Analysis of Unreal Rendering System (05) - Light Source and Shadow

Table of contents

5.1 Overview

5.1.1 Content of this article

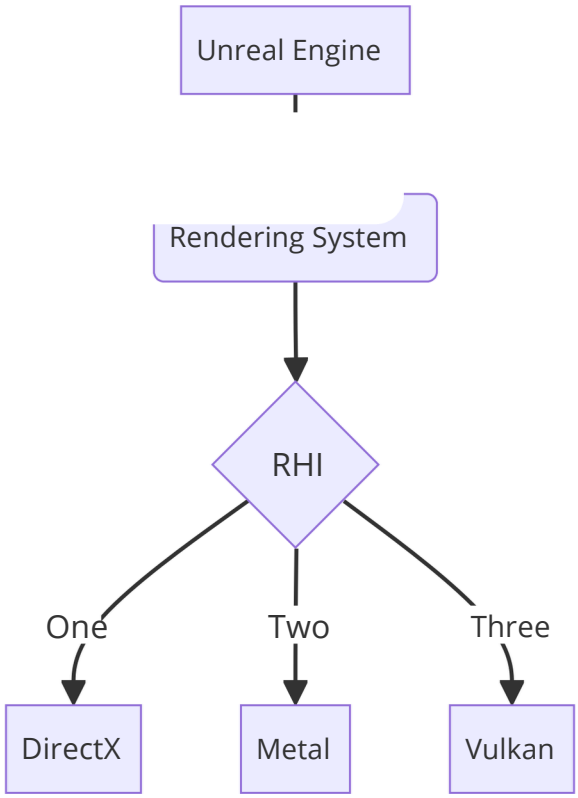
This article mainly describes the following contents of UE:

- Shader code and main modules.
- UE's light source.
- Shadows in UE.
- The mechanism and implementation of BasePass, including C++ and Shader. The
- mechanism and implementation of LightingPass, including C++ and Shader.

Among them, BasePass and LightingPass have been roughly analyzed in the previous article, and this article continues to analyze in depth.

This article will introduce **mermaid** syntax for the first time, which is similar to *LATAX*

using text description as a drawing tool is simple, lightweight, compact, and has high-definition lossless image quality. It goes perfectly with Markdown.



Vector legends made with mermaid syntax open a new era of text drawing.

High-energy warning ahead. This article is over 50,000 words long and is currently the longest article in this series, so be mentally prepared.

5.1.2 Basic Concepts

Some of the basic concepts and analysis of rendering involved in this article are as follows:

concept	abbreviation	English translation	Analysis
Shader	-	Shaders, Shader Code	Code snippets used for GPU execution. Common types include VS (vertex shader) and PS (pixel shader). It also refers generally to Shader code files.
Vertex Shader	VS	Vertex Shader	A code snippet executed during the vertex processing stage of the GPU, often used to handle vertex transformations.
Pixel Shader	PS	Pixel Shader	In OpenGL, it is called Fragment Shader, which is a code fragment specifically executed in the pixel processing stage, used to process rasterized pixels, such as lighting calculations.

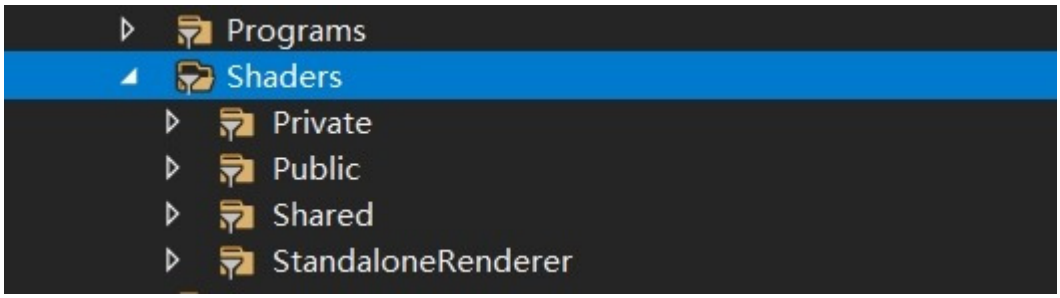
concept	abbreviation	English translation	Analysis
High Level Shading Language	HLSL	High-level Shading language	DirectX's dedicated shader language, Vulkan and OpenGL's are called GLSL, and Metal's are called MSL.

5.2 Shader System

Since the following chapters will directly analyze the shader code, this chapter will first make a brief analysis of UE's shader system so that everyone can transition to subsequent chapters well.

5.2.1 Shader Overview

UE's built-in Shader files are in the Engine\Shaders directory:



The total number of all Shader files exceeds 600. The suffix of Shader header files is ush, which can be included by other Shader files, and the implementation files are usf, which usually cannot be included by other Shader files. The following is a list of Shader directories and some modules:

```
Shaders/
    Private/
        BasePassCommon.ush
        BasePassPixelShader.usf
        BasePassVertexCommon.ush
        BasePassVertexShader.usf
        BRDF.ush            CapsuleLight.ush
        CapsuleLightIntegrate.ush
        CapsuleShadowShaders.usf
        Common.ush
        CommonViewUniformBuffer.ush
        DeferredLightingCommon.ush
        DeferredLightPixelShaders.usf
        DeferredLightVertexShaders.usf
        DeferredShadingCommon.ush
        Definitions.usf
        LightGridCommon.ush
        LightGridInjection.usf
        LocalVertexFactory.ush
```

```
    MaterialTemplate.ush
    RectLight.ush
    RectLightIntegrate.ush
    ShadingCommon.ush
    ShadingModels.ush
    ShadingModelsMaterial.ush
    ShadowDepthCommon.ush
    ShadowDepthPixelShader.usf
    ShadowDepthVertexShader.usf
    ShadowProjectionCommon.ush
    ShadowProjectionPixelShader.usf
    ShadowProjectionVertexShader.usf
    SHCommon.ush
    VertexFactoryCommon.ush
    (……)

  Public/
    FP16Math.ush
    Platform.ush
    ShaderVersion.ush
    Platform/
      D3D/
        D3DCommon.ush
      Metal/
        MetalCommon.ush
        MetalSubpassSupport.ush
        Vulkan/
        VulkanCommon.ush
        VulkanSubpassSupport.ush
        Shared/
    (……)
StandaloneRenderer/
    D3D/
      GammaCorrectionCommon.hlsl
      SlateDefaultVertexShader.hlsl (……)

    OpenGL/
      SlateElementPixelShader.glsl
      SlateVertexShader.glsl
      (……)
```

5.2.2 Shader module hierarchy

The first-tiershader modules are the lowest-level and most basic modules. These modules will not reference other modules, but will be referenced by many other modules. These modules are mainly:

- BasePassCommon.ush
- BRDF.ush
- CapsuleLight.ush
- Common.ush
- CommonViewUniformBuffer.ush
- Definitions.usf
- FP16Math.ush

- Platform.ush
- ShaderVersion.ush
- LightGridCommon.ush
- LocalVertexFactoryCommon.ush
- ShadingCommon.ush
- ShadowDepthCommon.ush
- ShadowProjectionCommon.ush
- SHCommon.ush
- (……)

The important or basic modules of the second tierwill reference the basic modules of the first tier, but will also be referenced by other tiers or modules:

- BasePassVertexCommon.ush
- CapsuleLightIntegrate.ush
- DeferredLightingCommon.ush
- DeferredShadingCommon.ush
- LocalVertexFactory.ush
- MaterialTemplate.ush
- RectLight.ush
- RectLightIntegrate.ush
- ShadingModels.ush
- ShadingModelsMaterial.ush
- VertexFactoryCommon.ush
- (……)

Finally, the**third-tier**modules are important modules that reference the first- and second-tier modules:

- BasePassPixelShader.usf
- BasePassVertexShader.usf
- CapsuleShadowShaders.usf
- DeferredLightPixelShaders.usf
- DeferredLightVertexShaders.usf
- ShadowProjectionPixelShader.usf
- ShadowProjectionVertexShader.usf
- (……)

5.2.3 Shader Basic Module

This section selects some basic or important shader modules and analyzes their interfaces, types, and definitions, so that everyone can have a general understanding of UE's built-in shader modules,

understand which modules UE comes with, and which built-in interfaces it has, so that you will have a clearer understanding when facing subsequent shader code analysis.

5.2.3.1 Platform.ush

It mainly defines macros, variables and tool interfaces related to graphics API (DirectX, OpenGL, Vulkan, Metal) and FEATURE_LEVEL. Some codes are as follows:

```
// FEATURE_LEVELMacro definition
#define    FEATURE_LEVEL_ES2_REMOVED        1
#define    FEATURE_LEVEL_ES3_1                2
#define    FEATURE_LEVEL_SM3                3
#define    FEATURE_LEVEL_SM4                4
#define    FEATURE_LEVEL_SM5                5
#define    FEATURE_LEVEL_MAX                6

//Initialize platform-dependent macros.
#ifndef    COMPILER_HLSLCC
#define    COMPILER_HLSLCC    0
#endif

#ifndef    COMPILER_HLSL
#define    COMPILER_HLSL    0
#endif

#ifndef    COMPILER_GLSL
#define    COMPILER_GLSL    0
#endif

#ifndef    COMPILER_GLSL_ES3_1
#define    COMPILER_GLSL_ES3_1    0
#endif

#ifndef    COMPILER_METAL
#define    COMPILER_METAL    0
#endif

#ifndef    SM5_PROFILE
#define    SM5_PROFILE    0
#endif

(.....)

//Floating point precision
#ifndef    FORCE_FLOATS
#define    FORCE_FLOATS    0
#endif

#if(!(ES3_1_PROFILE || METAL_PROFILE) || FORCE_FLOATS)
    #define    half    float
    #define    half1    float1
    #define    half2    float2
    #define    half3    float3
    #define    half4    float4
    #define    half3x3    float3x3
    #define    half4x4    float4x4
```

```
        #define half4x3 float4x3
        #define fixed float
        #define fixed1 float1
        #define fixed2 float2
        #define fixed3 float3
        #define fixed4 float4
        #define fixed3x3 float3x3
        #define fixed4x4 float4x4
        #define fixed4x3 float4x3
#endif
        #define
//PROFILEMacro Definition
#if    #define PROFILE
        #defineFEATURE_LEVEL FEATURE_LEVEL_SM5
        #define
#elifSM5_PROFILE
        #define SM5 = full dx11 features (high end UE4 rendering)
        #defineFEATURE_LEVEL FEATURE_LEVEL_SM5

#elifSM4_PROFILE
        #defineFEATURE_LEVEL FEATURE_LEVEL_SM4

(......)

//Branching statements.
#ifndef UNROLL
#define UNROLL
#endif

#ifndef UNROLL_N
#define UNROLL_N(N)
#endif

#ifndef LOOP
#define LOOP
#endif

#ifndef BRANCH
#define BRANCH
#endif

#ifndef FLATTEN
#define FLATTEN
#endif

(......)

//Tool class interface (when the compiler does not support it)
#if!COMPILER_SUPPORTS_MINMAX3

float min3(floata,floatb,floatc ); float max3(floata,floatb,float
c ); float2min3( float2 a, float2 b, float2 c ); float2max3
( float2 a, float2 b, float2 c );


(......)

#endif
```

```
//Unzip the data.
#if!defined(COMPILER_SUPPORTS_UNPACKBYTEN)
float UnpackByte0(uint v) {returnfloat(v &0xff); } float UnpackByte1(uint v) {return
float((v >>8) &0xff); } float UnpackByte2(uint v) {returnfloat((v >>16) &0xff); } float
UnpackByte3(uint v) {returnfloat(v >>twenty four); }

#endif// !COMPILER_SUPPORTS_UNPACKBYTEN

//Encapsulates some macros.
#ifndefSNORM
#ifCOMPILER_HLSLCC || PS4_PROFILE
    #define    SNORM
    #define    UNORM
#else
    #define    SNORM    snorm
    #define    UNORM    unorm
#endif
#endif

#ifndef    INFINITE_FLOAT
    #if    COMPILER_HLSLCC
        #define    INFINITE_FLOAT    3.402823e+38
    #else
        #define    INFINITE_FLOAT    1.#INF
    #endif
#endif
```

5.2.3.2 Common.ush

This module mainly includes macros, types, local variables, static variables, basic tool interfaces, etc. related to the graphics API or Feature Level, as follows:

```
//Type definition or encapsulation
#if    PIXELSHADER
    #define    MaterialFloat        half
    #define    MaterialFloat2       half2
    #define    MaterialFloat3       half3
    #define    MaterialFloat4       half4
    #define    MaterialFloat3x3        half3x3
    #define    MaterialFloat4x4        half4x4
    #define    MaterialFloat4x3        half4x3
#else
    // Material translated vertex shader code always uses floats, // Because it's used
    for things like world position and UVs
    #define    MaterialFloat        float
    #define    MaterialFloat2       float2
    #define    MaterialFloat3       float3
    #define    MaterialFloat4       float4
    #define    MaterialFloat3x3        float3x3
    #define    MaterialFloat4x4        float4x4
    #define    MaterialFloat4x3        float4x3
#endif

#if    POST_PROCESS_ALPHA
    #defineSceneColorLayout float4
```

```
    #defineCastFloat4ToSceneColorLayout(x) (x)
    #defineSetSceneColorLayoutToFloat4(dest,value) dest = (value)
#else
    #define    SceneColorLayout float3
    #define    CastFloat4ToSceneColorLayout(x) ((x).rgb)
    #defineSetSceneColorLayoutToFloat4(dest,value) dest.rgb = (value).rgb
#endif

structFScreenVertexOutput {

#ifMETAL_PROFILE || COMPILER_GLSL_ES3_1
    noperspective float2 UV : TEXCOORD0;
#else
    noperspective MaterialFloat2 UV : TEXCOORD0;
#endif
    float4 Position : SV_POSITION;
};

structFPixelShaderIn
{
    float4    SvPosition;
    uint Coverage;
    bool bIsFrontFace;
};

structFPixelShaderOut
{
    float4    MRT[8];
    uint Coverage;
    float    Depth;
};

//Macro and constant definitions
#defineUSE_GLOBAL_CLIP_PLANE (PLATFORM_SUPPORTS_GLOBAL_CLIP_PLANE &&
PROJECT_ALLOW_GLOBAL_CLIP_PLANE && !MATERIAL_DOMAIN_POSTPROCESS && !MATERIAL_DOMAIN_UI)

#define    POSITIVE_INFINITY    (asfloat(0x7F800000))
#define    NEGATIVE_INFINITY    (asfloat(0xFF800000))

const staticMaterialFloat PI =3.1415926535897932f; const static float
MaxHalfFloat =65504.0f; const static floatMax10BitsFloat =64512.0f;


#definePOW_CLAMP 0.000001f

//Common variable definitions
Texture2D            LightAttenuationTexture;
SamplerState         LightAttenuationTextureSampler;

//Basics, tool interface
float    ClampToHalfFloatRange(float    X);
float2   ClampToHalfFloatRange(float2   X);
float3   ClampToHalfFloatRange(float3   X);
float4   ClampToHalfFloatRange(float4   X);
float2Tile1Dto2D(floatxsize,floatidx); MaterialFloatLuminance
(MaterialFloat3 LinearColor); MaterialFloat
length2(MaterialFloat2                  v);
length2(MaterialFloat3                  v);
MaterialFloat
```

MaterialFloatlength2(MaterialFloat4 v); uintMod(uint a,
uint b); uint2Mod(uint2 a, uint2 b); uint3Mod(uint3 a,
uint3 b);

MaterialFloatUnClampedPow(MaterialFloat X, MaterialFloat Y); MaterialFloat2UnClampedPow
(MaterialFloat2 X, MaterialFloat2 Y); MaterialFloat3UnClampedPow(MaterialFloat3 X,
MaterialFloat3 Y); MaterialFloat4UnClampedPow(MaterialFloat4 X, MaterialFloat4 Y);
MaterialFloatClampedPow(MaterialFloat X,MaterialFloat Y); MaterialFloat2ClampedPow
(MaterialFloat2 X,MaterialFloat2 Y); MaterialFloat3ClampedPow(MaterialFloat3 X,MaterialFloat3
Y); MaterialFloat4ClampedPow(MaterialFloat4 X,MaterialFloat4 Y);
MaterialFloatPositiveClampedPow(MaterialFloat X,MaterialFloat Y); MaterialFloat2
PositiveClampedPow(MaterialFloat2 X,MaterialFloat2 Y); MaterialFloat3PositiveClampedPow
(MaterialFloat3 X,MaterialFloat3 Y); MaterialFloat4PositiveClampedPow(MaterialFloat4
X,MaterialFloat4 Y); uintReverseBits32(uint bits);

uintReverseBitsN(uint Bitfield,constuint BitCount);

//math
float Square(floatx); float2Square(float2 x); float3Square
(float3 x); float4Square(float4 x); float Pow2(floatx);
float2Pow2(float2 x); float3Pow2(float3 x); float4Pow2
(float4 x); float Pow3(floatx); float2Pow3(float2 x);
float3Pow3(float3 x); float4Pow3(float4 x); float Pow4(
floatx); float2Pow4(float2 x); float3Pow4(float3 x);
float4Pow4(float4 x); float Pow5(floatx); float2Pow5
(float2 x); float3Pow5(float3 x); float4Pow5(float4 x);
float Pow6(floatx); float2Pow6(float2 x); float3Pow6
(float3 x); float4Pow6(float4 x); MaterialFloatAtanFast
(MaterialFloat x);

//Derivation
float DDX(floatInput);
float2 DDX(float2 Input);
float3 DDX(float3 Input);
float4 DDX(float4 Input);
float DDY(floatInput);
float2 DDY(float2 Input);
float3 DDY(float3 Input);
float4 DDY(float4 Input);

//Data encoding and decoding
MaterialFloat EncodeLightAttenuation(MaterialFloat InColor);

MaterialFloat4 EncodeLightAttenuation(MaterialFloat4 InColor);
MaterialFloat4 RGBTEncode(MaterialFloat3
MaterialFloat3 RGBTDecode(MaterialFloat4 Encodedre); RGBT);
MaterialFloat4RGBMEncode(MaterialFloat3 Color); MaterialFloat4RGBMEncodeFast(MaterialFloat3 Color); MaterialFloat3RGBMDecode(MaterialFloat4 rgbm, MaterialFloat MaxValue); MaterialFloat3 RGBMDecode(MaterialFloat4 rgbm);

MaterialFloat4RGBTEncode8BPC(MaterialFloat3 Color, MaterialFloat Range); MaterialFloat3RGBTDecode8BPC (MaterialFloat4 RGBT, MaterialFloat Range); uintDecodeRTWriteMask(float2 ScreenPos, Texture2D<uint> RTWriteMaskTexture, uint NumEncodedTextures);

float2 EncodeVelocityToTexture(float2 In); DecodeVelocityFromTexture
float2 (float2 In); DecodePackedTwoChannelValue(float2 PackedHeight);
float DecodeHeightValue(floatInValue); DecodePackedHeight(float2
float PackedHeight);
float

//Texture Sampling
aterialFloat4Texture1DSample(Texture1D Tex, SamplerState Sampler,floatUV); MaterialFloat4Texture2DSample (Texture2D Tex, SamplerState Sampler, float2 UV); MaterialFloatTexture2DSample_A8(Texture2D Tex, SamplerState Sampler, float2 UV); MaterialFloat4Texture3DSample(Texture3D Tex, SamplerState Sampler, float3 UV); MaterialFloat4 TextureCubeSample(TextureCube Tex, SamplerState Sampler, float3 UV); MaterialFloat4Texture2DArraySample (Texture2DArray Tex, SamplerState Sampler, float3 UV); MaterialFloat4Texture1DSampleLevel(Texture1D Tex, SamplerState Sampler,floatUV, MaterialFloat Mip);

MaterialFloat4Texture2DSampleLevel(Texture2D Tex, SamplerState Sampler, float2 UV, MaterialFloat Mip);

MaterialFloat4Texture2DSampleBias(Texture2D Tex, SamplerState Sampler, float2 UV, MaterialFloat MipBias);

MaterialFloat4Texture2DSampleGrad(Texture2D Tex, SamplerState Sampler, float2 UV, MaterialFloat2 DDX, MaterialFloat2 DDY);
MaterialFloat4Texture3DSampleLevel(Texture3D Tex, SamplerState Sampler, float3 UV, MaterialFloat Mip);

MaterialFloat4Texture3DSampleBias(Texture3D Tex, SamplerState Sampler, float3 UV, MaterialFloat MipBias);

MaterialFloat4Texture3DSampleGrad(Texture3D Tex, SamplerState Sampler, float3 UV, MaterialFloat3 DDX, MaterialFloat3 DDY);
MaterialFloat4TextureCubeSampleLevel(TextureCube Tex, SamplerState Sampler, float3 UV, MaterialFloat Mip);

MaterialFloatTextureCubeSampleDepthLevel(TextureCube TexDepth, SamplerState Sampler, float3 UV, MaterialFloat Mip);
MaterialFloat4TextureCubeSampleBias(TextureCube Tex, SamplerState Sampler, float3 UV, MaterialFloat MipBias);

MaterialFloat4TextureCubeSampleGrad(TextureCube Tex, SamplerState Sampler, float3 UV, MaterialFloat3 DDX, MaterialFloat3 DDY);
MaterialFloat4TextureExternalSample(TextureExternal Tex, SamplerState Sampler, float2 UV);

MaterialFloat4TextureExternalSampleGrad(TextureExternal Tex, SamplerState Sampler, float2 UV, ...);

