInput.positionWS = ray.originWS + t * ray.jitterDirWS;

    context.shadowValue = 1.0;

    // Evaluate sun shadows.
    if (_DirectionalShadowIndex >= 0)
    {
        DirectionalLightData light = _DirectionalLightDatas[_DirectionalShadowIndex];

        // Prep the light so that it works with non-volumetrics-aware code.
        light.contactShadowMask  = 0;
        light.shadowDimmer       = light.volumetricShadowDimmer;

        float3 L = -light.forward;

        // Is it worth sampling the shadow map?
```

```
        if ((light.volumetricLightDimmer > 0) &&
(light.volumetricShadowDimmer > 0))
        {
            #if SHADOW_VIEW_BIAS
                // Our shadows only support normal bias. Volumetrics has no
access to the surface normal.
                // We fake view bias by invoking the normal bias code with
the view direction.
                float3 shadowN = -ray.jitterDirWS;
            #else
                float3 shadowN = 0; // No bias
            #endif // SHADOW_VIEW_BIAS

            // 采样阴影
            context.shadowValue =
GetDirectionalShadowAttenuation(context.shadowContext,

 posInput.positionSS, posInput.positionWS,
                                            shadowN,

 light.shadowIndex, L);
        }
    }
    else
    {
        context.shadowValue = 1;
    }

    for (uint i = 0; i < _DirectionalLightCount; ++i)
    {
        DirectionalLightData light = _DirectionalLightDatas[i];

        // Prep the light so that it works with non-volumetrics-aware code.
        light.contactShadowMask  = 0;
        light.shadowDimmer       = light.volumetricShadowDimmer;

        float3 L = -light.forward;

        // Is it worth evaluating the light?
        float3 color; float attenuation;
        if (light.volumetricLightDimmer > 0)
        {
            float4 lightColor = EvaluateLight_Directional(context,
posInput, light);
            // The volumetric light dimmer, unlike the regular light
dimmer, is not pre-multiplied.
            lightColor.a *= light.volumetricLightDimmer;
            lightColor.rgb *= lightColor.a; // Composite

            #if SHADOW_VIEW_BIAS
                // Our shadows only support normal bias. Volumetrics has no
access to the surface normal.
                // We fake view bias by invoking the normal bias code with
the view direction.
                float3 shadowN = -ray.jitterDirWS;
            #else
```

```
                float3 shadowN = 0; // No bias
            #endif // SHADOW_VIEW_BIAS

            // This code works for both surface reflection and thin object transmission.
            SHADOW_TYPE shadow = EvaluateShadow_Directional(context, posInput, light, unused, shadowN);
            lightColor.rgb *= ComputeShadowColor(shadow, light.shadowTint, light.penumbraTint);

            // Important:
            // Ideally, all scattering calculations should use the jittered versions
            // of the sample position and the ray direction. However, correct reprojection
            // of asymmetrically scattered lighting (affected by an anisotropic phase
            // function) is not possible. We work around this issue by reprojecting
            // lighting not affected by the phase function. This basically removes
            // the phase function from the temporal integration process. It is a hack.
            // The downside is that anisotropy no longer benefits from temporal averaging,
            // and any temporal instability of anisotropy causes causes visible jitter.
            // In order to stabilize the image, we use the voxel center for all
            // anisotropy-related calculations.
            float cosTheta = dot(L, ray.centerDirWS);
            // 计算相位函数，得到一个概率值
            float phase    = CornetteShanksPhasePartVarying(anisotropy, cosTheta);

            // Compute the amount of in-scattered radiance.
            // Note: the 'weight' accounts for transmittance from 't0' to 't'.
            // 这里的weight考虑了衰减和Beer-Lambert定律
            lighting.radianceNoPhase += (weight * lightColor.rgb);
            lighting.radianceComplete += (weight * lightColor.rgb) * phase;
        }
    }

    return lighting;
}
```

```
// Samples the interval of homogeneous participating medium using the closed-form tracking approach
// (proportionally to the transmittance).
// Returns the offset from the start of the interval and the weight = (transmittance / pdf).
// Ref: Monte Carlo Methods for Volumetric Light Transport Simulation, p. 5.
void ImportanceSampleHomogeneousMedium(real rndVal, real extinction, real intervalLength,
                                       out real offset, out real weight)
{
    // pdf    = extinction * exp(extinction * (intervalLength - t)) / (exp(intervalLength * extinction) - 1)
    // pdf    = extinction * exp(-extinction * t) / (1 - exp(-extinction * intervalLength))
    // weight = TransmittanceFromOpticalDepth(t) / pdf
    // weight = exp(-extinction * t) / pdf
    // weight = (1 - exp(-extinction * intervalLength)) / extinction
    // weight = OpacityFromOpticalDepth(extinction * intervalLength) / extinctio

    real x = 1 - exp(-extinction * intervalLength);
    real c = rcp(extinction);