MaterialFloat4TextureExternalSampleLevel(TextureExternal Tex, SamplerState Sampler, float2 UV, MaterialFloat Mip);
MaterialFloat4Texture1DSample_Decal(Texture1D Tex, SamplerState Sampler,floatUV); MaterialFloat4 Texture2DSample_Decal(Texture2D Tex, SamplerState Sampler, float2 UV); MaterialFloat4Texture3DSample_Decal (Texture3D Tex, SamplerState Sampler, float3 UV); MaterialFloat4TextureCubeSample_Decal(TextureCube Tex, SamplerState Sampler, float3 UV); MaterialFloat4TextureExternalSample_Decal(TextureExternal Tex, SamplerState Sampler, float2 UV);

```
MaterialFloat4Texture2DArraySampleLevel(Texture2DArray Tex, SamplerState Sampler, float3 UV, MaterialFloat Mip);

MaterialFloat4Texture2DArraySampleBias(Texture2DArray Tex, SamplerState Sampler, float3 UV, MaterialFloat
MipBias);
MaterialFloat4Texture2DArraySampleGrad(Texture2DArray Tex, SamplerState Sampler, float3 UV, ...);



//Special sampling, noise
float4PseudoVolumeTexture(Texture2D Tex, SamplerState TexSampler, float3 inPos, float2 xysize,floatnumframes, ...);

float AntialiasedTextureMask(Texture2D Tex, SamplerState Sampler, float2 UV,float ThresholdConst,intChannel );

float Noise3D_Multiplexer(intFunction, float3 Position,intQuality,boolbTiling,float RepeatSize);

MaterialFloatMaterialExpressionNoise(float3 Position,floatScale,intQuality,int Function, ...);

MaterialFloat4MaterialExpressionVectorNoise(MaterialFloat3 Position,intQuality,int Function,boolbTiling,float
TileSize);


//Lighting calculation
MaterialFloatPhongShadingPow(MaterialFloat X, MaterialFloat Y);


//Data conversion
float    ConvertTangentUnormToSnorm8(floatInput);
float2     ConvertTangentUnormToSnorm8(float2          Input);
float3     ConvertTangentUnormToSnorm8(float3          Input);
float4     ConvertTangentUnormToSnorm8(float4          Input);
float    ConvertTangentUnormToSnorm16(float          Input);
float2     ConvertTangentUnormToSnorm16(float2          Input);
float3     ConvertTangentUnormToSnorm16(float3          Input);
float4     ConvertTangentUnormToSnorm16(float4          Input);
float    ConvertTangentSnormToUnorm8(floatInput);
float2     ConvertTangentSnormToUnorm8(float2          Input);
float3     ConvertTangentSnormToUnorm8(float3          Input);
float4     ConvertTangentSnormToUnorm8(float4          Input);
float    ConvertTangentSnormToUnorm16(float          Input);
float2     ConvertTangentSnormToUnorm16(float2          Input);
float3     ConvertTangentSnormToUnorm16(float3          Input);
float4     ConvertTangentSnormToUnorm16(float4          Input);


//Coordinate and space conversion
float2CalcScreenUVFromOffsetFraction(float4 ScreenPosition, float2 OffsetFraction); float4
           GetPerPixelLightAttenuation(float2 UV);
float    ConvertFromDeviceZ(floatDeviceZ); ConvertToDeviceZ(
float    floatSceneDepth);
float2     ScreenPositionToBufferUV(float4 ScreenPosition); SvPositionToBufferUV
float2     (float4 SvPosition); SvPositionToTranslatedWorld(float4 SvPosition);
float3     SvPositionToResolvedTranslatedWorld(float4 SvPosition);
float3     SvPositionToWorld(float4 SvPosition); SvPositionToScreenPosition(float4
float3     SvPosition); SvPositionToResolvedScreenPosition(float4
float4     SvPositionToViewportUV(float4 BufferUVToViewportUV(float2
float4     ViewportUVToBufferUV(float2 ViewportUVToScreenPos(flSovaPt2o sition);
float2
float2     ScreenPosToViewportUV(float2              SvPosition);
float2                                          BufferUV);
float2                                          ViewportUV);
float2
ViewportUV);      ScreenPos);      float3ScreenToViewPos(float2
ViewportUV,floatSceneDepth);
```

```
MaterialFloat2ScreenAlignedPosition( float4 ScreenPosition ); MaterialFloat2
ScreenAlignedUV( MaterialFloat2 UV ); MaterialFloat2GetViewportCoordinates
(MaterialFloat2                                    InFragmentCoordinates);

MaterialFloat3      TransformTangentVectorToWorld(MaterialFloat3x3          TangentToWorld,
MaterialFloat3      InTangentVector);
MaterialFloat3      TransformWorldVectorToTangent(MaterialFloat3x3          TangentToWorld,
MaterialFloat3      InWorldVector);
float3TransformWorldVectorToView(float3                 InTangentVector);
void GenerateCoordinateSystem(float3 ZAxis, out float3 XAxis, out float3 YAxis);

//Geometry interaction: intersection, distance, etc.
float2LineBoxIntersect(float3 RayOrigin, float3 RayEnd, float3 BoxMin, float3 BoxMax); MaterialFloat
ComputeDistanceFromBoxToPoint(MaterialFloat3 Mins, MaterialFloat3 Maxs, MaterialFloat3
                InPoint);
MaterialFloat       ComputeSquaredDistanceFromBoxToPoint(MaterialFloat3 BoxCenter,
MaterialFloat3 BoxExtent, MaterialFloat3 InPoint);
float ComputeDistanceFromBoxToPointInside(float3 BoxCenter, float3 BoxExtent, float3 InPoint);

bool RayHitSphere(float3 RayOrigin, float3 UnitRayDirection, float3 SphereCenter,float SphereRadius);

bool RaySegmentHitSphere(float3 RayOrigin, float3 UnitRayDirection,floatRayLength, float3 SphereCenter,float
SphereRadius);
float2RayIntersectSphere(float3 RayOrigin, float3 RayDirection, float4 Sphere); MaterialFloatGetBoxPushout
(MaterialFloat3 Normal,MaterialFloat3 Extent);

//Drawing interface
void DrawRectangle(in float4 InPosition, in float2 InTexCoord, out float4 OutPosition, out float2 OutTexCoord);

void DrawRectangle(in float4 InPosition, in float2 InTexCoord, out float4 OutPosition, out float4 OutUVAndScreenPos);

void DrawRectangle(in float4 InPosition, out float4 OutPosition);

(.....)
```

From this, we can see that the common shader module encapsulates a large number of basic types, interfaces, variables, etc. The above only shows some interfaces, and the implementation code is removed for simplicity. If you want to see the implementation, please check the UE source code yourself.

5.2.3.3 Definitions.ush

This module mainly predefines some common macros to prevent syntax errors when other modules reference them. Some macros are as follows:

```
#ifndef    MATERIAL_TWOSIDED
#define        MATERIAL_TWOSIDED                                    0
#endif

#ifndef    MATERIAL_TANGENTSPACENORMAL
#define        MATERIAL_TANGENTSPACENORMAL                          0
#endif

#ifndef    MATERIAL_TWOSIDED_SEPARATE_PASS
#define        MATERIAL_TWOSIDED_SEPARATE_PASS                      0
```

```
#endif

#ifndef     MATERIALBLENDING_MASKED
#define     MATERIALBLENDING_MASKED                                 0
#endif

#ifndef     MATERIALBLENDING_TRANSLUCENT
#define     MATERIALBLENDING_TRANSLUCENT                        0
#endif

#ifndef     TRANSLUCENT_SHADOW_WITH_MASKED_OPACITY
#define     TRANSLUCENT_SHADOW_WITH_MASKED_OPACITY                  0
#endif

#ifndef     MATERIAL_SHADINGMODEL_DEFAULT_LIT
#define     MATERIAL_SHADINGMODEL_DEFAULT_LIT                   0
#endif

#ifndef     MATERIAL_SHADINGMODEL_SUBSURFACE
#define     MATERIAL_SHADINGMODEL_SUBSURFACE                    0
#endif

#ifndef     MATERIAL_SHADINGMODEL_UNLIT
#define         MATERIAL_SHADINGMODEL_UNLIT                             0
#endif

#ifndef     MATERIAL_SINGLE_SHADINGMODEL
#define         MATERIAL_SINGLE_SHADINGMODEL                    0
#endif

#ifndef     HAS_PRIMITIVE_UNIFORM_BUFFER
#define         HAS_PRIMITIVE_UNIFORM_BUFFER                    0
#endif

#ifndef     GBUFFER_HAS_VELOCITY
#define     GBUFFER_HAS_VELOCITY                                0
#endif

#ifndef     GBUFFER_HAS_TANGENT
#define     GBUFFER_HAS_TANGENT                                 0
#endif

#definePC_D3D                                                   SM5_PROFILE

(.....)
```

5.2.3.4 ShadingCommon.ush

This module mainly defines all the shading models of materials and provides a small number of related tool class interfaces:

```
//Material shading model,    Each type has a corresponding lighting algorithm and process.
#define     SHADINGMODELID_UNLIT                0
#define     SHADINGMODELID_DEFAULT_LIT              1
#define     SHADINGMODELID_SUBSURFACE               2
#define     SHADINGMODELID_PREINTEGRATED_SKIN       3
```

```
#define SHADINGMODELID_CLEAR_COAT 4
#define SHADINGMODELID_SUBSURFACE_PROFILE 5
#define SHADINGMODELID_TWOSIDED_FOLIAGE 6
#define SHADINGMODELID_HAIR 7
#define SHADINGMODELID_CLOTH 8
#define SHADINGMODELID_EYE 9
#define SHADINGMODELID_SINGLELAYERWATER 10
#define SHADINGMODELID_THIN_TRANSLUCENT 11
#define SHADINGMODELID_NUM 12
#define SHADINGMODELID_MASK 0xF // ShadingModelIDOnly occupiedGBufferof
// Apart fromShadingsModelIDOther4bitFor other purposes.
#define HAS_ANISOTROPY_MASK (1 << 4)
#define SKIP_PRECSHADOW_MASK (1 << 5)
#define ZERO_PRECSHADOW_MASK (1 << 6)
#define SKIP_VELOCITY_MASK (1 << 7)

// HairReflectivity Component(R, TT, TRT, Local Scattering, Global Scattering, Multi Scattering,...)
#define HAIR_COMPONENT_R 0x1u
#define HAIR_COMPONENT_TT 0x2u
#define HAIR_COMPONENT_TRT 0x4u
#define HAIR_COMPONENT_LS 0x8u
#define HAIR_COMPONENT_GS 0x10u
#define HAIR_COMPONENT_MULTISCATTER 0x20u
#define HAIR_COMPONENT_TT_MODEL 0x40u

//Shading model debug colors.
float3GetShadingModelColor(uint ShadingModelID);

//Reflection from non-conductorsF0.
float DielectricSpecularToF0(float Specular)
{
    return0.08f* Specular;
}

//Non-conductiveF0Convert toIOR(refractive index).
float DielectricF0ToIor(floatF0)
{
    return2.0f/ (1.0f-sqrt(F0)) -1.0f;
}

//Non-conductiveIOR(Refractive index) converted toF0.
float DielectricIorToF0(floatIor) {
    const floatF0Sqrt = (Ior-1)/(Ior+1); const floatF0 =
    F0Sqrt*F0Sqrt; returnF0;
}

//Calculate the surface of an objectF0.
float3ComputeF0(floatSpecular, float3 BaseColor,floatMetallic)
{
    returnlerp(DielectricSpecularToF0(Specular).xxx, BaseColor, Metallic.xxx);
}
```

It should be noted that the UE's default ShadingModelID only occupies 4 bits, a maximum of 16, and the current UE built-in shading model has occupied 13, which means that the maximum number of custom ShadingModels can be 3.

5.2.3.5 BasePassCommon.ush

This module defines some variables, macro definitions, interpolation structures and tool class interfaces of BasePass:

```
//Transparent objectsBasePassDefinition
#if    MATERIALBLENDING_ANY_TRANSLUCENT
       #define    ForwardLightData       TranslucentBasePass.Shared.Forward
       #define    ReflectionStruct       TranslucentBasePass.Shared.Reflection
       #define    PlanarReflectionStruct          TranslucentBasePass.Shared.PlanarReflection
       #define    FogStruct TranslucentBasePass.Shared.Fog
       #define    ActualSSProfilesTexture          TranslucentBasePass.Shared.SSProfilesTexture
//    Opaque objectsBasePassDefinition
#else
       #define    ForwardLightData       OpaqueBasePass.Shared.Forward
       #define    ReflectionStruct       OpaqueBasePass.Shared.Reflection
       #define    PlanarReflectionStruct          OpaqueBasePass.Shared.PlanarReflection
       #define    FogStruct OpaqueBasePass.Shared.Fog ActualSSProfilesTexture
       #define    OpaqueBasePass.Shared.SSProfilesTexture
#endif

//BasePassRelated macro definitions
#undef NEEDS_LIGHTMAP_COORDINATE
#define    NEEDS_LIGHTMAP_COORDINATE              (HQ_TEXTURE_LIGHTMAP    ||LQ_TEXTURE_LIGHTMAP)
#define    TRANSLUCENCY_NEEDS_BASEPASS_FOGGING          (MATERIAL_ENABLE_TRANSLUCENCY_FOGGING      &&
MATERIALBLENDING_ANY_TRANSLUCENT    && !MATERIAL_USES_SCENE_COLOR_COPY)
#defineOPAQUE_NEEDS_BASEPASS_FOGGING                (!MATERIALBLENDING_ANY_TRANSLUCENT      &&
 FORWARD_SHADING)

#defineNEEDS_BASEPASS_VERTEX_FOGGING !MATERIAL_COM(TPRUATNE_SFLOUGC_EPNECRY__PNIXEELD&&S|_B ASEPASS_FOGG
OPAQUE_NEEDS_BASEPASS_FOGGING && PROJECT_VERTEX_FOGGING_FOR_OPAQUE)

#defineNEEDS_BASEPASS_PIXEL_FOGGING MATERIAL_COMP(TURTAEN_FSOLUGC_PEENRC_YP_INXELE D&&S _BASEPASS_FOGGIN
OPAQUE_NEEDS_BASEPASS_FOGGING && !PROJECT_VERTEX_FOGGING_FOR_OPAQUE)


#defineNEEDS_BASEPASS_PIXEL_VOLUMETRIC_FOGGING ||              (MATERIALBLENDING_ANY_TRANSLUCENT
 FORWARD_SHADING)

#defineNEEDS_LIGHTMAP                                    (NEEDS_LIGHTMAP_COORDINATE)

#defineUSES_GBUFFER                                    (FEATURE_LEVEL >= FEATURE_LEVEL_SM4 &&
(MATERIALBLENDING_SOLID || MATERIALBLENDING_MASKED) && !SIMPLE_FORWARD_SHADING && !
FORWARD_SHADING)

(.....)

//The parent class of interpolation structures.
structFSharedBasePassInterpolants {

      //for texture-lightmapped translucency we can pass the vertex fog in its own interpolator
```

```
#if NEEDS_BASEPASS_VERTEX_FOGGING
    float4 VertexFog                    : TEXCOORD7;
#endif

#if !TESSELLATION_SUPPORTED // Note:
    TEXCOORD8 is unused

    #ifUSE_WORLD_POSITION_EXCLUDING_SHADER_OFFSETS float3
    PixelPositionExcludingWPO : TEXCOORD9;
    #endif
#endif

#if TRANSLUCENCY_PERVERTEX_LIGHTING_VOLUME float3
    AmbientLightingVector : TEXCOORD12;
#endif

#ifTRANSLUCENCY_PERVERTEX_LIGHTING_VOLUME &&
 TRANSLUCENCY_LIGHTING_VOLUMETRIC_PERVERTEX_DIRECTIONAL
    float3 DirectionalLightingVector : TEXCOORD13;
#endif

#if TRANSLUCENCY_PERVERTEX_FORWARD_SHADING float3
    VertexDiffuseLighting : TEXCOORD12;
#endif

#if PRECOMPUTED_IRRADIANCE_VOLUME_LIGHTING
    #if TRANSLUCENCY_LIGHTING_VOLUMETRIC_PERVERTEX_NONDIRECTIONAL
        float3 VertexIndirectAmbient : TEXCOORD14;
    #elifTRANSLUCENCY_LIGHTING_VOLUMETRIC_PERVERTEX_DIRECTIONAL float4
        VertexIndirectSH[3] : TEXCOORD14;
    #endif
#endif

#if WRITES_VELOCITY_TO_GBUFFER
    // .xy is clip position, pre divide by w; .w is clip W; .z is 0 or 1 to mask out the velocity output

    float4 VelocityPrevScreenPosition : VELOCITY_PREV_POS;
    #if WRITES_VELOCITY_TO_GBUFFER_USE_POS_INTERPOLATOR
        float4 VelocityScreenPosition : VELOCITY_POS;
    #endif
#endif
};

#if TESSELLATION_SUPPORTED
    //VS -> PSThe interpolation structure of .
    structFBasePassInterpolantsVSToPS: FSharedBasePassInterpolants {


        // Note: TEXCOORD8 is unused

        #ifUSE_WORLD_POSITION_EXCLUDING_SHADER_OFFSETS float3
        PixelPositionExcludingWPO : TEXCOORD9;
        #endif
    };
    // VS -> DS (domain shader)The interpolation structure of .
    structFBasePassInterpolantsVSToDS: FSharedBasePassInterpolants {


        #ifUSE_WORLD_POSITION_EXCLUDING_SHADER_OFFSETS float3
        WorldPositionExcludingWPO : TEXCOORD9;
```

```
        #endif
    };
#else
    #defineFBasePassInterpolantsVSToPS FSharedBasePassInterpolants
#endif

//Sampler package.
#if  SUPPORTS_INDEPENDENT_SAMPLERS
    #define  SharedAmbientInnerSampler      View.SharedBilinearClampedSampler
    #define  SharedAmbientOuterSampler      View.SharedBilinearClampedSampler
    #define  SharedDirectionalInnerSampler      View.SharedBilinearClampedSampler
    #define SharedDirectionalOuterSampler      View.SharedBilinearClampedSampler
#else
    #define SharedAmbientInnerSampler
TranslucentBasePass.TranslucencyLightingVolumeAmbientInnerSampler
    #defineSharedAmbientOuterSampler
TranslucentBasePass.TranslucencyLightingVolumeAmbientOuterSampler
    #defineSharedDirectionalInnerSampler
TranslucentBasePass.TranslucencyLightingVolumeDirectionalInnerSampler
    #defineSharedDirectionalOuterSampler
TranslucentBasePass.TranslucencyLightingVolumeDirectionalOuterSampler
#endif

//Tool interface.
void ComputeVolumeUVs(float3 WorldPosition, float3 LightingPositionOffset, out float3 InnerVolumeUVs, out float3
OuterVolumeUVs, outfloatFinalLerpFactor);
float4GetAmbientLightingVectorFromTranslucentLightingVolume(float3 InnerVolumeUVs, float3 OuterVolumeUVs,float
FinalLerpFactor);
float3GetDirectionalLightingVectorFromTranslucentLightingVolume(float3 InnerVolumeUVs, float3 OuterVolumeUVs,
floatFinalLerpFactor);
```

5.2.3.6 BRDF.ush

The bidirectional reflectance distribution function module provides many basic lighting algorithms and related auxiliary interfaces:

```
//BxDFContext, which stores dot products between commonly used
vectors. structBxDFContext
{
    float    NoV;
    float    NoL;
    float    VoL;
    float    NoH;
    float    VoH;
    float    XoV;
    float    XoL;
    float    XoH;
    float    YoV;
    float    YoL;
    float    YoH;
};

//initializationBxFContext
void Init( inout BxDFContext Context, half3 N, half3 V, half3 L );

//initializationBxFContext, includingX, YTwo vectors (the tangent and the cross product of the tangent and the normal).
```

```
void Init( inout BxDFContext Context, half3 N, half3 X, half3 Y, half3 V, half3 L ) {

	Context.NoL = dot(N, L);
	Context.NoV = dot(N, V);
	Context.VoL = dot(V, L);
	floatInvLenH = rsqrt(2+2* Context.VoL );
	Context.NoH = saturate( ( Context.NoL + Context.NoV ) * InvLenH ); Context.VoH =
	saturate( InvLenH + InvLenH * Context.VoL );


	Context.XoV = dot(X, V);
	Context.XoL = dot(X, L);
	Context.XoH = (Context.XoL + Context.XoV) * InvLenH; Context.YoV =
	dot(Y, V);
	Context.YoL = dot(Y, L);
	Context.YoH = (Context.YoL + Context.YoV) * InvLenH;
}

//The largest sphericalNoH.
void SphereMaxNoH( inout BxDFContext Context,floatSinAlpha,boolbNewtonIteration );

//Lambert diffuse reflectance.
float3Diffuse_Lambert( float3 DiffuseColor ) {

	returnDiffuseColor * (1/PI);
}

// BurleyDiffuse reflection.
float3Diffuse_Burley( float3 DiffuseColor,floatRoughness,floatNoV,floatNoL,float VoH )

{
	floatFD90 =0.5+2* VoH * VoH * Roughness; floatFdV =1+ (FD90 -
	1) * Pow5(1- NoV ); floatFdL =1+ (FD90 -1) * Pow5(1- NoL );
	returnDiffuseColor * ( (1/ PI) * FdV * FdL );

}

// OrenNayarDiffuse reflection.
float3Diffuse_OrenNayar( float3 DiffuseColor,floatRoughness,floatNoV,floatNoL, floatVoH )

{
	floata = Roughness * Roughness; floats = a;// /
	( 1.29 + 0.5 * a ); floats2 = s * s;

	floatVoL =2* VoH * VoH -1; floatCosri = VoL - NoV * // double angle identity
	NoL; floatC1 =1-0.5* s2 / (s2 +0.33);

	floatC2 =0.45* s2 / (s2 +0.09) * Cosri * ( Cosri >=0? rcp( max( NoL, NoV ) ) :1
) ;
	returnDiffuseColor / PI * ( C1 + C2 ) * (1+ Roughness *0.5);
}

// GotandaDiffuse reflection.
float3Diffuse_Gotanda( float3 DiffuseColor,floatRoughness,floatNoV,floatNoL,float VoH );


// BlinnNormal distribution.
float D_Blinn(floata2,floatNoH ) {
```

```
        floatn =2/a2 -2;
        return(n+2) / (2*PI) * PhongShadingPow(NoH, n);                                    // 1 mad, 1 exp, 1 mul, 1
log
}

// BeckmannNormal distribution.
float D_Beckmann(floata2,floatNoH ) {

        floatNoH2 = NoH * NoH;
        return exp( (NoH2 -1) / (a2 * NoH2) ) / ( PI * a2 * NoH2 * NoH2 );
}

//GGXNormal distribution.
float D_GGX(floata2,floatNoH ) {

        floatd = (NoH * a2 - NoH ) * NoH +1; returna2 /                 // 2 mad
        ( PI*d*d );                                                     // 4 mul, 1 rcp
}

//GGXAnisotropic normal distribution.
float D_GGXaniso(floatax,floatay,floatNoH,floatXoH,floatYoH ) {

        floata2 = ax * ay;
        float3 V = float3(ay * XoH, ax * YoH, a2 * NoH); floatS = dot(V, V);


        return(1.0f/ PI) * a2 * Square(a2 / S);
}

//Inverse normal distribution function.
float D_InvBlinn(floata2,floatNoH ) {

        floatA =4;
        floatCos2h = NoH * NoH; float
        Sin2h =1- Cos2h;
        //return rcp( PI * (1 + A*m2) ) * ( 1 + A * ClampedPow( Sin2h, 1 / m2 - 1 ) ); returnrcp(PI * (1+ A*a2) ) * (1+
        A*exp( -Cos2h / a2 ) );
}

float D_InvBeckmann(floata2,floatNoH ) {

        floatA =4;
        floatCos2h = NoH * NoH; floatSin2h =
        1- Cos2h; floatSin4h = Sin2h * Sin2h;

        returnrcp(PI * (1+ A*a2) * Sin4h ) * ( Sin4h + A *exp( -Cos2h / (a2 * Sin2h) ) );
}

float D_InvGGX(floata2,floatNoH ) {

        floatA =4;
        floatd = (NoH - a2 * NoH ) * NoH + a2;
        returnrcp(PI * (1+ A*a2) ) * (1+4* a2*a2 / ( d*d ) );
}

//      The following is the visibility function, often referred to as geometry in papers (G)Item.
//      Implicit visibility function.
float     Vis_Implicit()
```

```
{
    return     0.25;
}

// NeumannVisibility function.
float Vis_Neumann(floatNoV,floatNoL ) {

    return1/ (4* max( NoL, NoV ) );
}

// KelemenVisibility function.
float Vis_Kelemen(floatVoH ) {

    // constant to prevent NaN returnrcp(4* VoH *
    VoH +1e-5);
}

// SchlickVisibility function.
float Vis_Schlick(floata2,floatNoV,floatNoL ) {

    floatk =sqrt(a2) *0.5;
    floatVis_SchlickV = NoV * (1- k) + k; floatVis_SchlickL = NoL * (1-
    k) + k; return0.25/ (Vis_SchlickV * Vis_SchlickL);

}

// SmithVisibility function.
float Vis_Smith(floata2,floatNoV,floatNoL ) {

    floatVis_SmithV = NoV +sqrt(NoV * (NoV - NoV * a2) + a2 ); floatVis_SmithL = NoL +
    sqrt(NoL * (NoL - NoL * a2) + a2 ); returnrcp(Vis_SmithV * Vis_SmithL);

}

// SmithJointApproximate visibility function.
float Vis_SmithJointApprox(floata2,floatNoV,floatNoL ) {

    floata =sqrt(a2);
    floatVis_SmithV = NoL * ( NoV * (1- a ) + a ); floatVis_SmithL = NoV *
    ( NoL * (1- a ) + a ); return0.5*rcp(Vis_SmithV + Vis_SmithL);

}

// SmithJointVisibility function.
float Vis_SmithJoint(floata2,floatNoV,floatNoL)

    floatVis_SmithV = NoL *sqrt(NoV * (NoV - NoV * a2) + a2); floatVis_SmithL = NoV
    *sqrt(NoL * (NoL - NoL * a2) + a2); return0.5*rcp(Vis_SmithV + Vis_SmithL);

}

// SmithJointAnisotropic visibility function.
float Vis_SmithJointAniso(floatax,floatay,floatNoV,floatNoL,floatXoV,floatXoL, floatYoV,floatYoL)


    floatVis_SmithV = NoL * length(float3(ax * XoV, ay * YoV, NoV)); floatVis_SmithL = NoV *
    length(float3(ax * XoL, ay * YoL, NoL)); return0.5*rcp(Vis_SmithV + Vis_SmithL);
```

```
}

//Cloth visibility distribution function.
float Vis_Cloth(floatNoV,floatNoL ) {

	returnrcp(4* (NoL + NoV - NoL * NoV ) );
}

//No Fresnel function.
float3F_None( float3 SpecularColor ) {

	returnSpecularColor;
}

// SchlickFresnel function.
float3F_Schlick( float3 SpecularColor,floatVoH ) {

	floatFc = Pow5(1- VoH );					// 1 sub, 3 mul
	returnsaturate(50.0* SpecularColor.g ) * Fc + (1- Fc) * SpecularColor;
}

//The complete Fresnel function.
float3F_Fresnel( float3 SpecularColor,floatVoH ) {

	float3 SpecularColorSqrt =sqrt( clamp( float3(0,0,0), float3(0.99,0.99,0.99), SpecularColor ) );

	float3 n = (1+ SpecularColorSqrt ) / (1- SpecularColorSqrt ); float3 g =sqrt( n*n + VoH*VoH
	-1);
	return0.5* Square( (g - VoH) / (g + VoH) ) * (1+ Square( ((g+VoH)*VoH -1) / ((g-VoH)*VoH +1) ) ); }



//EnvironmentalBRDF
void ModifyGGXAnisotropicNormalRoughness(float3 WorldTangent,floatAnisotropy, inout floatRoughness, inout
float3 N, float3 V);
void GetAnisotropicRoughness(floatAlpha,floatAnisotropy, outfloatax, outfloatay); half3EnvBRDF( half3 SpecularColor,
half Roughness, half NoV ); half3EnvBRDFApprox( half3 SpecularColor, half Roughness, half NoV ); half
EnvBRDFApproxNonmetal(half Roughness, half NoV);

void EnvBRDFApproxFullyRough(inout half3 DiffuseColor, inout half3 SpecularColor); void
EnvBRDFApproxFullyRough(inout half3 DiffuseColor, inout half SpecularColor);
```

The vectors and`void Init( inout BxDFContext Context, half3 N, half3 X, half3 Y, half3 V, half3 L )` in the above interface represent the tangent and the perpendicular vector of the tangent and normal in the world space. Use cases:`X` `Y`

```
half3 X = GBuffer.WorldTangent; half3 Y =
normalize(cross(N, X)); Init(Context, N, X, Y, V,
L);
```

Let's review the Cook-Torrance BRDF formula:

$$fcthethek-ttherrancand=\frac{D(h)\cdot F(l,h)\cdot G(l,v,h)}{4(n\cdot l)(n\cdot v)}$$

In the above code,D_the interface at the beginning corresponds to the Cook-Torrance item.D
For more detailed analysis, please refer to my other article on PBR,Learn the Principles and Implementation of PBR from the Shallow to the Deep and related chapters:3.1.4 Bidirectional Reflectance Distribution Function (BRDF) .F_ F Vis_ G

5.2.3.7 VertexFactoryCommon.ush

This module mainly defines auxiliary interfaces related to vertex transformation:

```
//Vertex Transformation
float3TransformLocalToWorld(float3 LocalPosition, uint PrimitiveId); float4
TransformLocalToWorld(float3 LocalPosition);
float4TransformLocalToTranslatedWorld(float3 LocalPosition, uint PrimitiveId); float4
TransformLocalToTranslatedWorld(float3 LocalPosition); float3RotateLocalToWorld(float3 LocalDirection,
uint PrimitiveId); float3
        RotateLocalToWorld(float3       LocalDirection);
float3  RotateWorldToLocal(float3       WorldDirection);

//The following two interfaces andDeferredShadingCommon.ushofUnitVectorToOctahedronandOctahedronToUnitVectorThe names are different, but the functions they implement are the same.
float2UnitToOct( float3 N ); float3
OctToUnit( float2 Oct );

(.....)
```

5.2.3.8 BasePassVertexCommon.ush

This module defines some common structures and macros for BasePass:

```
#include"Common.ush"

#ifMATERIALBLENDING_TRANSLUCENT || MATERIALBLENDING_ADDITIVE || MATERIALBLENDING_MODULATE
    #defineSceneTexturesStruct TranslucentBasePass.SceneTextures
#endif

#include   "/Engine/Generated/Material.ush"
#include   "BasePassCommon.ush"
#include   "/Engine/Generated/VertexFactory.ush"

#if  NEEDS_BASEPASS_VERTEX_FOGGING
    #include   "HeightFogCommon.ush"
    #if BASEPASS_ATMOSPHERIC_FOG
        #include"AtmosphereCommon.ush"
    #endif
#endif

//fromVSSend toPS/DSThe structure
of . structFBasePassVSToPS
{
    FVertexFactoryInterpolantsVSToPS FBasePassFlanctteorrpyoInlatenrtpsVoSlaTnotPsS;
    BasePassInterpolants; float4 Position : SV_POSITION;

};

#ifUSING_TESSELLATION
```

```
    structFBasePassVSToDS
    {
        FVertexFactoryInterpolantsVSToDS FactoryInterpolants;
        FBasePassInterpolantsVSToDS BasePassInterpolants; float4 Position :
        VS_To_DS_Position;
        OPTIONAL_VertexID_VS_To_DS
    };

    #define    FBasePassVSOutput FBasePassVSToDS
    #define    VertexFactoryGetInterpolants        VertexFactoryGetInterpolantsVSToDS
    #define    FPassSpecificVSToDS                 FBasePassVSToDS
    #define    FPassSpecificVSToPS       FBasePassVSToPS
#else
    #define    FBasePassVSOutput FBasePassVSToPS VertexFactoryGetInterpolants
    #define    VertexFactoryGetInterpolantsVSToPS
#endif
```

5.2.3.9 ShadingModels.ush

This module mainly includes types and auxiliary interfaces related to shading models and lighting calculations:

```
//Area light data.
structFAreaLight
{
    float           SphereSinAlpha;
    float           SphereSinAlphaSoft;
    float           LineCosSubtended;

    float3            FalloffColor;

    FRect           Rect;
    FRectTexture    Texture;
    bool           bIsRect;
};

//Direct light data.
structFDirectLighting
{
    float3      Diffuse;
    float3      Specular;
    float3      Transmission;
};

//Shadow data, used to store shadow casting results.
structFShadowTerms
{
    float     floatSurfaceShadow;
    FHairTransmittanceDataTransmittanceShadow;
                TransmissionThickness;
                                Hair Transmittance;
};

//----Lighting auxiliary interface---- //
Energy normalization.
floatEnergyNormalization( inoutfloata2,floatVoH, FAreaLight AreaLight );
```

```
//GGXHighlight.
float3SpecularGGX(floatRoughness,floatAnisotropy, float3 SpecularColor, ...); //GGXDouble highlight.

float3DualSpecularGGX(floatAverageRoughness,floatLobe0Roughness,float Lobe1Roughness,...);

bool IsAreaLight(FAreaLight AreaLight);
float New_a2(floata2,floatSinAlpha,floatVoH ); float ApproximateHG(
floatcosJ,floatg);
float3CalcThinTransmission(floatNoL,floatNoV, FGBufferData GBuffer);
void GetProfileDualSpecular(FGBufferData GBuffer, outfloatAverageToRoughness0, ...); bool IsAreaLight(FAreaLight
AreaLight);
float New_a2(floata2,floatSinAlpha,floatVoH );


//Refraction related.
float RefractBlend(floatVoH,floatEta); float
RefractBlendClearCoatApprox(floatVoH); float3Refract(float3 V,
float3 H,floatEta); BxDFContextRefractClearCoatContext
(BxDFContext                                                    Context);

// ---- Shading Modelillumination----
float3SimpleShading( float3 DiffuseColor, float3 SpecularColor,floatRoughness, float3 L, float3 V, half3 N );

FDirectLightingDefaultLitBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L, ...); FDirectLightingHairBxDF
(FGBufferData GBuffer, half3 N, half3 V, half3 L, ...); FDirectLightingClearCoatBxDF(FGBufferData GBuffer, half3 N,
half3 V, half3 L, ... ); FDirectLightingSubsurfaceProfileBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L, . . . );



FDirectLightingClothBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L, ... ); FDirectLightingSubsurfaceBxDF
(FGBufferData GBuffer, half3 N, half3 V, half3 L, ... ); FDirectLightingTwoSidedBxDF(FGBufferData GBuffer, half3 N,
half3 V, half3 L, ... ); FDirectLightingEyeBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L, ); FDirectLighting
PreintegratedSkinBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L, . . .


    ) ;
//    Integrated lighting, will be based onShadingModelIDCall the different interfaces above.
FDirectLightingIntegrateBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L, ... ); FDirectLightingEvaluateBxDF
(FGBufferData GBuffer, half3 N, half3 V, half3 L, ... );
```

5.2.3.10 DeferredShadingCommon.ush

This module mainly defines the common interfaces, macros, variables, and types of delayed lighting shading:

```
//Color space conversion
float3RGBToYCoCg( float3 RGB ); float3
YCoCgToRGB( float3 YCoCg );

//Vector compression and decompression (unit vectors,
octahedrons, semi-octahedrons) float2UnitVectorToOctahedron
( float3 N ); float3OctahedronToUnitVector( float2 Oct ); float2
UnitVectorToHemiOctahedron( float3 N ); float3
HemiOctahedronToUnitVector( float2 Oct );


//Data compression and decompression
float3Pack1212To888( float2 x ); float2Pack888To1212
( float3 x ); float Encode71(floatScalar, uint Mask);
```

```
float Decode71(floatScalar, out uint Mask);

//Normal encoding and decoding
float3EncodeNormal( float3 N ); float3
DecodeNormal( float3 N );
void EncodeNormal( inout float3 N, out uint Face ); void
DecodeNormal( inout float3 N, in uint Face );

//color,      Encoding and decoding of subsurface color and other data.
float3    EncodeBaseColor(float3
float3    DecodeBBaasseeCCoololorr);          (float3
float3    EncodeSubsurfaceColor(BflaasaetC3 olor);
float3    EncodeSubsurfaceProfile(float      SubsurfaceColor);
float3    SubsurfaceDensityFromOpacity(float    SubsurfaceProfile);
float     EncodeIndirectIrradiance(float          Opacity);
float     DecodeIndirectIrradiance(float      IndirectIrradiance);
float                                         IndirectIrradiance);
float4EncodeWorldTangentAndAnisotropy(float3 WorldTangent,floatAnisotropy); float
ComputeAngleFromRoughness(floatRoughness,const floatThreshold =0.04f); float
ComputeRoughnessFromAngle(floatAngle,const floatThreshold =0.04f); float AddAngleToRoughness(float
Angle,floatRoughness );
float EncodeShadingModelIdAndSelectiveOutputMask(uint ShadingModelId, uint
SelectiveOutputMask);
uint    DecodeShadingModelId(floatInPackedChannel);
uint DecodeSelectiveOutputMask(floatInPackedChannel);

//Detection interface.
bool   IsSubsurfaceModel(intShadingModel); UseSubsurfaceProfile(int
bool   HasCustomGBufferData(int HasAnSishoatdrionpgyM(inotdSeell)e;ctiveOutputMask);
bool   CastContactShadow(FGBufferDataS GhaBduifnfegrMDoadtae)l;I D);
bool   HasDynamicIndirectShadowCasterRepresentation(FGBufferData
bool
bool                                                          GBufferData);

// GBufferThe structure of the data,    is the largest set of geometric data.
structFGBufferData
{
      float3    WorldNormal;
      float3    WorldTangent;
      float3    DiffuseColor;
      float3    SpecularColor;
      float3    BaseColor;
      float     Metallic;
      float     Specular;
      float4    CustomData;
      float     IndirectIrradiance;
      float4    PrecomputedShadowFactors;
      float     Roughness;
      float     Anisotropy;
      float     GBufferAO;
      uint ShadingModelID;
      uint SelectiveOutputMask;
      float     PerObjectGBufferData;
      float     CustomDepth;
      uint CustomStencil;
      float     Depth;
      float4    Velocity;
      float3    StoredBaseColor;
      float     StoredSpecular;
```

```
    floatStoredMetallic;
};

//Screen space data, includingGBufferand
AO. structFScreenSpaceData {

    // GBuffer (material attributes from forward rendering pass) FGBufferData
    GBuffer;
    floatAmbientOcclusion;
};

// GBufferData operation interface.
void SetGBufferForUnlit(out float4 OutGBufferB);
void EncodeGBuffer(FGBufferData GBuffer, out float4 OutGBufferA, ...); FGBufferData
DecodeGBufferData(float4 InGBufferA, ...);
FGBufferDataGetGBufferDataUint(uint2 PixelPos,boolbGetNormalizedNormal); FScreenSpaceData
GetScreenSpaceData(float2 UV,boolbGetNormalizedNormal) float3
        ExtractSubsurfaceColor(FGBufferData BufferData)
uint    ExtractSubsurfaceProfileInt(FGBufferData BufferData)
uint    GetShadingModelId(float2 UV);
void AdjustBaseColorAndSpecularColorForSubsurfaceProfileLighting(inout float3 BaseColor, . . . );


//Chessboard sampling.
bool    CheckerFromPixelPos(uint2 PixelPos);
bool    CheckerFromSceneColorUV(float2 UVSceneColor);
```

It should be noted that the core structureFGBufferDatais a universal structure that can be used between VS and PS of BasePass and LightingPass, and can also be used in forward and deferred rendering. It is the largest data collection, and some properties are only valid in some cases, see ShadingModelsMaterial.ush for detailsSetGBufferForShadingModel.

5.2.3.11 DeferredLightingCommon.ush

This module defines common interfaces, macros, variables, types, etc. related to deferred lighting.

```
//Single deferred light data (actually also available for forward
rendering). structFDeferredLightData {

    float3    Position;
    float     InvRadius;
    float3    Color;
    float     FalloffExponent;
    float3    Direction;
    float3    Tangent;
    float     SoftSourceRadius;
    float2    SpotAngles;
    float     SourceRadius;
    float     SourceLength;
    float     SpecularScale;
    float     ContactShadowLength;
    float     ContactShadowNonShadowCastingIntensity;
    float2 DistanceFadeMAD;
    float4    ShadowMapChannelMask;
    bool    ContactShadowLengthInWS;
    bool bInverseSquared;
```

```
    bool    bRadialLight;
    bool    bSpotLight;
    bool    bRectLight;
    uint    ShadowedBits;
    float   RectLightBarnCosAngle;
    float   RectLightBarnLength;

    FHairTransmittanceData HairTransmittance;
};

//Simple deferred light data, simulating simple light sources, often used in specific simple shading models to accelerate and limit
feature bases. structFSimpleDeferredLightData {

    float3    Position;
    float     InvRadius;
    float3    Color;
    float     FalloffExponent;
    bool bInverseSquared;
};

//Calculates the fade level based on the light source and camera data.0It is closer than the receding plane.1Farther than the fading far plane.
float DistanceFromCameraFade(floatSceneDepth, FDeferredLightData LightData, float3 WorldPosition, float3
CameraPosition);

//----Shadow related interfaces----
//Shadow ray detection. If no shadow is detected, a negative number is returned. If the ray hits a dynamic shadow caster,bOutHitCastDynamicShadow fortrue.

float ShadowRayCast(float3 RayOriginTranslatedWorld, float3 RayDirection,floatRayLength, intNumSteps,floatStepOffset,
outboolbOutHitCastContactShadow ); //Computes shadows for the specified light source.

void GetShadowTerms(FGBufferData GBuffer, FDeferredLightData LightData, float3 WorldPosition, float3 L,
float4 LightAttenuation,floatDither, inout FShadowTerms Shadow);



//Get the geometry corresponding to the light source.
FRectGetRect(float3 ToLight, FDeferredLightData LightData); FCapsuleLightGetCapsule(float3
ToLight, FDeferredLightData LightData);

//----Lighting calculation interface----
//Get the direct light of the specified light source. The lighting result is split into diffuse and highlight items.
FDeferredLightingSplitGetDynamicLightingSplit(float3 WorldPosition, float3 CameraVector, FGBufferData GBuffer, ...);

//Gets the direct light of the specified light source.
float4GetDynamicLighting(float3 WorldPosition, float3 CameraVector, FGBufferData GBuffer, . . . );

//Gets the direct light of the specified simple light source.
float3GetSimpleDynamicLighting(float3 WorldPosition, float3 CameraVector, float3 WorldNormal, ...);
```

5.2.3.12 ShadingModelsMaterial.ush

Provides an interface for setting GBuffe based on materials and specified parameters:

```
#defineSUBSURFACE_PROFILE_OPACITY_THRESHOLD 1

void SetGBufferForShadingModel(
    in out FGBufferData GBuffer,
```

```
    inconstFMaterialPixelParameters MaterialParameters, const
            float    Opacity,
    const   half3    BaseColor,
    const half       Metallic,
    half             Specular,
    float            Roughness,
    float            Anisotropy,
    float3             SubsurfaceColor,
    float            SubsurfaceProfile,
    const float      Dither,
    const   uint ShadingModel)
{
    GBuffer.WorldNormal      =MaterialParameters.WorldNormal;
    GBuffer.WorldTangent       =MaterialParameters.WorldTangent;
    GBuffer.BaseColor      = BaseColor;
    GBuffer.Metallic       = Metallic;
    GBuffer.Specular       = Specular;
    GBuffer.Roughness      = Roughness;
    GBuffer.Anisotropy       = Anisotropy;
    GBuffer.ShadingModelID          =ShadingModel;

    (......)

#if  MATERIAL_SHADINGMODEL_SUBSURFACE
    else if(ShadingModel == SHADINGMODELID_SUBSURFACE) {

        GBuffer.CustomData.rgb       = EncodeSubsurfaceColor(SubsurfaceColor); Opacity;
        GBuffer.CustomData.a =
    }
#endif
#if  MATERIAL_SHADINGMODEL_PREINTEGRATED_SKIN
    else if(ShadingModel == SHADINGMODELID_PREINTEGRATED_SKIN) {

        GBuffer.CustomData.rgb       = EncodeSubsurfaceColor(SubsurfaceColor); Opacity;
        GBuffer.CustomData.a =
    }
#endif

    (......)
}
```

5.2.3.13 LocalVertexFactoryCommon.ush

The local vertex factory common module defines the data interpolation structure and some auxiliary interfaces of the vertex factory:

```
//Vertex FactoryVS -> PSInterpolation of .
structFVertexFactoryInterpolantsVSToPS {

    TANGENTTOWORLD_INTERPOLATOR_BLOCK

#if  INTERPOLATE_VERTEX_COLOR
    half4 Color : COLOR0;
#endif

#ifUSE_INSTANCING
```

```
        // x = per-instance random, y = per-instance fade out amount, z = hide/show flag, w dither fade
                cutoff
        float4      PerInstanceParams : COLOR1;
#endif

#if    NUM_TEX_COORD_INTERPOLATORS
        float4 TexCoords[(NUM_TEX_COORD_INTERPOLATORS+1)/2]                    :    TEXCOORD0;
#elifUSE_PARTICLE_SUBUVS float4 TexCoords[1] :
  TEXCOORD0;
#endif

#if    NEEDS_LIGHTMAP_COORDINATE
        float4 LightMapCoordinate                :    TEXCOORD4;
#endif

#if    INSTANCED_STEREO
        nointerpolation uint EyeIndex : PACKED_EYE_INDEX;
#endif
#if    VF_USE_PRIMITIVE_SCENE_DATA
        nointerpolation uint PrimitiveId : PRIMITIVE_ID;
        #if NEEDS_LIGHTMAP_COORDINATE
            nointerpolation uint LightmapDataIndex : LIGHTMAP_ID;
        #endif
#endif
#ifVF_STRAND_HAIR || VF_CARDS_HAIR
        nointerpolation uint HairPrimitiveId float2              : HAIR_PRIMITIVE_ID;// Control point ID
        HairPrimitiveUV                                          : HAIR_PRIMITIVE_UV;// U: parameteric
distance between the two surrounding control points. V: parametric distance along the width.

#endif
};

//UV
#ifNUM_TEX_COORD_INTERPOLATORS || USE_PARTICLE_SUBUVS
float2GetUV(FVertexFactoryInterpolantsVSToPS Interpolants,intUVIndex);
void SetUV(inout FVertexFactoryInterpolantsVSToPS Interpolants,intUVIndex, float2 InValue);

#endif

//color
float4GetColor(FVertexFactoryInterpolantsVSToPS Interpolants);
void SetColor(inout FVertexFactoryInterpolantsVSToPS Interpolants, float4 InValue);


//Light map coordinates
void SetLightmapDataIndex(inout FVertexFactoryInterpolantsVSToPS Interpolants, uint LightmapDataIndex);

#ifNEEDS_LIGHTMAP_COORDINATE
void GetLightMapCoordinates(FVertexFactoryInterpolantsVSToPS Interpolants, out float2 LightmapUV0, out float2
LightmapUV1, out uint LightmapDataIndex);
void GetShadowMapCoordinate(FVertexFactoryInterpolantsVSToPS Interpolants, out float2 ShadowMapCoordinate,
out uint LightmapDataIndex);
void SetLightMapCoordinate(inout FVertexFactoryInterpolantsVSToPS Interpolants, float2 InLightMapCoordinate,
float2 InShadowMapCoordinate);
#endif


//Tangent
float4    GetTangentToWorld2(FVertexFactoryInterpolantsVSToPS                Interpolants);
float4    GetTangentToWorld0(FVertexFactoryInterpolantsVSToPS                Interpolants);
```

```
void SetTangents(inout FVertexFactoryInterpolantsVSToPS Interpolants, float3 InTangentToWorld0,
float3 InTangentToWorld2,floatInTangentToWorldSign);

//Graphicsid.
uintGetPrimitiveId(FVertexFactoryInterpolantsVSToPS Interpolants);
void SetPrimitiveId(inout FVertexFactoryInterpolantsVSToPS Interpolants, uint PrimitiveId);
```