    // TODO: return 'rcpPdf' to support imperfect importance sampling...
    weight = x * c;
    offset = -log(1 - rndVal * x) * c;
}
```

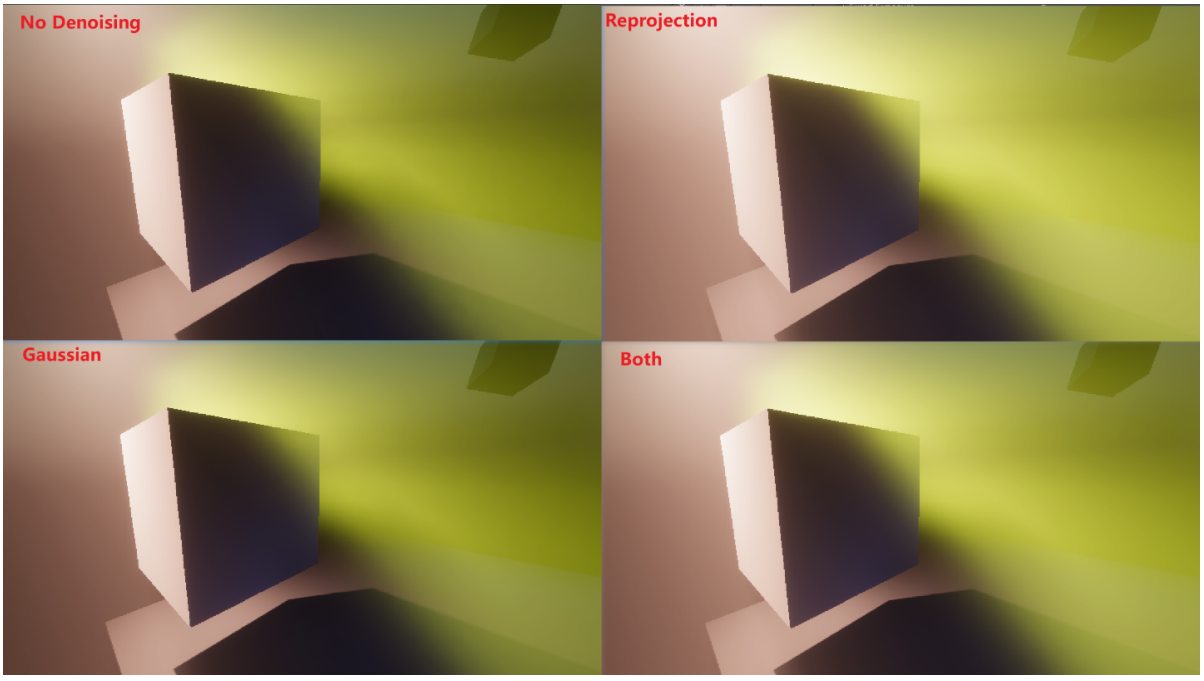
关于采样权重的计算，综合考虑了总衰减值和Beer-Lambert定律。

x为1-EXP(-extinction * intervalLength), 根据Beer-Lambert，越厚的物体透光率越差，这里算出经过intervalLength厚的介质后，剩余的比例