5.2.3.14 LocalVertexFactory.ush

The local vertex factory module defines data types and interfaces related to bone skinning, vertex shaders, etc.:

```
#ifGPUSKIN_PASS_THROUGH
#include"GpuSkinCommon.ush"
#endif

#include    "/Engine/Generated/UniformBuffers/PrecomputedLightingBuffer.ush"

(.....)

#i    USE_INSTANCING
f     USE_DITHERED_LOD_TRANSITION float4
#i    InstancingViewZCompareZero; float4              // w contains random lod scale
f                InstancingViewZCompareOne;
      float4     InstancingViewZConstant;
      float4     InstancingWorldViewOriginZero;
      float4     InstancingWorldViewOriginOne;
#endif

float4     InstancingOffset;
float4     InstancingFadeOutParams;
uint InstanceOffset;
#endif        // USE_INSTANCING

(.....)

#if   MANUAL_VERTEX_FETCH
      #define    VF_ColorIndexMask_Index        0
      #define VF_NumTexcoords_Index      1
      #define    FV_LightMapIndex_Index        2
      #define    VF_VertexOffset 3

      Buffer<float4>       VertexFetch_InstanceOriginBuffer;
      Buffer<float4>       VertexFetch_InstanceTransformBuffer;
      Buffer<float4>       VertexFetch_InstanceLightmapBuffer;

      #ifUSE_INSTANCING && USE_INSTANCING_BONEMAP
        Buffer<float4>             VertexFetch_InstancePrevTransformBuffer;
            Buffer<uint>           VertexFetch_InstanceBoneMapBuffer;
      #endif
#endif//!       MANUAL_VERTEX_FETCH

//From bound verticesBufferVertex input data obtained
from . structFVertexFactoryInput {
```

```
	//Location.
	float4		Position		: ATTRIBUTE0;

	//Tangents and Colors
#if	!MANUAL_VERTEX_FETCH
	#if METAL_PROFILE
		float3 TangentX			:	ATTRIBUTE1;
		// TangentZ.w contains sign of tangent basis determinant float4 TangentZ :
		ATTRIBUTE2;

		float4		Color			: ATTRIBUTE3;
	#else
		half3		TangentX		: ATTRIBUTE1;
		// TangentZ.w contains sign of tangent basis determinant half4 TangentZ :
		ATTRIBUTE2;

		half4		Color			: ATTRIBUTE3;
	#endif
#endif

	//Texture coordinates.
#if	NUM_MATERIAL_TEXCOORDS_VERTEX
	#if !MANUAL_VERTEX_FETCH
		#if	GPUSKIN_PASS_THROUGH
			// These must match GPUSkinVertexFactory.usf float2
			TexCoords[NUM_MATERIAL_TEXCOORDS_VERTEX]			: ATTRIBUTE4;
			#ifNUM_MATERIAL_TEXCOORDS_VERTEX > 4
				#errorToo many texture coordinate sets defined on GPUSkin vertex input.
Max:	4.
			#endif
		#else
			#ifNUM_MATERIAL_TEXCOORDS_VERTEX > 1
				float4 PackedTexCoords4[NUM_MATERIAL_TEXCOORDS_VERTEX/2] : ATTRIBUTE4;
			#endif
			#ifNUM_MATERIAL_TEXCOORDS_VERTEX == 1 float2
				PackedTexCoords2 : ATTRIBUTE4;
			#elifNUM_MATERIAL_TEXCOORDS_VERTEX == 3 float2
				PackedTexCoords2 : ATTRIBUTE5;
			#elifNUM_MATERIAL_TEXCOORDS_VERTEX == 5 float2
				PackedTexCoords2 : ATTRIBUTE6;
			#elifNUM_MATERIAL_TEXCOORDS_VERTEX == 7 float2
				PackedTexCoords2 : ATTRIBUTE7;
			#endif
		#endif
	#endif
#elifUSE_PARTICLE_SUBUVS && !MANUAL_VERTEX_FETCH float2
	TexCoords[1] : ATTRIBUTE4;
#endif

	//Instantiate data.
#ifUSE_INSTANCING && !MANUAL_VERTEX_FETCH
	float4 InstanceOrigin : ATTRIBUTE8;// per-instance random in w
	half4 InstanceTransform1 : ATTRIBUTE9;// hitproxy.r + 256 * selected in .w half4 InstanceTransform2 :
	ATTRIBUTE10;// hitproxy.g in .w half4 InstanceTransform3 : ATTRIBUTE11;// hitproxy.b in .w float4
	InstanceLightmapAndShadowMapUVBias : ATTRIBUTE12;

#endif//USE_INSTANCING
```

```
        //Graphicsid,For use fromGPU SceneAccess
#if     data. VF_USE_PRIMITIVE_SCENE_DATA uint
        PrimitiveId : ATTRIBUTE13;
#endif

        //Light map coordinates.
#ifNEEDS_LIGHTMAP_COORDINATE && !MANUAL_VERTEX_FETCH float2
        LightMapCoordinate : ATTRIBUTE15;
#endif

        //InstantiationID.
#if     USE_INSTANCING
        uint InstanceId             :   SV_InstanceID;
#endif

        //vertexID.
#ifGPUSKIN_PASS_THROUGH || MANUAL_VERTEX_FETCH uint
        VertexId : SV_VertexID;
#endif
};

#ifRAYHITGROUPSHADER || COMPUTESHADER
#ifGPUSKIN_PASS_THROUGH
Buffer<float> GPUSkinCachePositionBuffer;
#endif

#endif

//Compute shader related.
#ifCOMPUTESHADER
FVertexFactoryInputLoadVertexFactoryInputForDynamicUpdate(uint TriangleIndex,int VertexIndex, uint
PrimitiveId);
#endif

//Only position information is input for vertex data.
structFPositionOnlyVertexFactoryInput {

        float4          Position            : ATTRIBUTE0;

#ifUSE_INSTANCING && !MANUAL_VERTEX_FETCH
        float4 InstanceOrigin : ATTRIBUTE8;// per-instance random in w
        half4 InstanceTransform1 : ATTRIBUTE9;// hitproxy.r + 256 * selected in .w half4 InstanceTransform2 :
        ATTRIBUTE10;// hitproxy.g in .w half4 InstanceTransform3 : ATTRIBUTE11;// hitproxy.b in .w

#endif          // USE_INSTANCING

#if     VF_USE_PRIMITIVE_SCENE_DATA uint
        PrimitiveId : ATTRIBUTE1;
#endif

#if     USE_INSTANCING
        uint InstanceId             :   SV_InstanceID;
#endif

#if     MANUAL_VERTEX_FETCH
        uint VertexId : SV_VertexID;
#endif
};
```

```
//An input for vertex data containing only positions and normals.
structFPositionAndNormalOnlyVertexFactoryInput {

	float4		Position		: ATTRIBUTE0;
	float4		Normal			: ATTRIBUTE2;

#ifUSE_INSTANCING && !MANUAL_VERTEX_FETCH
	float4 InstanceOrigin : ATTRIBUTE8;// per-instance random in w
	half4 InstanceTransform1 : ATTRIBUTE9;// hitproxy.r + 256 * selected in .w half4 InstanceTransform2 :
	ATTRIBUTE10;// hitproxy.g in .w half4 InstanceTransform3 : ATTRIBUTE11;// hitproxy.b in .w

#endif		// USE_INSTANCING

#if	VF_USE_PRIMITIVE_SCENE_DATA uint
	PrimitiveId : ATTRIBUTE1;
#endif

# ifUSE_INSTANCING
	 uint InstanceId		:	SV_InstanceID;
#endif

#if	MANUAL_VERTEX_FETCH
	uint VertexId : SV_VertexID;
#endif
};

//Cache the vertex data of intermediate calculation results to prevent
repeated calculations. structFVertexFactoryIntermediates {

	half3x3	TangentToLocal;
	half3x3	TangentToWorld;
	half TangentToWorldSign;

	half4 Color;
#if	USE_INSTANCING
		float4	InstanceOrigin;
		float4	InstanceTransform1;
		float4	InstanceTransform2;
		float4	InstanceTransform3;

		#if	USE_INSTANCING_BONEMAP
			float4	InstancePrevOrigin;
			float4	InstancePrevTransform1;
			float4	InstancePrevTransform2;
			float4	InstancePrevTransform3;
		#endif

		float4 InstanceLightmapAndShadowMapUVBias;

	// x = per-instance random, y = per-instance fade out amount, z = hide/show flag, w dither fade
		cutoff
	float4	PerInstanceParams;
#endif		// USE_INSTANCING
	uint PrimitiveId;

	float3 PreSkinPosition;
};
```

```
//Get the instantiated data interface.
#ifUSE_INSTANCING
float4x4    GetInstanceTransform(FVertexFactoryIntermediates GetInstancePrInevteTrmanesdfoiartmes);
float4x4    (FVertexFactoryIntermediates GetInstanceTransform(FPositionOnlyVerteInxFtearcmtoerdyliantpeust) ;
float4x4    GetInstanceTransform(FPositionAndNormalOnlyVertexFactoryInput      Input);
float4x4    GetInstanceToLocal3x3(FVertexFactoryIntermediates Intermediates);                  Input);
half3x3     GetInstanceShadowMapBias(FVertexFactoryIntermediates GetInstanceLightMapBias
float2      (FVertexFactoryIntermediates GetInstanceSelected(FVertexFactoryIntermInetdeiramteesd iates);
float2      Intermediates); GetInstanceRandom(FVertexFactoryIntermediates InteIrnmterdmiaetdesia);tes);
float
float
float3      GetInstanceOrigin(FVertexFactoryIntermediates Intermediates);
#endif          // USE_INSTANCING

//Get material parameters from the interpolation structure.
FMaterialPixelParametersGetMaterialPixelParameters(FVertexFactoryInterpolantsVSToPS Interpolants, float4
SvPosition);
half3x3CalcTangentToWorldNoScale(FVertexFactoryIntermediates Intermediates, half3x3 TangentToLocal);

FMaterialVertexParametersGetMaterialVertexParameters(FVertexFactoryInput Input, ...);

(......)

//Vertex data calculation and acquisition interface.
float4CalcWorldPosition(float4 Position, uint PrimitiveId);
half3x3CalcTangentToLocal(FVertexFactoryInput Input, outfloatTangentSign); half3x3CalcTangentToWorld
(FVertexFactoryIntermediates Intermediates, ...); FVertexFactoryIntermediatesGetVertexFactoryIntermediates
(FVertexFactoryInput half3x3VertexFactoryGetTangentToLocal( FVertexFactoryInput Input, ...); float4        Input);
VertexFactoryGetWorldPosition(FVertexFactoryInput Input, ...); float4
VertexFactoryGetRasterizedWorldPosition(FVertexFactoryInput Input, ...); float3
VertexFactoryGetPositionForVertexLighting(FVertexFactoryInput Input, ...); FVertexFactoryInterpolantsVSToPS

                                        VertexFactoryGetInterpolantsVSToPS(FVertexFactoryInput
Input,      . . . ) ;
float4      VertexFactoryGetWorldPosition(FPositionOnlyVertexFactoryInput Input); VertexFactoryGetWorldPosition
float4      (FPositionAndNormalOnlyVertexFactoryInput Input); VertexFactoryGetWorldNormal
float3      (FPositionAndNormalOnlyVertexFactoryInput Input);
float3VertexFactoryGetWorldNormal(FVertexFactoryInput Input, ...); float4
VertexFactoryGetPreviousWorldPosition(FVertexFactoryInput Input, ...); float4
VertexFactoryGetInstanceHitProxyId(FVertexFactoryInput Input, ...);
float4VertexFactoryGetTranslatedPrimitiveVolumeBounds(FVertexFactoryInterpolantsVSToPS Interpolants);

uintVertexFactoryGetPrimitiveId(FVertexFactoryInterpolantsVSToPS Interpolants);

(......)
```

Since the vertex factory needs to be compatible with many types, a large number of macros are added to control data members, resulting in bloated code and poor readability. This is also a drawback of the Uber Shader design framework.

5.3 BasePass

This section mainly elaborates on the rendering process, rendering status, Shader logic, etc. of BasePass.

5.3.1 BasePass rendering process

The previous article also involved the rendering process and some core logic of BasePass. This section briefly reviews it. BasePass comes afterFDeferredShadingSceneRenderer::RenderPrePass and before LightingPass:

```
void FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList) {

    (.....)

    //Draws scene depth.
    RenderPrePass(RHICmdList, ...);

    (.....)

    //RenderingBase Pass.
    RenderBasePass(RHICmdList, ...);

    (.....)

    //Rendering light sources.
    RenderLights(RHICmdList, ...);

    (.....)
}
```

Here is the logicRenderBasePassforRenderBasePassViewParallelparallel rendering:

```
bool FDeferredShadingSceneRenderer::RenderBasePass(FRHICommandListImmediate& RHICmdList,
FExclusiveDepthStencil::Type BasePassDepthStencilAccess, IPooledRenderTarget* ForwardScreenSpaceShadowMask,
boolbParallelBasePass,boolbRenderLightmapDensity) {

    (.....)

    FExclusiveDepthStencil::Type BasePassDepthStencilAccess_NoDepthWrite                    =
FExclusiveDepthStencil::Type(BasePassDepthStencilAccess ~FExclusiveDepth&Stencil::DepthWrite);

    //  Parallel Mode
    if  (bParallelBasePass)
    {
        //Drawing task waiting.
        FScopedCommandListWaitForTasks
Flusher(CVarRHICmdFlushRenderThreadTasksBasePass.GetValueOnRenderThread() >0||
CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() >0, RHICmdList);

        //Traverse allview,EachviewRender onceBase Pass.
        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

            FViewInfo& View = Views[ViewIndex];

            // Uniform Buffer
```

```
                TUniformBufferRef<FOpaqueBasePassUniformParameters>             BasePassUniformBuffer;
                CreateOpaqueBasePassUniformBuffer(RHICmdList, View,
ForwardScreenSpaceShadowMask, nullptr, nullptr, nullptr, BasePassUniformBuffer);

                // Render State
                FMeshPassProcessorRenderStateDrawRenderState(View,              BasePassUniformBuffer);
                SetupBasePassState(BasePassDepthStencilAccess,
ViewFamily.EngineShowFlags.ShaderComplexity, DrawRenderState);

                const boolbShouldRenderView = View.ShouldRenderView(); if
                (bShouldRenderView)
                {
                    Scene->UniformBuffers.UpdateViewUniformBuffer(View);

                    //Perform parallel rendering.
                    RenderBasePassViewParallel(View, RHICmdList, BasePassDepthStencilAccess,
DrawRenderState);
                }

                FSceneRenderTargets::Get(RHICmdList).BeginRenderingGBuffer(RHICmdList,
ERenderTargetLoadAction::ELoad, ERenderTargetLoadAction::ELoad, BasePassDepthStencilAccess, this-
>ViewFamily.EngineShowFlags.ShaderComplexity);
                RHICmdList.EndRenderPass();

                (......)
            }
        }

        (......)
}

void FDeferredShadingSceneRenderer::RenderBasePassViewParallel(FViewInfo& View,
FRHICommandListImmediate& ParentCmdList, FExclusiveDepthStencil::Type BasePassDepthStencilAccess,const
FMeshPassProcessorRenderState& InDrawRenderState) {

        //Data for parallel drawing: command queue, context, rendering state, etc.
        FBasePassParallelCommandListSetParallelSet(View, ParentCmdList,
            CVarRHICmdBasePassDeferredContexts.GetValueOnRenderThread() >0,
            CVarRHICmdFlushRenderThreadTasksBasePass.GetValueOnRenderThread() ==0&&
CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() ==0,
            this,
            BasePassDepthStencilAccess,
            InDrawRenderState);

        //Triggers parallel drawing instructions.
        View.ParallelMeshDrawCommandPasses[EMeshPass::BasePass].DispatchDraw(&ParallelSet, ParentCmdList);

}
```

RenderBasePassWhen it relies onFScopedCommandListWaitForTaskswaiting for the drawing command to complete, the following is the implementation code of the latter:

```
// Engine\Source\Runtime\RHI\Public\RHICommandList.h

//Refresh type immediately.
namespace EImmediateFlushType
```

```
{
	enum Type
	{
		WaitForOutstandingTasksOnly =0,//Wait only for currently unfinished tasks. DispatchToRHIThread,//Send toRHIThreads.
		WaitForDispatchToRHIThread,//Waiting to send toRHIThreads. FlushRHIThread,//refreshRHIThread toGPU.
		FlushRHIThreadFlushResources,//refreshRHIThread and refresh resources.
		FlushRHIThreadFlushResourcesFlushDeferredDeletes//refreshRHIThread and refresh resources and refresh delayed resource
		deletion instructions.


	};
};

structFScopedCommandListWaitForTasks {

	FRHICommandListImmediate& RHICmdList;//Need to waitRHICmdList. bool
	bWaitForTasks;//Whether to wait for the task.

	FScopedCommandListWaitForTasks(boolInbWaitForTasks, FRHICommandListImmediate& InRHICmdList)

		: RHICmdList(InRHICmdList) ,
		bWaitForTasks(InbWaitForTasks)
	{
	}

	//Waiting to be executed in the destructor.
	~FScopedCommandListWaitForTasks() {

		if(bWaitForTasks)
		{
			//	If it is independentRHIThreads only wait for currently
			if	unfinished tasks. (IsRunningRHIInSeparateThread())
			{

RHICmdList.ImmediateFlush(EImmediateFlushType::WaitForOutstandingTasksOnly);
			}
			//DependentRHIThread, direct refreshRHIThreads.
			else

			{
				RHICmdList.ImmediateFlush(EImmediateFlushType::FlushRHIThread);
			}
		}
	}
};
```

As for whether you need to wait for BasePass rendering to complete, it can be enabled by one of the two console commands CVarRHICmdFlushRenderThreadTasksBasePass (r.RHICmdFlushRenderThreadTasksBasePass) or CVarRHICmdFlushRenderThreadTasks (r.RHICmdFlushRenderThreadTasks).

5.3.2 BasePass rendering status

This section elaborates on the various rendering states, shader bindings, and drawing parameters used by BasePass when rendering. We know that BasePass is used toFBasePassMeshProcessorcollect

many shader bindings and drawing parameters. From the Processor process, we can easily know that the VS and PS used by BasePass when drawing areTBasePassVSandTBasePassPS:

```
// Engine\Source\Runtime\Renderer\Private\BasePassRendering.cpp

void FBasePassMeshProcessor::Process(constFMeshBatch& RESTRICT MeshBatch, ...) {

	(......)

	TMeshProcessorShaders<
		TBasePassVertexShaderPolicyParamType<LightMapPolicyType>, FBaseHS,

		FBaseDS,
		TBasePassPixelShaderPolicyParamType<LightMapPolicyType>>		BasePassShaders;

	GetBasePassShaders<LightMapPolicyType>(MaterialResource,VertexFactory->GetType(),
. . . ) ;

	(......)
}

// Engine\Source\Runtime\Renderer\Private\BasePassRendering.h

template <typename LightMapPolicyType>
void GetBasePassShaders(constFMaterial& Material, ...) {

	(......)

	VertexShader = Material.GetShader<TBasePassVS<LightMapPolicyType,false> > (VertexFactoryType);

	PixelShader = Material.GetShader<TBasePassPS<LightMapPolicyType,false> > (VertexFactoryType);


	(......)
}
```

First analyzeTBasePassVS,It has two parent classes:

```
// Engine\Source\Runtime\Renderer\Private\BasePassRendering.h

template<typename LightMapPolicyType>
classTBasePassVertexShaderPolicyParamType: public FMeshMaterialShader, public
LightMapPolicyType::VertexParametersType
{
protected:
	(......)

	TBasePassVertexShaderPolicyParamType(const
FMeshMaterialShaderType::CompiledShaderInitializerType&		Initializer):
FMeshMaterialShader(Initializer)
	{
		LightMapPolicyType::VertexParametersType::Bind(Initializer.ParameterMap);
		ReflectionCaptureBuffer.Bind(Initializer.ParameterMap, TEXT("ReflectionCapture"));
	}

public:
```

```
	//GetShaderBinding.
	void GetShaderBindings(constFScene* Scene, ERHIFeatureLevel::Type FeatureLevel, ...) const;

	void GetElementShaderBindings(constFShaderMapPointerTable& PointerTable,const FScene* Scene, ...)
const;

	//Reflection ballUniform Buffer. LAYOUT_FIELD(FShaderUniformBufferParameter,
	ReflectionCaptureBuffer);
};


template<typename LightMapPolicyType> class
TBasePassVertexShaderBaseType: public
TBasePassVertexShaderPolicyParamType<LightMapPolicyType> {

	(.....)

public:
	static bool ShouldCompilePermutation(constFMeshMaterialShaderPermutationParameters& Parameters)

	{
		returnLightMapPolicyType::ShouldCompilePermutation(Parameters);
	}

	static void ModifyCompilationEnvironment(constFMaterialShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
	{
		LightMapPolicyType::ModifyCompilationEnvironment(Parameters, OutEnvironment);
		Super::ModifyCompilationEnvironment(Parameters, OutEnvironment);
	}
};

//BasePassVertex shader.
template<typename LightMapPolicyType,boolbEnableAtmosphericFog> classTBasePassVS: public
TBasePassVertexShaderBaseType<LightMapPolicyType> {

	(.....)

public:
	//Disable some of the arrangementsshaderCompile.
	static bool ShouldCompilePermutation(constFMeshMaterialShaderPermutationParameters& Parameters)

	{
		(.....)

		return	bShouldCache
			&& (IsFeatureLevelSupported(Parameters.Platform, ERHIFeatureLevel::SM5));
	}

	//Modify the compilation environment: macro definition.
	static void ModifyCompilationEnvironment(constFMaterialShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
	{
		Super::ModifyCompilationEnvironment(Parameters, OutEnvironment);
		OutEnvironment.SetDefine(TEXT("BASEPASS_ATMOSPHERIC_FOG"), !
IsMetalMRTPlatform(Parameters.Platform) ? bEnableAtmosphericFog :0);
```

```
    }
};
```

According to the above code, itTBasePassVSprovides interfaces such as obtaining shader binding, changing the compilation environment, and only compiling the specified combination of shaders. In addition, it also has properties such as reflection ball and light map type. The following is a code analysis of obtaining shader binding:

```
// Engine\Source\Runtime\Renderer\Private\BasePassRendering.inl

//Get the specified elementVSofshaderBinding.
template<typename LightMapPolicyType>
voidTBasePassVertexShaderPolicyParamType<LightMapPolicyType>::GetShaderBindings(
    constFScene* Scene,
    ERHIFeatureLevel::Type FeatureLevel,
    const FPrimitiveSceneProxy* PrimitiveSceneProxy,
    const FMaterialRenderProxy& MaterialRenderProxy,
    const FMaterial& Material, const const
    TBasePassVSFShaderElementData::LightMapPolicyType& DrawRenderState,
                                                        ShaderElementData,
    FMeshDrawSingleShaderBindings& ShaderBindings const
{
    //Get it firstFMeshMaterialShaderofshaderBinding.
    FMeshMaterialShader::GetShaderBindings(Scene, FeatureLevel, PrimitiveSceneProxy,
MaterialRenderProxy, Material, DrawRenderState, ShaderElementData, ShaderBindings);

    // If there is a scene instance, get the reflection ball of the scene
    if Buffer. (Scene)
    {
        FRHIUniformBuffer* ReflectionCaptureUniformBuffer = Scene-
> UniformBuffers.ReflectionCaptureUniformBuffer.GetReference(); ShaderBindings.Add(ReflectionCaptureBuffer,
ReflectionCaptureUniformBuffer);
    }
    //If there is no scene, a default reflection sphere is created.Buffer.
    else
    {
        ShaderBindings.Add(ReflectionCaptureBuffer,
        DrawRenderState.GetReflectionCaptureUniformBuffer());
    }

    //Finally, get it from the light map typeshaderBinding.
    LightMapPolicyType::GetVertexShaderBindings(
        PrimitiveSceneProxy,
        ShaderElementData.LightMapPolicyElementData, this,

        ShaderBindings);
}

//Get the specifiedFMeshBatchElementofVSofshader
Binding. template<typename LightMapPolicyType>
voidTBasePassVertexShaderPolicyParamType<LightMapPolicyType>::GetElementShaderBindings(
    const FShaderMapPointerTable& PointerTable, FScene*
    const Scene,
    const FSceneView* ViewIfDynamicMeshCommand,
    const FVertexFactory* VertexFactory,
    const EVertexInputStreamType InputStreamType,
```

```
        ERHIFeatureLevel::Type FeatureLevel,
        const FPrimitiveSceneProxy* PrimitiveSceneProxy, const
        TBasePassVertexShaderBatchedMeshBatch<LightMapPolicyType>&
                FMeshBatchElement& BatchElement,
                                                                ShaderElementData,
        FMeshDrawSingleShaderBindings& ShaderBindings,
        FVertexInputStreamArray& VertexStreams) const
{
        //Direct callFMeshMaterialShaderThe corresponding interface.
        FMeshMaterialShader::GetElementShaderBindings(PointerTable, ViewIfDynamicMSecsehnCeo,mmand,
VertexFactory, InputStreamType, FeatureLevel, PrimitiveSceneProxy, MeshBatch, BatchElement,
ShaderElementData, ShaderBindings, VertexStreams);

}
```

The code to be analyzed belowTBasePassPSis similar to VS, and also has two parent classes:

```
// Engine\Source\Runtime\Renderer\Private\BasePassRendering.h

template<typename LightMapPolicyType>
classTBasePassPixelShaderPolicyParamType: public FMeshMaterialShader, public
LightMapPolicyType::PixelParametersType
{
public:
        static void ModifyCompilationEnvironment(constFMaterialShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment);
        static bool ValidateCompiledResult(EShaderPlatform Platform,const
FShaderParameterMap& ParameterMap, TArray<FString>& OutError);

        TBasePassPixelShaderPolicyParamType(const
FMeshMaterialShaderType::CompiledShaderInitializerType&                Initializer):
FMeshMaterialShader(Initializer)
        {
                LightMapPolicyType::PixelParametersType::Bind(Initializer.ParameterMap);
                ReflectionCaptureBuffer.Bind(Initializer.ParameterMap, TEXT("ReflectionCapture"));

                (......)
        }

        //GetshaderBinding. void
        GetShaderBindings(
                constFScene* Scene,
                ERHIFeatureLevel::Type FeatureLevel,
                const FPrimitiveSceneProxy*  PrimitiveSceneProxy,
                const FMaterialRenderProxy&  MaterialRenderProxy,
                const    FMaterial& Material, const                const
                TBasePassVertexShaderBatchedMeshBatch<LightMapPolicyType>&  DrawRenderState,
                                                                ShaderElementData,
                FMeshDrawSingleShaderBindings&  ShaderBindingsconst;

private:
        // ReflectionCaptureBuffer.
        LAYOUT_FIELD(FShaderUniformBufferParameter,  ReflectionCaptureBuffer);
};

template<typename LightMapPolicyType> class
TBasePassPixelShaderBaseType: public
```

```
TBasePassPixelShaderPolicyParamType<LightMapPolicyType> {

public:
	static bool ShouldCompilePermutation(constFMeshMaterialShaderPermutationParameters& Parameters)

	{
		returnLightMapPolicyType::ShouldCompilePermutation(Parameters);
	}

	static void ModifyCompilationEnvironment(constFMaterialShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
	{
		LightMapPolicyType::ModifyCompilationEnvironment(Parameters, OutEnvironment);
		Super::ModifyCompilationEnvironment(Parameters, OutEnvironment);
	}
};

//BasePassThe pixel shader.
template<typename LightMapPolicyType,boolbEnableSkyLight>
classTBasePassPS: public TBasePassPixelShaderBaseType<LightMapPolicyType> {

public:
	//Enable the specified arrangementshader.
	static bool ShouldCompilePermutation(constFMeshMaterialShaderPermutationParameters& Parameters)

	{
		(.....)
		return	bCacheShaders
			&& (IsFeatureLevelSupported(Parameters.Platform, ERHIFeatureLevel::SM5))
			&&
TBasePassPixelShaderBaseType<LightMapPolicyType>::ShouldCompilePermutation(Parameters);
	}

	//Modify the compilation environment.
	static void ModifyCompilationEnvironment(constFMaterialShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
	{
		OutEnvironment.SetDefine(TEXT("SCENE_TEXTURES_DISABLED"),
Parameters.MaterialParameters.MaterialDomain != MD_Surface);
		OutEnvironment.SetDefine(TEXT("COMPILE_BASEPASS_PIXEL_VOLUMETRIC_FOGGING"),
		DoesPlatformSupportVolumetricFog(Parameters.Platform));
		OutEnvironment.SetDefine(TEXT("ENABLE_SKY_LIGHT"), bEnableSkyLight);
		OutEnvironment.SetDefine(TEXT("PLATFORM_FORCE_SIMPLE_SKY_DIFFUSE"),
		ForceSimpleSkyDiffuse(Parameters.Platform));


TBasePassPixelShaderBaseType<LightMapPolicyType>::ModifyCompilationEnvironment(Parameters, OutEnvironment);

	}
};
```

SimilarTBasePassVSto ,TBasePassPSit provides interfaces such as obtaining shader binding, changing the compilation environment, and only compiling the specified permutation and combination shader. In addition, it also has properties such as reflection ball and light map type.

The following is its code for obtaining shader binding (because it is too similar to VS, no comments are given):

```
// Engine\Source\Runtime\Renderer\Private\BasePassRendering.inl

template<typename LightMapPolicyType>
void TBasePassPixelShaderPolicyParamType<LightMapPolicyType>::GetShaderBindings(
	const FScene* Scene,
	ERHIFeatureLevel::Type FeatureLevel,
	const FPrimitiveSceneProxy* PrimitiveSceneProxy,
	const FMaterialRenderProxy& MaterialRenderProxy,
	const FMaterial& Material,
	const TBasePassShaderElementData<LightMapPolicyType>& DrawRenderState,
	const ShaderElementData,
	FMeshDrawSingleShaderBindings& ShaderBindings
	const
{
	FMeshMaterialShader::GetShaderBindings(Scene, FeatureLevel, PrimitiveSceneProxy, MaterialRenderProxy, Material, DrawRenderState, ShaderElementData, ShaderBindings);

	if(Scene)
	{
		FRHIUniformBuffer* ReflectionCaptureUniformBuffer = Scene->UniformBuffers.ReflectionCaptureUniformBuffer.GetReference(); ShaderBindings.Add(ReflectionCaptureBuffer, ReflectionCaptureUniformBuffer);
	}
	else
	{
		ShaderBindings.Add(ReflectionCaptureBuffer,
		DrawRenderState.GetReflectionCaptureUniformBuffer());
	}

	LightMapPolicyType::GetPixelShaderBindings(
		PrimitiveSceneProxy,
		ShaderElementData.LightMapPolicyElementData, this,

		ShaderBindings);
}
```

After analyzing the VS and PS of BasePass, we will turn our attention FDeferredShadingSceneRenderer::RenderBasePassto analyze other rendering states of BasePass. Since the previous chapter4.3.6 BasePass has analyzed RenderState and materials, here we will directly give the results of the default settings:

- **BlendState**:TStaticBlendStateWriteMask<CW_RGBA, CW_RGBA, CW_RGBA, CW_RGBA, CW_NONE>, and the RGBA is now available.
- **DepthStencilState**:TStaticDepthStencilState<true, CF_DepthNearOrEqual>, depth writing and testing are enabled, and the comparison function is NearOrEqual.
- **Material**:MeshBatch.MaterialRenderProxy->GetMaterialWithFallback(...), uses the material collected by FMeshBatch, which is the material of the mesh itself.
- **UniformBuffer**:
  - FOpaqueBasePassUniformParameters, a dedicated uniform buffer for BasePass.
  - Scene->UniformBuffers, scene-related uniform buffers.

Other unspecified rendering states are the same as the default states.

5.3.3 BasePass Shader

The previous article has analyzed the nested logic of BasePass drawing. The outermost layer is scene, view, and grid in order:

```
foreach(scene in scenes)
{
    foreach(view in views)
    {
        foreach(mesh in meshes)
        {
            DrawMesh(...);  //Executed once per render callBasePassVertexShaderandBasePassPixelShader
The code.
        }
    }
}
```

This means that the number of times BasePassVertexShader and BasePassPixelShader are executed is:

$$Nscandnand \cdot NviandIn \cdot Nmandsh$$

This also indirectly illustratesFMeshDrawCommandthe necessity of sorting, which can reduce the exchange of data between the CPU and GPU, reduce rendering state switching, improve cache hit rate, increase instantiation probability, and reduce Draw Call. However, for real-time games, in most cases, the number of scenes and views is 1, that is, the number of VS and PS executions is only related to the number of grids.

The next two sections will analyze the VS and PS Shader logic of BasePass.

5.3.3.1 BasePassVertexShader

The entry of BasePassVertexShader is in BasePassVertexShader.usf:

```
#include "BasePassVertexCommon.ush"
#include "SHCommon.ush"

(.....)

// Base PassThe main
entrance. void Main(
    FVertexFactoryInput Input,
    OPTIONAL_VertexID
    out FBasePassVSOutput Output
#ifUSE_GLOBAL_CLIP_PLANE && !USING_TESSELLATION
    , outfloatOutGlobalClipPlaneDistance : SV_ClipDistance
# endif
#if INSTANCED_STEREO
    , uint InstanceId : SV_InstanceID
    #if !MULTI_VIEW
        , outfloatOutClipDistance: SV_ClipDistance1
```

```
	#else
		, out uint ViewportIndex : SV_ViewPortArrayIndex
	# endif
# endif
	)
{
	(.....)

	uint EyeIndex =0; ResolvedView =
	ResolveView();

	//Get intermediate derived data for a vertex.
	FVertexFactoryIntermediates VFIntermediates = GetVertexFactoryIntermediates(Input); //Get the world space
	position (without offset).
	float4 WorldPositionExcludingWPO = VertexFactoryGetWorldPosition(Input, VFIntermediates);

	float4 WorldPosition = WorldPositionExcludingWPO; float4
	ClipSpacePosition;

	//Tangent lines in local space.
	float3x3 TangentToLocal = VertexFactoryGetTangentToLocal(Input, VFIntermediates); //Get the vertex related
	parameters of the material.
	FMaterialVertexParameters VertexParameters = GetMaterialVertexParameters(Input, VFIntermediates,
WorldPosition.xyz, TangentToLocal);

	// WorldPositionAdded material offset. {

		WorldPosition.xyz += GetMaterialWorldPositionOffset(VertexParameters);
	}

	(.....)

	//Compute clip space position.
	{
		float4 RasterizedWorldPosition = VertexFactoryGetRasterizedWorldPosition(Input, VFIntermediates,
		WorldPosition);
		ClipSpacePosition = INVARIANT(mul(RasterizedWorldPosition,
		ResolvedView.TranslatedWorldToClip));
		Output.Position = INVARIANT(ClipSpacePosition);
	}

	(.....)

	//Global clipping plane distance.
#if USE_GLOBAL_CLIP_PLANE
	OutGlobalClipPlaneDistance		= dot(ResolvedView.GlobalClippingPlane,
float4(WorldPosition.xyz - ResolvedView.PreViewTranslation.xyz,1));
#endif
	#ifUSE_WORLD_POSITION_EXCLUDING_SHADER_OFFSETS
		Output.BasePassInterpolants.PixelPositionExcludingWPO =
		WorldPositionExcludingWPO.xyz;
	#endif

	//Output the data that needs to be interpolated.
	Output.FactoryInterpolants = VertexFactoryGetInterpolants(Input, VFIntermediates, VertexParameters);


	(.....)
```

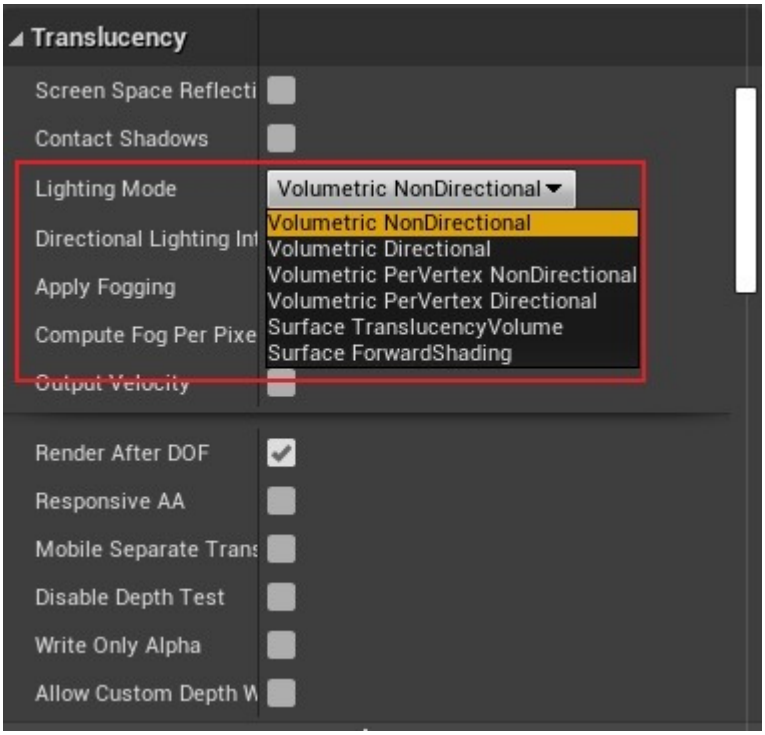
```
        //The fog color needed to calculate transparency.
#if NEEDS_BASEPASS_VERTEX_FOGGING
#ifBASEPASS_ATMOSPHERIC_FOG
Output.BasePassInterpolants.VertexFog                    =
CalculateVertexAtmosphericFog(WorldPosition.xyz,
ResolvedView.TranslatedWorldCameraOrigin);
        #else
        Output.BasePassInterpolants.VertexFog = CalculateHeightFog(WorldPosition.xyz -
        ResolvedView.TranslatedWorldCameraOrigin);
        #endif

        (.....)
#endif

        //Per-vertex lighting for transparent objects.
#if TRANSLUCENCY_ANY_PERVERTEX_LIGHTING
        float3 WorldPositionForVertexLightingTranslated                =
VertexFactoryGetPositionForVertexLighting(Input, VFIntermediates,                WorldPosition.xyz);
        float3 WorldPositionForVertexLighting = WorldPositionForVertexLightingTranslated -
        ResolvedView.PreViewTranslation.xyz;
#endif

        //There are two types of per-vertex lighting for transparent objects: per-vertex lighting volumetric (TRANSLUCENCY_PERVERTEX_LIGHTING_VOLUME)and per-vertex
forward shading (TRANSLUCENCY_PERVERTEX_FORWARD_SHADING).
#if    TRANSLUCENCY_PERVERTEX_LIGHTING_VOLUME
        float4 VolumeLighting;
        float3 InterpolatedLighting =0;

        float3     InnerVolumeUVs;
        float3     OuterVolumeUVs;
        float     FinalLerpFactor;

        float3 LightingPositionOffset =0;
        ComputeVolumeUVs(WorldPositionForVertexLighting,          LightingPositionOffset,
InnerVolumeUVs, OuterVolumeUVs, FinalLerpFactor);

        #ifTRANSLUCENCY_LIGHTING_VOLUMETRIC_PERVERTEX_DIRECTIONAL

                Output.BasePassInterpolants.AmbientLightingVector =
GetAmbientLightingVectorFromTranslucentLightingVolume(InnerVolumeUVs,          OuterVolumeUVs,
FinalLerpFactor).xyz;
                Output.BasePassInterpolants.DirectionalLightingVector =
GetDirectionalLightingVectorFromTranslucentLightingVolume(InnerVolumeUVs, FinalLerpFactor);     OuterVolumeUVs,


        #elifTRANSLUCENCY_LIGHTING_VOLUMETRIC_PERVERTEX_NONDIRECTIONAL

                Output.BasePassInterpolants.AmbientLightingVector =
GetAmbientLightingVectorFromTranslucentLightingVolume(InnerVolumeUVs, OuterVolumeUVs,
FinalLerpFactor).xyz;

        #endif
#elifTRANSLUCENCY_PERVERTEX_FORWARD_SHADING float4
        VertexLightingClipSpacePosition =
mul(float4(WorldPositionForVertexLightingTranslated,          1),
ResolvedView.TranslatedWorldToClip);
        float2 SvPosition = (VertexLightingClipSpacePosition.xy /
```

```
VertexLightingClipSpacePosition.w * float2(.5f,-.5f) +.5f) *
ResolvedView.ViewSizeAndInvSize.xy;
	uint GridIndex = ComputeLightGridCellIndex((uint2)SvPosition,
VertexLightingClipSpacePosition.w, EyeIndex);
	Output.BasePassInterpolants.VertexDiffuseLighting =
GetForwardDirectLightingForVertexLighting(GridIndex, WorldPositionForVertexLighting, Output.Position.w,
VertexParameters.TangentToWorld[2], EyeIndex);

#endif


	//Precomputed Illuminance Volumetric Lighting
	#ifPRECOMPUTED_IRRADIANCE_VOLUME_LIGHTING && TRANSLUCENCY_ANY_PERVERTEX_LIGHTING float3
		BrickTextureUVs = ComputeVolumetricLightmapBrickTextureUVs(WorldPositionForVertexLighting);


		#if	TRANSLUCENCY_LIGHTING_VOLUMETRIC_PERVERTEX_NONDIRECTIONAL FOneBandSHVectorRGB
			IrradianceSH = GetVolumetricLightmapSH1(BrickTextureUVs);
			Output.BasePassInterpolants.VertexIndirectAmbient = float3(IrradianceSH.RV,
IrradianceSH.GV, IrradianceSH.BV);
		#elifTRANSLUCENCY_LIGHTING_VOLUMETRIC_PERVERTEX_DIRECTIONAL
			// Need to interpolate directional lighting so we can incorporate a normal in
the pixel shader
			FTwoBandSHVectorRGB IrradianceSH = GetVolumetricLightmapSH2(BrickTextureUVs);
			Output.BasePassInterpolants.VertexIndirectSH[0] Output.BasePa=ssIInrtreardpiaonlacnetSsH.V.RerVt;exIndirectS
			Output.BasePassInterpolants.VertexIndirectSH[2]                    = IrradianceSH.GV;
                                                                               = IrradianceSH.BV;
		#endif
	#endif

	//Processing speed buffer.
#if	WRITES_VELOCITY_TO_GBUFFER {

		float4 PrevTranslatedWorldPosition = float4(0,0,0,1); BRANCH

		if(GetPrimitiveData(VFIntermediates.PrimitiveId).OutputVelocity>0) {

			PrevTranslatedWorldPosition = VertexFactoryGetPreviousWorldPosition( Input,
VFIntermediates );
			VertexParameters = GetMaterialVertexParameters(Input, VFIntermediates,
PrevTranslatedWorldPosition.xyz, TangentToLocal);
			PrevTranslatedWorldPosition.xyz +=
GetMaterialPreviousWorldPositionOffset(VertexParameters);

			#if !USING_TESSELLATION
				PrevTranslatedWorldPosition =		mul(float4(PrevTranslatedWorldPosition.xyz,
1),	ResolvedView.PrevTranslatedWorldToClip);
			#endif
		}

		(.....)

		//compute the old screen pos with the old world position and the old camera
matrix
		Output.BasePassInterpolants.VelocityPrevScreenPosition                    =
		PrevTranslatedWorldPosition;
		//The screen space position where the velocity is stored.
		#ifWRITES_VELOCITY_TO_GBUFFER_USE_POS_INTERPOLATOR
		Output.BasePassInterpolants.VelocityScreenPosition = ClipSpacePosition;
```

```
        #endif
    }
#endif        // WRITES_VELOCITY_TO_GBUFFER

    OutputVertexID( Output );
}
```

Although the vertex shader of the Base Pass does not calculate dynamic lighting, it is not as simple as imagined: it includes functions such as processing intermediate derived data, coordinate transformation, clipping planes, fog effects for transparent objects, per-vertex lighting for transparent objects, and processing speed buffers.