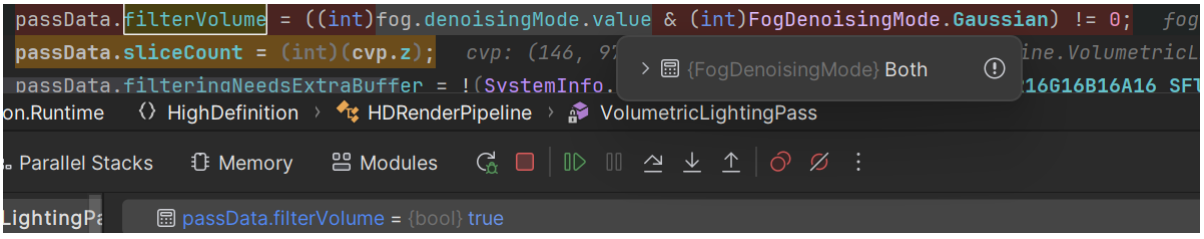
c: 衰减值extinction的倒数，也就是衰减距离，对于衰减距离越短的，对应的权重也比较小。

Denoising

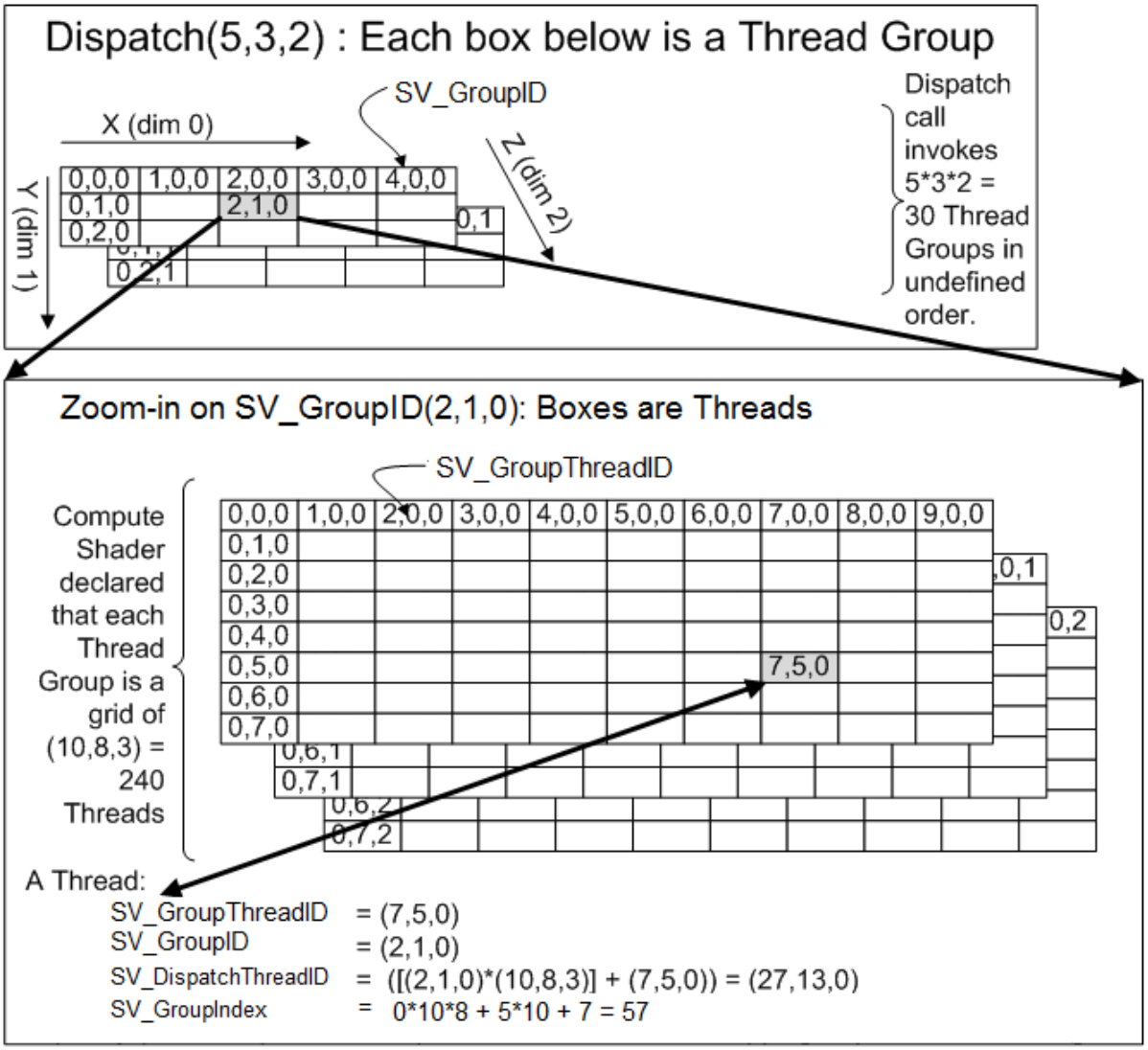
```
Runtime/Lighting/VolumetricLighting/VolumetricLightingFiltering.compute
```



Gaussian



在线程组派发的时候，对于xy每个线程组有8个线程数，对于z则每个线程组1个线程。