Per-vertex lighting for transparent objects only works on semi-transparent objects and can be specified in the material properties panel used by the object:



The following will analyze and process the interfaces of vertex intermediate derived data GetVertexFactoryIntermediates,material vertex parametersGetMaterialVertexParameters , vertex interpolation dataVertexFactoryGetInterpolantsVSToPS,etc.:

```
// Engine\Shaders\Private\LocalVertexFactory.ush

//Processing of intermediate derived data.
FVertexFactoryIntermediatesGetVertexFactoryIntermediates(FVertexFactoryInput {                    Input)

    FVertexFactoryIntermediates Intermediates;

    //GraphicsID(EnableGPU Sceneis effective).
#if  VF_USE_PRIMITIVE_SCENE_DATA
    Intermediates.PrimitiveId          =  Input.PrimitiveId;
#else
    Intermediates.PrimitiveId          =  0;
#endif
```

```
	//Vertex colors.
#if MANUAL_VERTEX_FETCH
	Intermediates.Color =
LocalVF.VertexFetch_ColorComponentsBuffer[(LocalVF.VertexFetch_Parameters[VF_VertexOffset]
+ Input.VertexId) & LocalVF.VertexFetch_Parameters[VF_ColorIndexMask_Index]]
FMANUALFETCH_COLOR_COMPONENT_SWIZZLE;// Swizzle vertex color.
#else
	Intermediates.Color = Input.Color FCOLOR_COMPONENT_SWIZZLE;// Swizzle vertex color.
#endif

	//Instantiate data.
#ifUSE_INSTANCING && MANUAL_VERTEX_FETCH && !USE_INSTANCING_BONEMAP uint InstanceId =
	GetInstanceId(Input.InstanceId); Intermediates.InstanceTransform1 =
	InstanceVF.VertexFetch_InstanceTransformBuffer[3* (InstanceId + InstanceOffset) +0];

	Intermediates.InstanceTransform2 = InstanceVF.VertexFetch_InstanceTransformBuffer[3* (InstanceId +
InstanceOffset) +1];
	Intermediates.InstanceTransform3 = InstanceVF.VertexFetch_InstanceTransformBuffer[3* (InstanceId +
InstanceOffset) +2];
	Intermediates.InstanceOrigin = InstanceVF.VertexFetch_InstanceOriginBuffer[(InstanceId
+ InstanceOffset)];
	Intermediates.InstanceLightmapAndShadowMapUVBias =
InstanceVF.VertexFetch_InstanceLightmapBuffer[(InstanceId + InstanceOffset)];
#elifMANUAL_VERTEX_FETCH && USE_INSTANCING_BONEMAP uint
	InstanceIndex =
VertexFetch_InstanceBoneMapBuffer[LocalVF.VertexFetch_Parameters[VF_VertexOffset] + Input.VertexId];

	Intermediates.InstanceTransform1 = VertexFetch_InstanceTransformBuffer[4* InstanceIndex +0];

	Intermediates.InstanceTransform2 = VertexFetch_InstanceTransformBuffer[4* InstanceIndex +1];

	Intermediates.InstanceTransform3 = VertexFetch_InstanceTransformBuffer[4* InstanceIndex +2];

	Intermediates.InstanceOrigin = VertexFetch_InstanceTransformBuffer[4* InstanceIndex +
3];

	Intermediates.InstancePrevTransform1 = VertexFetch_InstancePrevTransformBuffer[4* InstanceIndex +0];

	Intermediates.InstancePrevTransform2 = VertexFetch_InstancePrevTransformBuffer[4* InstanceIndex +1];

	Intermediates.InstancePrevTransform3 = VertexFetch_InstancePrevTransformBuffer[4* InstanceIndex +2];

	Intermediates.InstancePrevOrigin = VertexFetch_InstancePrevTransformBuffer[4* InstanceIndex +3];


	Intermediates.InstanceLightmapAndShadowMapUVBias = float4(0,0,0,0);
#elifUSE_INSTANCING
	Intermediates.InstanceTransform1
	Intermed=iatIensp.Iunts.tIannstcaenTcreaTnrsafonrsmfo2rm 1;
	Intermediates.InstanceTransform3 =Input.InstanceTransform3;
	Intermediates.InstanceLightmapAndShadowMapUVBias =
	Input.InstanceLightmapAndShadowMapUVBias;

#endif

	//Tangents and tangent transformation matrices.
	floatTangentSign;
	Intermediates.TangentToLocal = CalcTangentToLocal(Input, TangentSign);
```

```
        Intermediates.TangentToWorld =
        CalcTangentToWorld(Intermediates,Intermediates.TangentToLocal);
        Intermediates.TangentToWorldSign = TangentSign *
        GetPrimitiveData(Intermediates.PrimitiveId).InvNonUniformScaleAndDeterminantSign.w;


        //Instantiate data.
#ifUSE_INSTANCING && !USE_INSTANCING_BONEMAP // x =
        per-instance random
        // y = per-instance fade out factor
        // z = zero or one depending of if it is shown at all // w is dither cutoff

        // PerInstanceParams.z stores a hide/show flag for this instance floatSelectedValue =
        GetInstanceSelected(Intermediates); Intermediates.PerInstanceParams.x =
        GetInstanceRandom(Intermediates);
        float3 InstanceLocation = TransformLocalToWorld(GetInstanceOrigin(Intermediates),
        Intermediates.PrimitiveId).xyz;
        Intermediates.PerInstanceParams.y =1.0- saturate((length(InstanceLocation +
ResolvedView.PreViewTranslation.xyz) - InstancingFadeOutParams.x) * InstancingFadeOutParams.y);

        // InstancingFadeOutParams.z,w are RenderSelected and RenderDeselected respectively.
        Intermediates.PerInstanceParams.z = InstancingFadeOutParams.z * SelectedValue + InstancingFadeOutParams.w
                                        * (1-SelectedValue);
        #if USE_DITHERED_LOD_TRANSITION
            floatRandomLOD = InstancingViewZCompareZero.w *
            Intermediates.PerInstanceParams.x;
            floatViewZZero = length(InstanceLocation - InstancingWorldViewOriginZero.xyz) + RandomLOD;

            floatViewZOne = length(InstanceLocation - InstancingWorldViewOriginOne.xyz) + RandomLOD;

            Intermediates.PerInstanceParams.w =
                dot(float3(ViewZZero.xxx > InstancingViewZCompareZero.xyz),
InstancingViewZConstant.xyz) * InstancingWorldViewOriginZero.w +
                dot(float3(ViewZOne.xxx > InstancingViewZCompareOne.xyz),
InstancingViewZConstant.xyz) * InstancingWorldViewOriginOne.w;
            Intermediates.PerInstanceParams.z *=abs(Intermediates.PerInstanceParams.w) <
.999;
        #else
            Intermediates.PerInstanceParams.w =0;
        #endif
#elifUSE_INSTANCING && USE_INSTANCING_BONEMAP
        Intermediates.PerInstanceParams.x               =0;
        Intermediates.PerInstanceParams.y               =1;
        Intermediates.PerInstanceParams.z               =1;
        Intermediates.PerInstanceParams.w               = 0;
#endif          // USE_INSTANCING


        //GPUSkinning data.
#if     GPUSKIN_PASS_THROUGH
        uint PreSkinVertexOffset = LocalVF.PreSkinBaseVertexIndex + Input.VertexId *3;
        Intermediates.PreSkinPosition.x =
LocalVF.VertexFetch_PreSkinPositionBuffer[PreSkinVertexOffset                   + 0];
        Intermediates.PreSkinPosition.y =
        LocalVF.VertexFetch_PreSkinPositionBuffer[PreSkinVertexOffset           + 1];
        Intermediates.PreSkinPosition.z =
        LocalVF.VertexFetch_PreSkinPositionBuffer[PreSkinVertexOffset           + 2];
#else
        Intermediates.PreSkinPosition = Input.Position.xyz;
#endif
```

```
	returnIntermediates;
}

//Get the vertex parameters of the material.
FMaterialVertexParametersGetMaterialVertexParameters(FVertexFactoryInput Input, FVertexFactoryIntermediates
Intermediates, float3 WorldPosition, half3x3 TangentToLocal) {

	FMaterialVertexParameters Result = (FMaterialVertexParameters)0;
	Result.WorldPosition		=WorldPosition;
	Result.VertexColor =		Intermediates.Color;

	// does not handle instancing! Result.TangentToWorld =
	Intermediates.TangentToWorld;

	//Instantiation parameters.
#if	USE_INSTANCING
	Result.InstanceLocalToWorld		=mul(GetInstanceTransform(Intermediates),
	GetPrimitiveData(Intermediates.PrimitiveId).LocalToWorld);
	Result.InstanceLocalPosition = Input.Position.xyz; Result.PerInstanceParams =
	Intermediates.PerInstanceParams;

	Result.InstanceId = GetInstanceId(Input.InstanceId);

	#if	USE_INSTANCING_BONEMAP
		Result.PrevFrameLocalToWorld = mul(GetInstancePrevTransform(Intermediates),
		GetPrimitiveData(Intermediates.PrimitiveId).PreviousLocalToWorld);
	#else
		Result.PrevFrameLocalToWorld = mul(GetInstanceTransform(Intermediates),
		GetPrimitiveData(Intermediates.PrimitiveId).PreviousLocalToWorld);
	#endif// USE_INSTANCING_BONEMAP
#else
	Result.PrevFrameLocalToWorld =
	GetPrimitiveData(Intermediates.PrimitiveId).PreviousLocalToWorld;
#endif		// USE_INSTANCING

	//Skin data and normals from the previous frame.
	Result.PreSkinnedPosition = Intermediates.PreSkinPosition.xyz; Result.PreSkinnedNormal =
	TangentToLocal[2];//TangentBias(Input.TangentZ.xyz);

	//Process texture coordinate data.
#ifMANUAL_VERTEX_FETCH && NUM_MATERIAL_TEXCOORDS_VERTEX const
		uint NumFetchTexCoords =
		LocalVF.VertexFetch_Parameters[VF_NumTexcoords_Index];
		UNROLL
		for(uint CoordinateIndex =0; CoordinateIndex < NUM_MATERIAL_TEXCOORDS_VERTEX; CoordinateIndex++)

		{
			// Clamp coordinates to mesh's maximum as materials can request more than are
available
			uint ClampedCoordinateIndex = min(CoordinateIndex, NumFetchTexCoords-1);
			Result.TexCoords[CoordinateIndex] =
LocalVF.VertexFetch_TexCoordBuffer[NumFetchTexCoords *
(LocalVF.VertexFetch_Parameters[VF_VertexOffset] + Input.VertexId) +
ClampedCoordinateIndex];
		}
#elifNUM_MATERIAL_TEXCOORDS_VERTEX
		#ifGPUSKIN_PASS_THROUGH
```

```
                UNROLL
                for(intCoordinateIndex =0; CoordinateIndex < NUM_MATERIAL_TEXCOORDS_VERTEX;
CoordinateIndex++)
                {
                        Result.TexCoords[CoordinateIndex] = Input.TexCoords[CoordinateIndex].xy;
                }
        #else
                #ifNUM_MATERIAL_TEXCOORDS_VERTEX > 1
                  UNROLL
                        for(intCoordinateIndex =0; CoordinateIndex <
NUM_MATERIAL_TEXCOORDS_VERTEX-1; CoordinateIndex+=2)
                        {
                                Result.TexCoords[CoordinateIndex] =
Input.PackedTexCoords4[CoordinateIndex/2].xy;
                                if( CoordinateIndex+1< NUM_MATERIAL_TEXCOORDS_VERTEX ) {

                                        Result.TexCoords[CoordinateIndex+1] =
Input.PackedTexCoords4[CoordinateIndex/2].zw;
                                }
                        }
                #endif          // NUM_MATERIAL_TEXCOORDS_VERTEX > 1
                #ifNUM_MATERIAL_TEXCOORDS_VERTEX % 2 == 1
                        Result.TexCoords[NUM_MATERIAL_TEXCOORDS_VERTEX-1] =
Input.PackedTexCoords2;
                #endif          // NUM_MATERIAL_TEXCOORDS_VERTEX % 2 == 1
        #endif
#endif//MANUAL_VERTEX_FETCH && NUM_MATERIAL_TEXCOORDS_VERTEX

        //Graphicsid.
        Result.PrimitiveId              =  Intermediates.PrimitiveId;
        returnResult;
}

//Get vertex interpolation data.
FVertexFactoryInterpolantsVSToPSVertexFactoryGetInterpolantsVSToPS(FVertexFactoryInput Input,
FVertexFactoryIntermediates Intermediates, FMaterialVertexParameters VertexParameters)

{
        FVertexFactoryInterpolantsVSToPS Interpolants; Interpolants =
        (FVertexFactoryInterpolantsVSToPS)0;

        //Processing texture data.
#if     NUM_TEX_COORD_INTERPOLATORS
        float2 CustomizedUVs[NUM_TEX_COORD_INTERPOLATORS];
        GetMaterialCustomizedUVs(VertexParameters, CustomizedUVs);
        GetCustomInterpolators(VertexParameters, CustomizedUVs);

        UNROLL
        for(intCoordinateIndex =0; CoordinateIndex < NUM_TEX_COORD_INTERPOLATORS; CoordinateIndex++)

        {
                SetUV(Interpolants, CoordinateIndex, CustomizedUVs[CoordinateIndex]);
        }

#elifNUM_MATERIAL_TEXCOORDS_VERTEX == 0 && USE_PARTICLE_SUBUVS
        #if MANUAL_VERTEX_FETCH
                SetUV(Interpolants,             0,
LocalVF.VertexFetch_TexCoordBuffer[LocalVF.VertexFetch_Parameters[VF_NumTexcoords_Index]                *
```

```
(LocalVF.VertexFetch_Parameters[VF_VertexOffset] + Input.VertexId)]);
    #else
        SetUV(Interpolants,0, Input.TexCoords[0]);
    #endif
#endif

    //Light map related data: coordinates, index, offset, shadow
#if  coordinates, etc. NEEDS_LIGHTMAP_COORDINATE float2
    LightMapCoordinate =0; float2 ShadowMapCoordinate =0;

    #if MANUAL_VERTEX_FETCH
        float2 LightMapCoordinateInput =
LocalVF.VertexFetch_TexCoordBuffer[LocalVF.VertexFetch_Parameters[VF_NumTexcoords_Index]                    *
(LocalVF.VertexFetch_Parameters[VF_VertexOffset] + Input.VertexId) +
LocalVF.VertexFetch_Parameters[FV_LightMapIndex_Index]];
    #else
        float2 LightMapCoordinateInput = Input.LightMapCoordinate;
    #endif

    uint LightmapDataIndex =0;

#if  VF_USE_PRIMITIVE_SCENE_DATA
    LightmapDataIndex = GetPrimitiveData(Intermediates.PrimitiveId).LightmapDataIndex +
    LocalVF.LODLightmapDataIndex;
#endif

    float4 LightMapCoordinateScaleBias =
    GetLightmapData(LightmapDataIndex).LightMapCoordinateScaleBias;

    #if  USE_INSTANCING
        LightMapCoordinate = LightMapCoordinateInput * LightMapCoordinateScaleBias.xy +
        GetInstanceLightMapBias(Intermediates);
    #else
        LightMapCoordinate = LightMapCoordinateInput * LightMapCoordinateScaleBias.xy +
        LightMapCoordinateScaleBias.zw;
    #endif
    #if  STATICLIGHTING_TEXTUREMASK float4 ShadowMapCoordinateScaleBias =
        GetLightmapData(LightmapDataIndex).ShadowMapCoordinateScaleBias;


        #if USE_INSTANCING
            ShadowMapCoordinate = LightMapCoordinateInput *
ShadowMapCoordinateScaleBias.xy + GetInstanceShadowMapBias(Intermediates);
        #else
            ShadowMapCoordinate = LightMapCoordinateInput *
ShadowMapCoordinateScaleBias.xy + ShadowMapCoordinateScaleBias.zw;
        #endif
    #endif        // STATICLIGHTING_TEXTUREMASK

    SetLightMapCoordinate(Interpolants,          LightMapCoordinate, ShadowMapCoordinate);
    SetLightmapDataIndex(Interpolants,           LightmapDataIndex);
#endif        //NEEDS_LIGHTMAP_COORDINATE

     SetTangents(Interpolants, Intermediates.TangentToWorld[0],
 Intermediates.TangentToWorld[2], Intermediates.TangentToWorldSign);
    SetColor(Interpolants, Intermediates.Color);
#if USE_INSTANCING
    Interpolants.PerInstanceParams = Intermediates.PerInstanceParams;
```

```
#endif

    //Graphicsid.
    SetPrimitiveId(Interpolants, Intermediates.PrimitiveId);

    returnInterpolants;
}
```

5.3.3.2 BasePassPixelShader

The entry of BasePassPixelShader is in BasePassPixelShader.usf:

```
#include"Common.ush"

//Encapsulates basic variables under different macro definitions.
#ifMATERIALBLENDING_TRANSLUCENT || MATERIALBLENDING_ADDITIVE || MATERIALBLENDING_MODULATE
    #define   SceneTexturesStruct       TranslucentBasePass.SceneTextures
    #define   EyeAdaptationStruct       TranslucentBasePass
    #define   SceneColorCopyTexture         TranslucentBasePass.SceneColorCopyTexture
    #define   PreIntegratedGF               TranslucentBasePass.PreIntegratedGFTexture
    #if SUPPORTS_INDEPENDENT_SAMPLERS
        #define   PreIntegratedGFSampler          View.SharedBilinearClampedSampler
        #define SceneColorCopySampler             View.SharedBilinearClampedSampler
    #else
        #define   PreIntegratedGFSampler          TranslucentBasePass.PreIntegratedGFSampler
        #define   SceneColorCopySampler           TranslucentBasePass.SceneColorCopySampler
    #endif
#else
    #defineEyeAdaptationStruct          OpaqueBasePass
#endif

(.....)

#include   "SHCommon.ush"
#include   "BasePassCommon.ush"
#include   "BRDF.ush"
#include   "DeferredShadingCommon.ush"

(.....)

//Normal curvature converted to roughness.
float NormalCurvatureToRoughness(float3 {           WorldNormal)

    float3 dNdx = ddx(WorldNormal); float3
    dNdy = ddy(WorldNormal); floatx =
    dot(dNdx, dNdx); floaty = dot(dNdy,
    dNdy);
    floatCurvatureApprox =pow(max(x, y), View.NormalCurvatureToRoughnessScaleBias.z); return
    saturate(CurvatureApprox * View.NormalCurvatureToRoughnessScaleBias.x +
    View.NormalCurvatureToRoughnessScaleBias.y);
}

#if   TRANSLUCENT_SELF_SHADOWING
    #include"ShadowProjectionCommon.ush"
#endif

#include"ShadingModelsMaterial.ush"
```

```
#ifMATERIAL_SHADINGMODEL_HAIR || SIMPLE_FORWARD_DIRECTIONAL_LIGHT ||
  MATERIAL_SHADINGMODEL_SINGLELAYERWATER
#include"ShadingModels.ush"
#endif

(.....)

//Volumetric and transparent lighting.
#ifTRANSLUCENCY_LIGHTING_SURFACE_LIGHTINGVOLUME ||
TRANSLUCENCY_LIGHTING_SURFACE_FORWARDSHADING || FORWARD_SHADING ||
MATERIAL_SHADINGMODEL_SINGLELAYERWATER
#include"ForwardLightingCommon.ush"
#endif
#if!FORWARD_SHADING
void GetVolumeLightingNonDirectional(float4 AmbientLightingVector, ...); void
GetVolumeLightingDirectional(float4 AmbientLightingVector, ...);
float3GetTranslucencyVolumeLighting(FMaterialPixelParameters MaterialParameters, ...);
#endif

//Sky light.
#ifSIMPLE_FORWARD_SHADING || PLATFORM_FORCE_SIMPLE_SKY_DIFFUSE
      #define GetEffectiveSkySHDiffuse              GetSkySHDiffuseSimple
#else
      #define    GetEffectiveSkySHDiffuse          GetSkySHDiffuse
#endif
void GetSkyLighting(FMaterialPixelParameters MaterialParameters, ...);

//Indirect lighting.
#if   SUPPORTS_INDEPENDENT_SAMPLERS
      #define    ILCSharedSampler1      View.SharedBilinearClampedSampler
      #define ILCSharedSampler2         View.SharedBilinearClampedSampler
#else
      #define    ILCSharedSampler1      IndirectLightingCache.IndirectLightingCacheTextureSampler1
      #define ILCSharedSampler2         IndirectLightingCache.IndirectLightingCacheTextureSampler2
#endif
void GetPrecomputedIndirectLightingAndSkyLight(FMaterialPixelParameters
MaterialParameters, ...);

//Simple forward light.
#ifSIMPLE_FORWARD_DIRECTIONAL_LIGHT || MATERIAL_SHADINGMODEL_SINGLELAYERWATER float3
GetSimpleForwardLightingDirectionalLight(FGBufferData GBuffer, ...);
#endif

//Pixel depth offset.
void ApplyPixelDepthOffsetForBasePass(inout FMaterialPixelParameters MaterialParameters, . . . );

//simulationAOMultiple rebounds.
float3AOMultiBounce( float3 BaseColor,floatAO );
float DotSpecularSG(floatRoughness, float3 N, float3 V, FSphericalGaussian LightSG ); //Applies environment normals.

void ApplyBentNormal( in FMaterialPixelParameters MaterialParameters, infloatRoughness, . . . ) ;


//BasePassMain entry point for the pixel shader.
void FPixelShaderInOut_MainPS(
      FVertexFactoryInterpolantsVSToPS FBasePassInInteterprpoolalanntsts,VSToPS
      BasePassInterpolants, in FPixelShaderIn In,
```

```
    inout FPixelShaderOut Out)
{
    //initializationGBufferandviewAnd
    other data. constuint EyeIndex =0;
    ResolvedView = ResolveView();

    float4 OutVelocity =0; float4
    OutGBufferD =0; float4
    OutGBufferE =0;

    //Get the material parameters of the pixel (the material here is the material edited by the material editor).
    FMaterialPixelParameters MaterialParameters = GetMaterialPixelParameters(Interpolants, In.SvPosition);

    FPixelMaterialInputs PixelMaterialInputs;

    //Lightmap virtual texture.
    VTPageTableResult LightmapVTPageTableResult = (VTPageTableResult)0.0f;
#if LIGHTMAP_VT_ENABLED
    {
        float2 LightmapUV0, LightmapUV1; uint
        LightmapDataIndex;
        GetLightMapCoordinates(Interpolants, LightmapUV0, LightmapUV1, LightmapDataIndex);
        LightmapVTPageTableResult = LightmapGetVTSampleInfo(LightmapUV0,
LightmapDataIndex, In.SvPosition.xy);
    }
#endif

    //Lightmaps andAO.
#ifHQ_TEXTURE_LIGHTMAP && USES_AO_MATERIAL_MASK && !MATERIAL_SHADINGMODEL_UNLIT {

        float2 LightmapUV0, LightmapUV1; uint
        LightmapDataIndex;
        GetLightMapCoordinates(Interpolants, LightmapUV0, LightmapUV1, LightmapDataIndex); // Must be computed
        before BaseColor, Normal, etc are evaluated MaterialParameters.AOMaterialMask =
        GetAOMaterialMask(LightmapVTPageTableResult, LightmapUV0 * float2(1,2), LightmapDataIndex,
        In.SvPosition.xy);

    }
#endif

    //Additional parameters for the material.
#if USE_WORLD_POSITION_EXCLUDING_SHADER_OFFSETS {

        float4 ScreenPosition = SvPositionToResolvedScreenPosition(In.SvPosition); float3
        TranslatedWorldPosition =
SvPositionToResolvedTranslatedWorld(In.SvPosition);
        //Calculates additional material parameters.
        CalcMaterialParametersEx(MaterialParameters, PixelMaterialInputs, In.SvPosition, ScreenPosition,
In.bIsFrontFace, TranslatedWorldPosition,
BasePassInterpolants.PixelPositionExcludingWPO);
    }
#else
    {
        float4 ScreenPosition = SvPositionToResolvedScreenPosition(In.SvPosition); float3
        TranslatedWorldPosition =
SvPositionToResolvedTranslatedWorld(In.SvPosition);
        //Calculates additional material parameters.
        CalcMaterialParametersEx(MaterialParameters, PixelMaterialInputs, In.SvPosition, ScreenPosition,
In.bIsFrontFace, TranslatedWorldPosition, TranslatedWorldPosition);
```

```
    }
#endif

    //Pixel depth offset.
#if  OUTPUT_PIXEL_DEPTH_OFFSET
    ApplyPixelDepthOffsetForBasePass(MaterialParameters,                PixelMaterialInputs,
BasePassInterpolants, Out.Depth);
#endif

    //Processing pixelsclip.
#if  !EARLY_Z_PASS_ONLY_MATERIAL_MASKING if(!
    bEditorWeightedZBuffering) {

#if  MATERIALBLENDING_MASKED_USING_COVERAGE
        Out.Coverage = DiscardMaterialWithPixelCoverage(MaterialParameters,
        PixelMaterialInputs);
#else
        GetMaterialCoverageAndClipping(MaterialParameters, PixelMaterialInputs);
#endif
    }
#endif

    //Get the basic properties of a material.
    half3 BaseColor = GetMaterialBaseColor(PixelMaterialInputs); half
        Metallic    =GetMaterialMetallic(PixelMaterialInputs);
    half    Specular    = GetMaterialSpecular(PixelMaterialInputs);

    floatMaterialAO = GetMaterialAmbientOcclusion(PixelMaterialInputs); floatRoughness =
    GetMaterialRoughness(PixelMaterialInputs); floatAnisotropy =
    GetMaterialAnisotropy(PixelMaterialInputs); uint ShadingModel =
    GetMaterialShadingModel(PixelMaterialInputs);

    half Opacity = GetMaterialOpacity(PixelMaterialInputs);

    // 0..1, SubsurfaceProfileId = int(x * 255) float
    SubsurfaceProfile =0;

    float3 SubsurfaceColor =0;

    //Subsurface scattering.
#ifMATERIAL_SHADINGMODEL_SUBSURFACE || MATERIAL_SHADINGMODEL_PREINTEGRATED_SKIN ||
MATERIAL_SHADINGMODEL_SUBSURFACE_PROFILE || MATERIAL_SHADINGMODEL_TWOSIDED_FOLIAGE ||
MATERIAL_SHADINGMODEL_CLOTH || MATERIAL_SHADINGMODEL_EYE
    if(ShadingModel == SHADINGMODELID_SUBSURFACE || ShadingModel ==
SHADINGMODELID_PREINTEGRATED_SKIN || ShadingModel == SHADINGMODELID_SUBSURFACE_PROFILE ||
ShadingModel == SHADINGMODELID_TWOSIDED_FOLIAGE || ShadingModel == SHADINGMODELID_CLOTH ||
ShadingModel == SHAD INGMODELID_EYE)
    {
        float4 SubsurfaceData = GetMaterialSubsurfaceData(PixelMaterialInputs);

        if(false)// Dummy if to make the ifdef logic play nicely {

        }
#ifMATERIAL_SHADINGMODEL_SUBSURFACE || MATERIAL_SHADINGMODEL_PREINTEGRATED_SKIN ||
  MATERIAL_SHADINGMODEL_TWOSIDED_FOLIAGE
        else if(ShadingModel == SHADINGMODELID_SUBSURFACE || ShadingModel ==
SHADINGMODELID_PREINTEGRATED_SKIN || ShadingModel == SHADINGMODELID_TWOSIDED_FOLIAGE)
        {
```

```
                SubsurfaceColor = SubsurfaceData.rgb * View.DiffuseOverrideParameter.w +
View.DiffuseOverrideParameter.xyz;
            }
#endif
#ifMATERIAL_SHADINGMODEL_CLOTH
            else if(ShadingModel == SHADINGMODELID_CLOTH) {

                SubsurfaceColor = SubsurfaceData.rgb;
            }
#endif

            SubsurfaceProfile = SubsurfaceData.a;
        }
#endif

        //Decal data.
        floatDBufferOpacity =1.0f;
#ifUSE_DBUFFER && MATERIALDECALRESPONSEMASK && !MATERIALBLENDING_ANY_TRANSLUCENT && !
  MATERIAL_SHADINGMODEL_SINGLELAYERWATER
        (.....)
#endif

        const floatBaseMaterialCoverageOverWater = Opacity;
        const floatWaterVisibility =1.0- BaseMaterialCoverageOverWater;

        //Volumetric lightmap.
        float3 VolumetricLightmapBrickTextureUVs; PRECOMPUTED_IRRADIANCE_VOLUME_LIGHTING
#if     VolumetricLightmapBrickTextureUVs =
        ComputeVolumetricLightmapBrickTextureUVs(MaterialParameters.AbsoluteWorldPosition);

#endif

        //collectGBufferdata.
        FGBufferData GBuffer = (FGBufferData)0;

        GBuffer.GBufferAO = MaterialAO; GBuffer.PerObjectGBufferData =
        GetPrimitiveData(MaterialParameters.PrimitiveId).PerObjectGBufferData;

        GBuffer.Depth = MaterialParameters.ScreenPosition.w;
        GBuffer.PrecomputedShadowFactors =
GetPrecomputedShadowMasks(LightmapVTPageTableResult, Interpolants,
MaterialParameters.PrimitiveId, MaterialParameters.AbsoluteWorldPosition,
VolumetricLightmapBrickTextureUVs);

        const floatGBufferDither = InterleavedGradientNoise(MaterialParameters.SvPosition.xy,
        View.StateFrameIndexMod8);
        //Set the parameters collected earlier toGBuffermiddle.
        SetGBufferForShadingModel(GBuffer, MaterialParameters, Opacity,BaseColor, Metallic, Specular, Roughness,

Anisotropy, SubsurfaceColor, SubsurfaceProfile, GBufferDither, ShadingModel);


#if     USES_GBUFFER
        GBuffer.SelectiveOutputMask          =  GetSelectiveOutputMask();
        GBuffer.Velocity =0;
#endif

        //Speed buffer.
#if     WRITES_VELOCITY_TO_GBUFFER
```

```
        BRANCH
        if(GetPrimitiveData(MaterialParameters.PrimitiveId).OutputVelocity >0||
View.ForceDrawAllVelocities !=0)
        {
        #ifWRITES_VELOCITY_TO_GBUFFER_USE_POS_INTERPOLATOR
                float3 Velocity = Calculate3DVelocity(BasePassInterpolants.VelocityScreenPosition,
                BasePassInterpolants.VelocityPrevScreenPosition);
        #else
                float3 Velocity = Calculate3DVelocity(MaterialParameters.ScreenPosition,
                BasePassInterpolants.VelocityPrevScreenPosition);
        #endif

                float4 EncodedVelocity = EncodeVelocityToTexture(Velocity);

                FLATTEN
                if(GetPrimitiveData(MaterialParameters.PrimitiveId).DrawsVelocity ==0.0&&
                View.ForceDrawAllVelocities== 0)
                {
                        Encoded Velocity       = 0.0;
                }

        #if   USES_GBUFFER
                GBuffer.Velocity          =  EncodedVelocity;
        #else
                OutVelocity = EncodedVelocity;
        #endif
        }
#endif

        //Specular color.
        GBuffer.SpecularColor = ComputeF0(Specular, BaseColor, Metallic);

        //Normal curvature converted to roughness.
#i      MATERIAL_NORMAL_CURVATURE_TO_ROUGHNESS
f       USE_WORLDVERTEXNORMAL_CENTER_INTERPOLATION floatGeometricAARoughness =
#i      NormalCurvatureToRoughness(MaterialParameters.WorldVertexNormal_Center);
f
#else
        floatGeometricAARoughness =
NormalCurvatureToRoughness(MaterialParameters.TangentToWorld[2].xyz);
#endif
        GBuffer.Roughness = max(GBuffer.Roughness, GeometricAARoughness);

#if   MATERIAL_SHADINGMODEL_CLEAR_COAT
        if(GBuffer.ShadingModelID == SHADINGMODELID_CLEAR_COAT) {

                GBuffer.CustomData.y = max(GBuffer.CustomData.y, GeometricAARoughness);
        }
#endif
#endif

        //Post-processing subsurface scattering.
#if   POST_PROCESS_SUBSURFACE
        // SubsurfaceProfile applies the BaseColor in a later pass. Any lighting output in the pass needs
base
        // to separate specular and diffuse lighting in a checkerboard pattern boolbChecker =
        CheckerFromPixelPos(MaterialParameters.SvPosition.xy); if
        (UseSubsurfaceProfile(GBuffer.ShadingModelID))
```

```
	{
		AdjustBaseColorAndSpecularColorForSubsurfaceProfileLighting(BaseColor,
GBuffer.SpecularColor, Specular, bChecker);
	}
#endif

	//Diffuse color.
	GBuffer.DiffuseColor = BaseColor - BaseColor * Metallic;

	//Simulation EnvironmentBRDFApproximates a fully rough effect.
	#if	!FORCE_FULLY_ROUGH
	if	(View.RenderingReflectionCaptureMask)
	#endif
	{
		EnvBRDFApproxFullyRough(GBuffer.DiffuseColor,		GBuffer.SpecularColor);
	}

	//Environment normal.
	float3 BentNormal = MaterialParameters.WorldNormal; // Clear Coat
	Bottom Normal
	BRANCHif(GBuffer.ShadingModelID == SHADINGMODELID_CLEAR_COAT &&
	CLEAR_COAT_BOTTOM_NORMAL)
	{
		constfloat2 oct1 = ((float2(GBuffer.CustomData.a, GBuffer.CustomData.z) *2) - (256.0/255.0)) +
		UnitVectorToOctahedron(GBuffer.WorldNormal);
		BentNormal = OctahedronToUnitVector(oct1);
	}

	//deal withAO.
	floatDiffOcclusion = MaterialAO; float
	SpecOcclusion = MaterialAO; //Applies
	environment normals.
	ApplyBentNormal( MaterialParameters, GBuffer.Roughness, BentNormal, DiffOcclusion, SpecOcclusion );

	//FIXME:ALLOW_STATIC_LIGHTING == 0 expects this to be AO
	GBuffer.GBufferAO = AOMultiBounce( Luminance( GBuffer.SpecularColor ), SpecOcclusion
) . g;

	half3 DiffuseColor =0; half3
	Color =0;
	floatIndirectIrradiance =0;

	half3 ColorSeparateSpecular =0; half3
	ColorSeparateEmissive =0;

	//NoUnlitLighting Model,	The subsurface scattering color is added to the diffuse color.
	#if	!MATERIAL_SHADINGMODEL_UNLIT float3
		DiffuseDir = BentNormal;
		float3 DiffuseColorForIndirect = GBuffer.DiffuseColor;

		//Subsurface scattering and precomputed skin shading models.
		#ifMATERIAL_SHADINGMODEL_SUBSURFACE || MATERIAL_SHADINGMODEL_PREINTEGRATED_SKIN if
		(GBuffer.ShadingModelID == SHADINGMODELID_SUBSURFACE || GBuffer.ShadingModelID
==		SHADINGMODELID_PREINTEGRATED_SKIN)
		{
			// Add subsurface energy to diffuse
			//@todo - better subsurface handling for these shading models with skylight precomputed GI
and
```

```
            DiffuseColorForIndirect += SubsurfaceColor;
        }
        #endif

        //Cloth shading model.
        #ifMATERIAL_SHADINGMODEL_CLOTH
        if(GBuffer.ShadingModelID == SHADINGMODELID_CLOTH) {

            DiffuseColorForIndirect += SubsurfaceColor *
saturate(GetMaterialCustomData0(MaterialParameters));
        }
        #endif

        //Hair shading model.
        #ifMATERIAL_SHADINGMODEL_HAIR
        if(GBuffer.ShadingModelID == SHADINGMODELID_HAIR) {

            FHairTransmittanceData TransmittanceData = InitHairTransmittanceData(true); float3 N =
            MaterialParameters.WorldNormal;
            float3 V = MaterialParameters.CameraVector; float3 L =
            normalize( V - N * dot(V,N) ); DiffuseDir = L;

            DiffuseColorForIndirect=2*PI * HairShading( GBuffer, L, V, N,1,
TransmittanceData,0,0.2, uint2(0,0));

            #ifUSE_HAIR_COMPLEX_TRANSMITTANCE
            GBuffer.CustomData.a =1.f/255.f;
            #endif
        }
        #endif

        //Computes precomputed indirect lighting and skylight.
        float3      DiffuseIndirectLighting;
        float3      SubsurfaceIndirectLighting;
        GetPrecomputedIndirectLightingAndSkyLight(MaterialParameters, BasePassInterpolIanntetsrp, olants,
LightmapVTPageTableResult, GBuffer, DiffuseDir, VolumetricLightmapBrickTextureUVs, DiffuseIndirectLighting,
SubsurfaceIndirectLighting, IndirectIrradiance);



        //Indirect lighting applicationsAO.
        floatIndirectOcclusion =1.0f; float2
        NearestResolvedDepthScreenUV =0; float
        DirectionalLightShadow =1.0f;

        #ifFORWARD_SHADING && (MATERIALBLENDING_SOLID || MATERIALBLENDING_MASKED) float2
            NDC = MaterialParameters.ScreenPosition.xy /
MaterialParameters.ScreenPosition.w;
            float2 ScreenUV = NDC * ResolvedView.ScreenPositionScaleBias.xy +
ResolvedView.ScreenPositionScaleBias.wz;
            NearestResolvedDepthScreenUV          = CalculateNearestResolvedDepthScreenUV(ScreenUV,
MaterialParameters.ScreenPosition.w);

            IndirectOcclusion = GetIndirectOcclusion(NearestResolvedDepthScreenUV,
GBuffer);
            DiffuseIndirectLighting *= SubsurfaceIenInddiriercetcOtLcicglhutsiinogn ;
            IndirectIrradiance *= IndirectOcclusio* n=; IndirectOcclusion;

        #endif
```

```
//Final diffuse reflection.
DiffuseColor += (DiffuseIndirectLighting * DiffuseColorForIndirect + SubsurfaceIndirectLighting *
SubsurfaceColor) * AOMultiBounce( GBuffer.BaseColor, DiffOcclusion );


//Forward rendering or transparent object lighting.
#if TRANSLUCENCY_PERVERTEX_FORWARD_SHADING
    Color += BasePassInterpolants.VertexDiffuseLighting * GBuffer.DiffuseColor;
#elifFORWARD_SHADING || TRANSLUCENCY_LIGHTING_SURFACE_FORWARDSHADING ||
TRANSLUCENCY_LIGHTING_SURFACE_LIGHTINGVOLUME || MATERIAL_SHADINGMODEL_SINGLELAYERWATER
    uint GridIndex =0;

    #ifFEATURE_LEVEL >= FEATURE_LEVEL_SM5 GridIndex =
      ComputeLightGridCellIndex((uint2)
(MaterialParameters.SvPosition.xy * View.LightProbeSizeRatioAndInvSizeRatio.zw -
ResolvedView.ViewRectMin.xy), MaterialParameters.SvPosition.w, EyeIndex);

        #ifFORWARD_SHADING || TRANSLUCENCY_LIGHTING_SURFACE_FORWARDSHADING ||
MATERIAL_SHADINGMODEL_SINGLELAYERWATER
            const floatDither =
InterleavedGradientNoise(MaterialParameters.SvPosition.xy,          View.StateFrameIndexMod8);
            FDeferredLightingSplit      ForwardDirectLighting =
GetForwardDirectLightingSplit(GridIndex, MaterialParamMeatteerrsia.ClPaamraemraeVteecrtso.Ar,b GsBouluftfeWr, orldPosition,
MaterialParameters.PrimitiveId, EyeIndex, Dither, DirectNioenaarelLsitgRhetsSohlavdedoDwe);pthScreenUV,


            #if MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT
                DiffuseColor += ForwardDirectLighting.DiffuseLighting.rgb;
                ColorSeparateSpecular      + =
ForwardDirectLighting.SpecularLighting.rgb;
            #else
                Color    + = ForwardDirectLighting.DiffuseLighting.rgb;
                Color    + = ForwardDirectLighting.SpecularLighting.rgb;
            #endif
        #endif
    #endif
    (......)
#endif

//Simple forward directional lighting.
#ifSIMPLE_FORWARD_DIRECTIONAL_LIGHT && !MATERIAL_SHADINGMODEL_SINGLELAYERWATER &&
!MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT
    float3 DirectionalLighting = GetSimpleForwardLightingDirectionalLight(
        GBuffer,
        DiffuseColorForIndirect,
        GBuffer.SpecularColor,
        GBuffer.Roughness,
        MaterialParameters.WorldNormal,
        MaterialParameters.CameraVector);

    #if   STATICLIGHTING_SIGNEDDISTANCEFIELD DirectionalLighting *=
        GBuffer.PrecomputedShadowFactors.x;
    #elifPRECOMPUTED_IRRADIANCE_VOLUME_LIGHTING
        DirectionalLighting *=
GetVolumetricLightmapDirectionalLightShadowing(VolumetricLightmapBrickTextureUVs);
    #elifCACHED_POINT_INDIRECT_LIGHTING
        DirectionalLighting *= IndirectLightingCache.DirectionalLightShadowing;
```

```
                #endif

                Color += DirectionalLighting;

            #endif
        #endif

        //Fog effect.
        #if NEEDS_BASEPASS_VERTEX_FOGGING
            float4 HeightFogging = BasePassInterpolants.VertexFog;
        #elifNEEDS_BASEPASS_PIXEL_FOGGING float4
            HeightFogging =
CalculateHeightFog(MaterialParameters.WorldPosition_CamRelative);
        #else
            float4 HeightFogging = float4(0,0,0,1);
        #endif

        float4 Fogging = HeightFogging;

        //Volumetric fog.
#ifNEEDS_BASEPASS_PIXEL_VOLUMETRIC_FOGGING && COMPILE_BASEPASS_PIXEL_VOLUMETRIC_FOGGING if
        (FogStruct.ApplyVolumetricFog >0) {

            float3 VolumeUV = ComputeVolumeUV(MaterialParameters.AbsoluteWorldPosition,
            ResolvedView.WorldToClip);
            Fogging = CombineVolumetricFog(HeightFogging, VolumeUV, EyeIndex);
        }
#endif

        //Per-pixel fog.
#if NEEDS_BASEPASS_PIXEL_FOGGING
        const floatOneOverPreExposure = USE_PREEXPOSURE ? ResolvedView.OneOverPreExposure :
1.0f;
        float4 NDCPosition = mul(float4(MaterialParameters.AbsoluteWorldPosition.xyz,1),
        ResolvedView.WorldToClip);
#endif

        //Sky atmospheric effect.
#ifNEEDS_BASEPASS_PIXEL_FOGGING && PROJECT_SUPPORT_SKY_ATMOSPHERE && MATERIAL_IS_SKY==0 if
        (ResolvedView.SkyAtmosphereApplyCameraAerialPerspectiveVolume >0.0f) {

            Fogging = GetAerialPerspectiveLuminanceTransmittanceWithFogOver(
                ResolvedView.RealTimeReflectionCapture,
ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeSizeAndInvSize,
                NDCPosition, MaterialParameters.AbsoluteWorldPosition.xyz*CM_TO_SKY_UNIT,
ResolvedView.WorldCameraOrigin.xyz*CM_TO_SKY_UNIT,
                View.CameraAerialPerspectiveVolume, View.CameraAerialPerspectiveVolumeSampler,
                ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthResolutionInv,
                ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthResolution,
                ResolvedView.SkyAtmosphereAerialPerspectiveStartDepthKm,
                ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthSliceLengthKm,
                ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthSliceLengthKmInv,
                OneOverPreExposure, Fogging);

        }
#endif

        //Clouds and fog.
#ifNEEDS_BASEPASS_PIXEL_FOGGING && MATERIAL_ENABLE_TRANSLUCENCY_CLOUD_FOGGING
```

```
    if(TranslucentBasePass.ApplyVolumetricCloudOnTransparent >0.0f) {

        Fogging = GetCloudLuminanceTransmittanceOverFog(
            NDCPosition, MaterialParameters.AbsoluteWorldPosition.xyz,
ResolvedView.WorldCameraOrigin.xyz,
            TranslucentBasePass.VolumetricCloudColor,
TranslucentBasePass.VolumetricCloudColorSampler,
            TranslucentBasePass.VolumetricCloudDepth,
TranslucentBasePass.VolumetricCloudDepthSampler,
            OneOverPreExposure, Fogging);
    }
#endif

    //Volumetric lighting of transparent objects.
#if(MATERIAL_SHADINGMODEL_DEFAULT_LIT || MATERIAL_SHADINGMODEL_SUBSURFACE) &&
(MATERIALBLENDING_TRANSLUCENT || MATERIALBLENDING_ADDITIVE) && !SIMPLE_FORWARD_SHADING && !
FORWARD_SHADING
    if(GBuffer.ShadingModelID == SHADINGMODELID_DEFAULT_LIT || GBuffer.ShadingModelID ==
    SHADINGMODELID_SUBSURFACE)
    {
        Color += GetTranslucencyVolumeLighting(MaterialParameters, PixelMaterialInputs, BasePassInterpolants,
GBuffer, IndirectIrradiance);
    }
#endif

    (......)
#endif

    //If it is a subsurface scattering or thin transparent shading model, you need to keepDiffuseColorSeparate states, and the diffuse should not be added directly to the
final color.
#if!POST_PROCESS_SUBSURFACE && !MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT Color +=
  DiffuseColor;
#endif

    //Self-luminous.
#if  !MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT
    Color += Emissive;
#endif

(......)

    // THIN_TRANSLUCENTLighting model.
#if  MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT
    float3 DualBlendColorAdd =0.0f; float3
    DualBlendColorMul =1.0f;

    {
        AccumulateThinTranslucentModel(DualBlendColorAdd,          DualBlendColorMul,
MaterialParameters, GBuffer, ...);

        Color =    0;
        Opacity    =1.0f;
    }
#endif

    //saveGBufferarriveMRTmiddle,    SavedRTThe position varies according to different shading models.
    #if  MATERIAL_DOMAIN_POSTPROCESS
        #ifMATERIAL_OUTPUT_OPACITY_AS_ALPHA
```

```
                Out.MRT[0] = half4(Color, Opacity);
            #else
                Out.MRT[0] = half4(Color,0);
            #endif
            Out.MRT[0] = RETURN_COLOR(Out.MRT[0]);
        // MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT must come first because it also has
MATERIALBLENDING_TRANSLUCENT defined
        #elifMATERIAL_SHADINGMODEL_THIN_TRANSLUCENT
            float3 AdjustedDualBlendAdd = Fogging.rgb + Fogging.a * DualBlendColorAdd; float3
            AdjustedDualBlendMul =                                    Fogging.a *DualBlendColorMul;

            #if    THIN_TRANSLUCENT_USE_DUAL_BLEND
                // no RETURN_COLOR because these values are explicit multiplies and adds Out.MRT[0]
                                =half4(AdjustedDualBlendAdd,0.0);
                Out.MRT[1]     = half4(AdjustedDualBlendMul,1.0);
            #else
                // In the fallback case, we are blending with the mode floatAdjustedAlpha
                = saturate(1-
dot(AdjustedDualBlendMul,float3(1.0f,1.0f,1.0f)/3.0f));
                Out.MRT[0]     =half4(AdjustedDualBlendAdd,AdjustedAlpha);
                Out.MRT[0]     = RETURN_COLOR(Out.MRT[0]);
            #endif
        #elifMATERIALBLENDING_ALPHAHOLDOUT //
            not implemented for holdout
            Out.MRT[0] = half4(Color * Fogging.a + Fogging.rgb * Opacity, Opacity); Out.MRT[0] =
            RETURN_COLOR(Out.MRT[0]);
        #elifMATERIALBLENDING_ALPHACOMPOSITE
            Out.MRT[0] = half4(Color * Fogging.a + Fogging.rgb * Opacity, Opacity); Out.MRT[0] =
            RETURN_COLOR(Out.MRT[0]);
        #elifMATERIALBLENDING_TRANSLUCENT
            Out.MRT[0] = half4(Color * Fogging.a + Fogging.rgb, Opacity); Out.MRT[0] =
            RETURN_COLOR(Out.MRT[0]);
        #elifMATERIALBLENDING_ADDITIVE
            Out.MRT[0] = half4(Color * Fogging.a * Opacity,0.0f); Out.MRT[0] =
            RETURN_COLOR(Out.MRT[0]);
        #elifMATERIALBLENDING_MODULATE
            // RETURN_COLOR not needed with modulative blending
            half3 FoggedColor = lerp(float3(1,1,1), Color, Fogging.aaa * Fogging.aaa); Out.MRT[0] =
            half4(FoggedColor, Opacity);
        #else    //Other shading models.
            {
                FLightAccumulator LightAccumulator = (FLightAccumulator)0; Color = Color *
                Fogging.a + Fogging.rgb;

#ifPOST_PROCESS_SUBSURFACE
                DiffuseColor = DiffuseColor * Fogging.a + Fogging.rgb;

                if(UseSubsurfaceProfile(GBuffer.ShadingModelID) &&
                    View.bSubsurfacePostprocessEnabled >0&&
View.bCheckerboardSubsurfaceProfileRendering >0)
                {
                    Color *= !bChecker;
                }
                LightAccumulator_Add(LightAccumulator, Color + DiffuseColor, DiffuseColor,
1.0f, UseSubsurfaceProfile(GBuffer.ShadingModelID));
#else
                LightAccumulator_Add(LightAccumulator, Color,0,1.0f,false);
#endif
```

```
                Out.MRT[0] = RETURN_COLOR(LightAccumulator_GetResult(LightAccumulator));

                #if   !USES_GBUFFER
                      Out.MRT[0].a =       0;
                #endif
           }
      #endif

      //WillGBufferData is encoded intoMRTmiddle.
      #if   USES_GBUFFER
           GBuffer.IndirectIrradiance             =  IndirectIrradiance;

           // -0.5 .. 0.5, could be optimized as lower quality noise would be sufficient floatQuantizationBias =
           PseudoRandom( MaterialParameters.SvPosition.xy ) -0.5f; EncodeGBuffer(GBuffer, Out.MRT[1], Out.MRT[2],
           Out.MRT[3], OutGBufferD, OutGBufferE, OutVelocity, QuantizationBias);

      #endif

      (......)

      //Speed Cache and BeyondGBuffer.
#if   USES_GBUFFER
      #if GBUFFER_HAS_VELOCITY
           Out.MRT[4] = OutVelocity;
      #endif

      Out.MRT[GBUFFER_HAS_VELOCITY?5:4] = OutGBufferD;

      #if   GBUFFER_HAS_PRECSHADOWFACTOR
           Out.MRT[GBUFFER_HAS_VELOCITY?6:5] = OutGBufferE;
      #endif
#else//NoGBuffer,This means that it is forward rendering, but the speed data can still be written toMRT[1]middle.
      #if   GBUFFER_HAS_VELOCITY
           Out.MRT[1] = OutVelocity;
      #endif
#endif

      //Processing pre-exposure.
#if!MATERIALBLENDING_MODULATE && USE_PREEXPOSURE
      #if MATERIAL_IS_SKY
           // Dynamic capture exposure is 1 as of today.
           const floatViewPreExposure = View.RealTimeReflectionCapture>0.0f?
View.RealTimeReflectionCapturePreExposure : View.PreExposure;
      #else
           const floatViewPreExposure = View.PreExposure;
      #endif

      #ifMATERIAL_DOMAIN_POSTPROCESS || MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT ||
MATERIALBLENDING_ALPHAHOLDOUT || MATERIALBLENDING_ALPHACOMPOSITE ||
MATERIALBLENDING_TRANSLUCENT    || MATERIALBLENDING_ADDITIVE
           Out.MRT[0].rgb        * =     ViewPreExposure;
      #else
           Out.MRT[0].rgba       * =  ViewPreExposure;
      #endif
#endif

      (......)
}
```

```
#ifNUM_VIRTUALTEXTURE_SAMPLES || LIGHTMAP_VT_ENABLED
	#if COMPILER_SUPPORTS_DEPTHSTENCIL_EARLYTEST_LATEWRITE
		#definePIXELSHADER_EARLYDEPTHSTENCIL		DEPTHSTENCIL_EARLYTEST_LATEWRITE
	#elif!OUTPUT_PIXEL_DEPTH_OFFSET
		#definePIXELSHADER_EARLYDEPTHSTENCIL		EARLYDEPTHSTENCIL
	#endif
#endif

//MRTSerial number macro definition, must be the same asFSceneRenderTargets::GetGBufferRenderTargets()Keep matching.
#definePIXELSHADEROUTPUT_BASEPASS 1
#ifUSES_GBUFFER
#definePIXELSHADEROUTPUT_MRT0 (!SELECTIVE_BASEPASS_OUTPUTS ||
NEEDS_BASEPASS_VERTEX_FOGGING || USES_EMISSIVE_COLOR || ALLOW_STATIC_LIGHTING ||
MATERIAL_SHADINGMODEL_SINGLELAYERWATER)
#definePIXELSHADEROUTPUT_MRT1 ((!SELECTIVE_BASEPASS_OUTPUTS || !
 MATERIAL_SHADINGMODEL_UNLIT))
#definePIXELSHADEROUTPUT_MRT2 ((!SELECTIVE_BASEPASS_OUTPUTS || !
 MATERIAL_SHADINGMODEL_UNLIT))
#definePIXELSHADEROUTPUT_MRT3 ((!SELECTIVE_BASEPASS_OUTPUTS || !
 MATERIAL_SHADINGMODEL_UNLIT))
	#if GBUFFER_HAS_VELOCITY
		#definePIXELSHADEROUTPUT_MRT4 WRITES_VELOCITY_TO_GBUFFER
		#definePIXELSHADEROUTPUT_MRT5 (!SELECTIVE_BASEPASS_OUTPUTS ||
WRITES_CUSTOMDATA_TO_GBUFFER)
		#definePIXELSHADEROUTPUT_MRT6 (GBUFFER_HAS_PRECSHADOWFACTOR &&
(!SELECTIVE_BASEPASS_OUTPUTS || WRITES_PRECSHADOWFACTOR_TO_GBUFFER && !
MATERIAL_SHADINGMODEL_UNLIT))
	#else//GBUFFER_HAS_VELOCITY
		#definePIXELSHADEROUTPUT_MRT4 (!SELECTIVE_BASEPASS_OUTPUTS ||
WRITES_CUSTOMDATA_TO_GBUFFER)
		#definePIXELSHADEROUTPUT_MRT5 (GBUFFER_HAS_PRECSHADOWFACTOR &&
(!SELECTIVE_BASEPASS_OUTPUTS || WRITES_PRECSHADOWFACTOR_TO_GBUFFER && !
MATERIAL_SHADINGMODEL_UNLIT))
	#endif//GBUFFER_HAS_VELOCITY
#else//USES_GBUFFER
	#definePIXELSHADEROUTPUT_MRT0 1
	// we also need MRT for thin translucency due to dual blending if we are not on the fallback path

	#definePIXELSHADEROUTPUT_MRT1 (WRITES_VELOCITY_TO_GBUFFER ||
(MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT && THIN_TRANSLUCENT_USE_DUAL_BLEND))
#endif//USES_GBUFFER
#definePIXELSHADEROUTPUT_A2C ((EDITOR_ALPHA2COVERAGE) != 0)
#definePIXELSHADEROUTPUT_COVERAGE (MATERIALBLENDING_MASKED_USING_COVERAGE && !
 EARLY_Z_PASS_ONLY_MATERIAL_MASKING)


//The code in the following file will call the aboveFPixelShaderInOut_MainPSInterface andGBuffercorrespondMRTThe output of .
#include"PixelShaderOutputCommon.ush"
```