```
ctx.cmd.DispatchCompute(data.volumetricLightingFilteringCS,
data.volumetricFilteringKernel, HDUtils.DivRoundUp((int)data.resolution.x, 8),
    HDUtils.DivRoundUp((int)data.resolution.y, 8),
    data.sliceCount);
```

```
#define GAUSSIAN_SIGMA 1.0
#define GROUP_SIZE_1D_XY 8
#define GROUP_SIZE_1D_Z 1

// 每个组是8x8尺寸，而要做3x3滤波，边缘的像素就需要采样到别的Group，因此这里对Size + 2
#define FILTER_SIZE_1D (GROUP_SIZE_1D_XY + 2) // With a 8x8 group, we have a 10x10 working area
#define LDS_SIZE FILTER_SIZE_1D * FILTER_SIZE_1D

// TODO: May use 1 uint for 2 pixels
// 每个组是8x8，而实际在组内需要存10x10的大小才能做3x3滤波
groupshared float gs_cacheR[LDS_SIZE];
groupshared float gs_cacheG[LDS_SIZE];
groupshared float gs_cacheB[LDS_SIZE];
groupshared float gs_cacheA[LDS_SIZE];
```

```
//                  8                8                  1
[numthreads(GROUP_SIZE_1D_XY, GROUP_SIZE_1D_XY, GROUP_SIZE_1D_Z)]
void FilterVolumetricLighting(uint3 dispatchThreadId : SV_DispatchThreadID,
                              int   groupIndex       : SV_GroupIndex,
                              uint2 groupId          : SV_GroupID,
                              uint2 groupThreadId    : SV_GroupThreadID)
{
    // Do Something
}
```

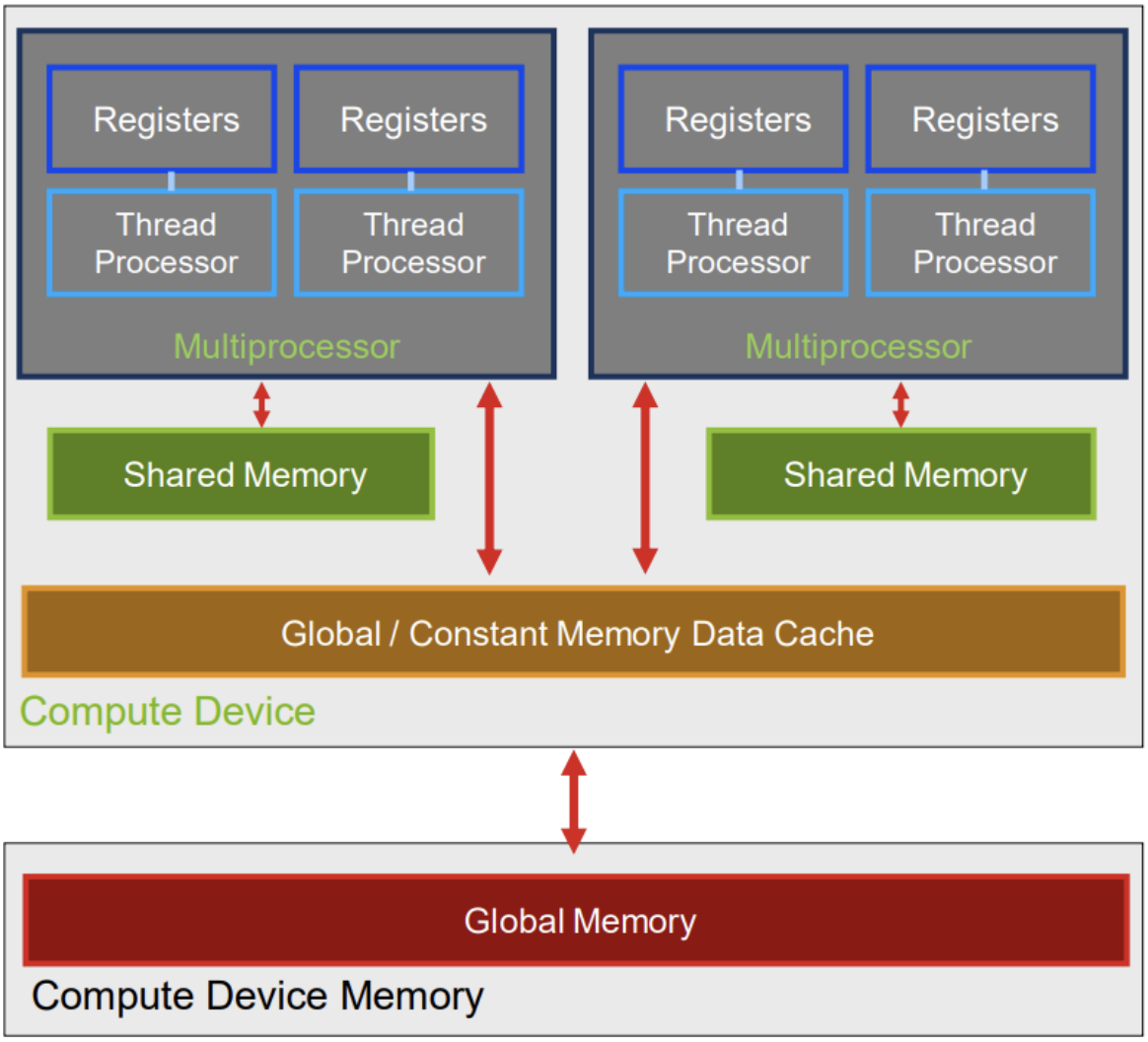
```
// Compute the coordinate that this thread needs to process
// groupId 为 线程组的id，每个线程组8个线程，因此 乘8，groupThreadId为线程组内线程的序号
// 得到2D grid的 xy 坐标
uint2 currentCoord  = groupId * GROUP_SIZE_1D_XY + groupThreadId;
// 由于在z方向，分配了TotalSlice个线程组，每个线程组只包含一个线程，因此直接对应slice的index
uint currentSlice = dispatchThreadId.z;

// Compute the output voxel coordinate
uint3 voxelCoord = uint3(currentCoord, currentSlice);
```

```
// groupIndex 代表线程在组内的索引，由于每个线程都会采样两次，总共需要10x10，因此索引限制在50
if (groupIndex < 50)
{
    // Load 2 values per thread.
    // 对于每个组，groupId * GROUP_SIZE_1D_XY得出每个组的起点坐标
    // 提前取数据到LDS中，可以在做滤波的时候减少采样的消耗
    PrefetchData(groupIndex, groupId * GROUP_SIZE_1D_XY, voxelCoord.z);
}

// Make sure all values are loaded in LDS by now.
GroupMemoryBarrierWithGroupSync();
```

```
void PrefetchData(uint groupIndex, uint2 groupOrigin, uint sliceIndex)
{
    // originXY 可能是负的啊？
    int2 originXY = groupOrigin - int2(1, 1);

    for (int i = 0; i < 2; ++i)
    {
        uint sampleID = i + (groupIndex * 2);
        int offsetX = sampleID % FILTER_SIZE_1D;
        int offsetY = sampleID / FILTER_SIZE_1D;

        int3 sampleCoord = int3(clamp(originXY.x + offsetX, 0, _VBufferViewportSize.x - 1),
```

```
                                clamp(originXY.y + offsetY, 0,
_VBufferViewportSize.y - 1),
                                sliceIndex);

        float4 sampleVal = _VBufferLighting[sampleCoord];

        int LDSIndex = offsetX + offsetY * FILTER_SIZE_1D;
        gs_cacheR[LDSIndex] = sampleVal.r;
        gs_cacheG[LDSIndex] = sampleVal.g;
        gs_cacheB[LDSIndex] = sampleVal.b;
        gs_cacheA[LDSIndex] = sampleVal.a;
    }
}
```

```
// Values used for accumulation
float sumW = 0.0;
float4 value = float4(0.0, 0.0, 0.0, 0.0);

const int radius = 1;
// 对每个切片做3x3高斯模糊
for (int idx = -radius; idx <= radius; ++idx)
{
    for (int idx2 = -radius; idx2 <= radius; ++idx2)
    {
        // Tap from LDS
        int2 tapAddress = (groupThreadId + 1) + int2(idx2, idx);
        uint ldsTapAddress = uint(tapAddress.x) % FILTER_SIZE_1D +
tapAddress.y * FILTER_SIZE_1D;
        float4 currentValue = GetSample(ldsTapAddress);

        // Compute the weight for this tap
        float weight = Gaussian(length(int2(idx, idx2)), GAUSSIAN_SIGMA);

        // Accumulate the value and weight
        value += currentValue * weight;
        sumW += weight;
    }

}

WriteOutput(voxelCoord, value / sumW);
```



OpaqueAtmosphericScattering

最终体积光计算结果存在LightingBuffer里，实际绘制到CameraTarget是在大气散射这个Pass进行叠加的，如下图所示