According to the above code analysis, the main steps of BasePassPixelShader are summarized:

- Initialize ResolvedView, GBuffer and other data.
- GetMaterialPixelParameters: Gets the material's parameters from the material node and interpolation data edited by the material editor. The detailed data is defined by

FMaterialPixelParameters:

```
// Engine\Shaders\Private\MaterialTemplate.ush

structFMaterialPixelParameters {

#ifNUM_TEX_COORD_INTERPOLATORS
	float2 TexCoords[NUM_TEX_COORD_INTERPOLATORS];
#endif

	half4	VertexColor;
	half3 WorldNormal;
	half3 WorldTangent;
	half3	ReflectionVector;
	half3 CameraVector;
	half3	LightVector;
	float4	SvPosition;
	float4	ScreenPosition;
	half UnMirrored;
	half TwoSidedSign;
	half3x3	TangentToWorld;

#if	USE_WORLDVERTEXNORMAL_CENTER_INTERPOLATION
	half3 WorldVertexNormal_Center;
#endif

	float3	AbsoluteWorldPosition;
	float3	WorldPosition_CamRelative;
	float3	WorldPosition_NoOffsets;
	float3	WorldPosition_NoOffsets_CamRelative;
	half3	LightingPositionOffset;

	floatAOMaterialMask;

#if	LIGHTMAP_UV_ACCESS
	float2 LightmapUVs;
#endif

#if	USE_INSTANCING
	half4 PerInstanceParams;
#endif

	uint PrimitiveId;

	(......)
};
```

- CalcMaterialParametersEx: Calculates additional material parameters based on FMaterialPixelParameters.
- Calculates GBuffer data based on the data in FMaterialPixelParameters and the data interpolated from the vertices.
  Calculates data for subsurface scattering, decals, volumetric lightmaps, and more.
- SetGBufferForShadingModel: Set the parameters collected in the previous step to GBuffer.
-

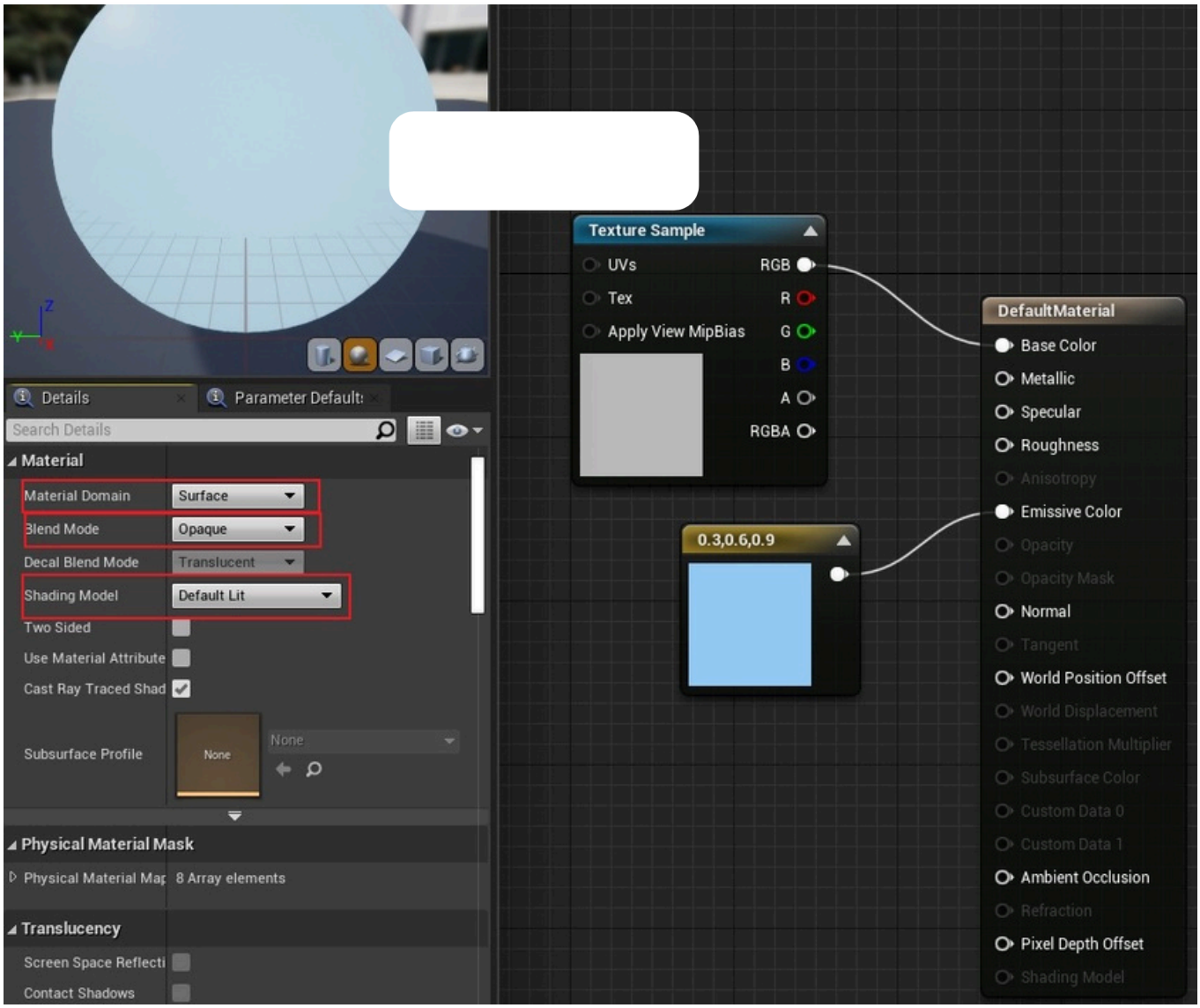
- Calculates or processes velocity buffer, specular color, diffuse color, ambient normal, AO, and adds subsurface scattering color to diffuse.
- Processes precomputed indirect lighting and skylight, indirect AO, and calculates the final diffuse term.
- Handle forward rendering lighting or parallel light, and handle lighting of transparent objects.
- Calculate fog effects, including height fog, exponential fog, volumetric fog, per-pixel fog, cloud fog, and sky atmospheric effects.
- Handling self-illumination.
- EncodeGBuffer: Encodes GBuffer base data into MRT.
- Write the non-basic data of GBuffer to the following RTs, such as velocity buffer, per-object shadows, etc.
- Processing pre-exposure.
-

Although the pixel shader of BasePass does not calculate dynamic lighting, it does not mean that it does not perform lighting processing. It mainly processes static light, ambient light, indirect light, fog effects, subsurface scattering and self-illumination.

5.3.4 BasePass Summary

After the analysis in the previous section, we can conclude that although the VS and PS of BasePass do not calculate dynamic lighting, they actually do a lot of processing related to geometric objects. The number of shader instructions generated is quite considerable. How to improve the rendering efficiency of BasePass is a topic worthy of in-depth study.

Next, we will take the material with ShadingModel DefaultLit as the research object, use RenderDoc to capture frames, and analyze the BasePassPixelShader it uses. The specific settings of the material are as follows:



Select the FPixelShaderInOut_MainPS code snippet from the intercepted frame data as follows:

```
void FPixelShaderInOut_MainPS(FVertexFactoryInterpolantsVSToPS Interpolants,
FBasePassInterpolantsVSToPS BasePassInterpolants, in FPixelShaderIn In, inout FPixelShaderOut Out)

{
	constuint EyeIndex =0; ResolvedView =
	ResolveView();

	float4 OutVelocity =0; float4
	OutGBufferD =0; float4
	OutGBufferE =0; float4
	OutGBufferF =0;

	FMaterialPixelParameters MaterialParameters = GetMaterialPixelParameters(Interpolants, In.SvPosition);

	FPixelMaterialInputs PixelMaterialInputs;

	floatLightmapVTPageTableResult = (float)0.0f; {

		float4 ScreenPosition = SvPositionToResolvedScreenPosition(In.SvPosition); float3
		TranslatedWorldPosition =
SvPositionToResolvedTranslatedWorld(In.SvPosition);
		CalcMaterialParametersEx(MaterialParameters, PixelMaterialInputs, In.SvPosition, ScreenPosition,
In.bIsFrontFace, TranslatedWorldPosition, TranslatedWorldPosition);
```

```
	}

	const boolbEditorWeightedZBuffering =false;

	if(!bEditorWeightedZBuffering) {

		GetMaterialCoverageAndClipping(MaterialParameters, PixelMaterialInputs);
	}

	float3	BaseColor	= GetMaterialBaseColor(PixelMaterialInputs);
	float	Metallic	=GetMaterialMetallic(PixelMaterialInputs);
	float	Specular	= GetMaterialSpecular(PixelMaterialInputs);
	floatMaterialAO = GetMaterialAmbientOcclusion(PixelMaterialInputs); floatRoughness =
	GetMaterialRoughness(PixelMaterialInputs); floatAnisotropy =
	GetMaterialAnisotropy(PixelMaterialInputs); uint ShadingModel =
	GetMaterialShadingModel(PixelMaterialInputs); floatOpacity =
	GetMaterialOpacity(PixelMaterialInputs); floatSubsurfaceProfile =0; float3 SubsurfaceColor =
	0; floatDBufferOpacity =1.0f;



	if(GetPrimitiveData(MaterialParameters.PrimitiveId).DecalReceiverMask >0&& View_ShowDecalsMask >0)

	{
		uint DBufferMask =0x07; if
		(DBufferMask)
		{
			float2 NDC = MaterialParameters.ScreenPosition.xy /
MaterialParameters.ScreenPosition.w;
			float2 ScreenUV = NDC * ResolvedView.ScreenPositionScaleBias.xy +
ResolvedView.ScreenPositionScaleBias.wz;
			FDBufferData DBufferData = GetDBufferData(ScreenUV, DBufferMask);

			ApplyDBufferData(DBufferData, MaterialParameters.WorldNormal, SubsurfaceColor,
Roughness, BaseColor, Metallic, Specular);
			DBufferOpacity = (DBufferData.ColorOpacity + DBufferData.NormalOpacity +
DBufferData.RoughnessOpacity) * (1.0f/3.0f);
		}
	}

	const floatBaseMaterialCoverageOverWater = Opacity;
	const floatWaterVisibility =1.0- BaseMaterialCoverageOverWater;

	float3 VolumetricLightmapBrickTextureUVs; VolumetricLightmapBrickTextureUVs =
	ComputeVolumetricLightmapBrickTextureUVs(MaterialParameters.AbsoluteWorldPosition);


	FGBufferData GBuffer = (FGBufferData)0;

	GBuffer.GBufferAO = MaterialAO; GBuffer.PerObjectGBufferData =
	GetPrimitiveData(MaterialParameters.PrimitiveId).PerObjectGBufferData;

	GBuffer.Depth = MaterialParameters.ScreenPosition.w;
	GBuffer.PrecomputedShadowFactors =
GetPrecomputedShadowMasks(LightmapVTPageTableResult, Interpolants,
MaterialParameters.PrimitiveId, MaterialParameters.AbsoluteWorldPosition,
VolumetricLightmapBrickTextureUVs);
```

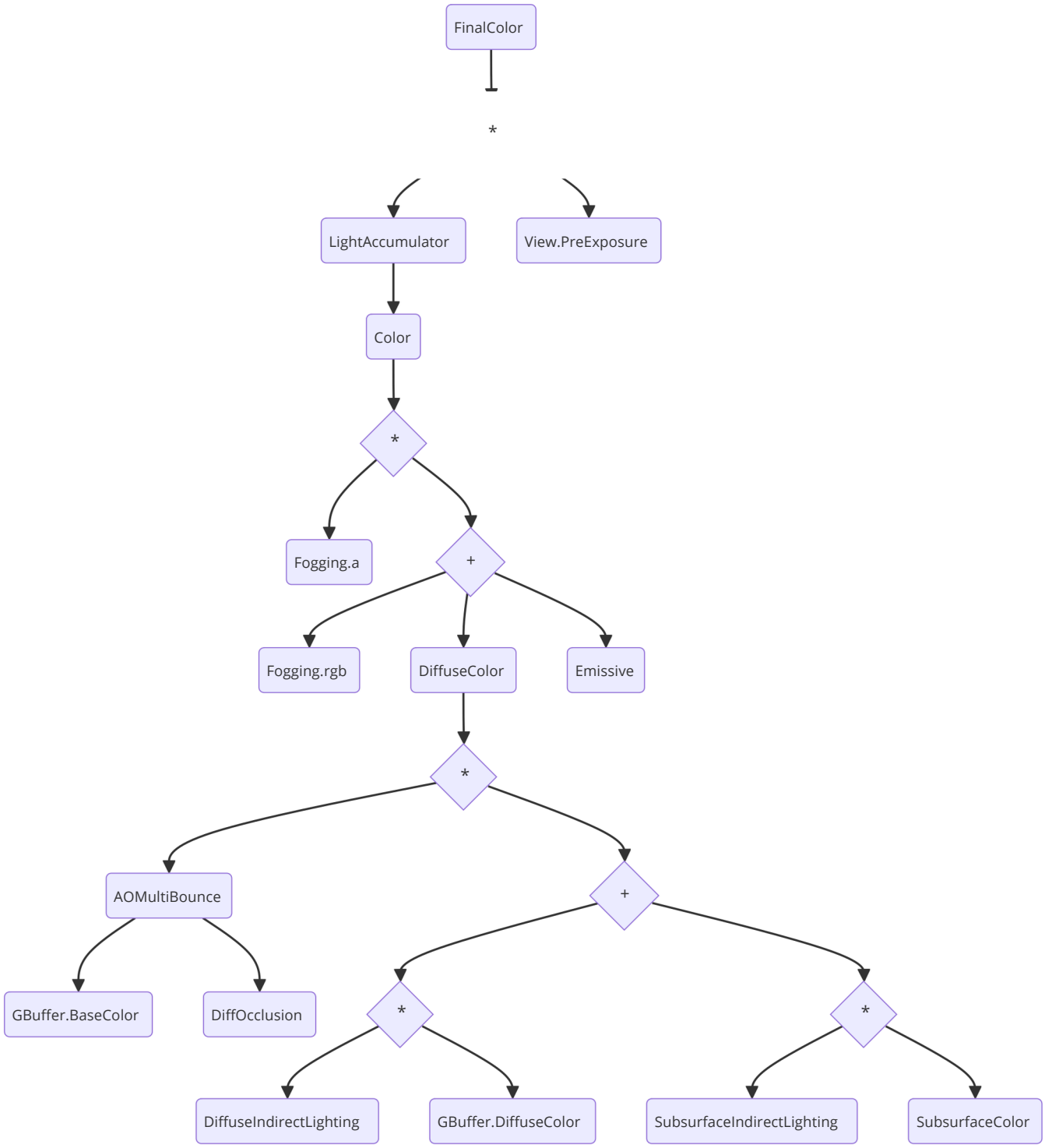
```
    const floatGBufferDither = InterleavedGradientNoise(MaterialParameters.SvPosition.xy,
    View_StateFrameIndexMod8);

SetGBufferForShadingModel(GBuffer,MaterialParameters,Opacity,BaseColor,Metallic,Specular,R
oughness,Anisotropy,SubsurfaceColor,SubsurfaceProfile,GBufferDither,ShadingModel);

    GBuffer.SelectiveOutputMask = GetSelectiveOutputMask();
    GBuffer.Velocity =0;
    GBuffer.SpecularColor = ComputeF0(Specular, BaseColor, Metallic); GBuffer.DiffuseColor
    = BaseColor - BaseColor * Metallic;

    GBuffer.DiffuseColor = GBuffer.DiffuseColor * View_DiffuseOverrideParameter.w +
    View_DiffuseOverrideParameter.xyz;
    GBuffer.SpecularColor = GBuffer.SpecularColor * View_SpecularOverrideParameter.w +
    View_SpecularOverrideParameter.xyz;

    if(View_RenderingReflectionCaptureMask) {

        EnvBRDFApproxFullyRough(GBuffer.DiffuseColor, GBuffer.SpecularColor);
    }

    float3 BentNormal = MaterialParameters.WorldNormal;

    if( GBuffer.ShadingModelID ==4&&0) {

        constfloat2 oct1 = ((float2(GBuffer.CustomData.a, GBuffer.CustomData.z) *2) - (256.0/255.0)) +
        UnitVectorToOctahedron(GBuffer.WorldNormal);
        BentNormal = OctahedronToUnitVector(oct1);
    }

    floatDiffOcclusion = MaterialAO; float
    SpecOcclusion = MaterialAO;

    ApplyBentNormal( MaterialParameters, GBuffer.Roughness, BentNormal, DiffOcclusion, SpecOcclusion );


    GBuffer.GBufferAO = AOMultiBounce( Luminance( GBuffer.SpecularColor ), SpecOcclusion
) . g;

    float3      DiffuseColor =0; Color
    float3      =0;
    floatIndirectIrradiance =0;

    float3      ColorSeparateSpecular       =  0;
    float3      ColorSeparateEmissive       =  0;

    float3 DiffuseDir = BentNormal;
    float3 DiffuseColorForIndirect = GBuffer.DiffuseColor; float3
            DiffuseIndirectLighting;
    float3      SubsurfaceIndirectLighting;

    GetPrecomputedIndirectLightingAndSkyLight(MaterialParameters, Interpolants, BasePassInterpolants,
LightmapVTPageTableResult, GBuffer, DiffuseDir, VolumetricLightmapBrickTextureUVs, DiffuseIndirectLighting,
SubsurfaceIndirectLighting, IndirectIrradiance);


    floatIndirectOcclusion =1.0f; float2
    NearestResolvedDepthScreenUV =0;
```

```
        floatDirectionalLightShadow =1.0f;
        DiffuseColor += (DiffuseIndirectLighting * DiffuseColorForIndirect + SubsurfaceIndirectLighting *
SubsurfaceColor) * AOMultiBounce( GBuffer.BaseColor, DiffOcclusion );

        float4 HeightFogging = float4(0,0,0,1);

        float4 Fogging = HeightFogging;
        float3 GBufferDiffuseColor = GBuffer.DiffuseColor; float3
        GBufferSpecularColor = GBuffer.SpecularColor;

        EnvBRDFApproxFullyRough(GBufferDiffuseColor, GBufferSpecularColor);

        Color = lerp(Color, GBufferDiffuseColor, View_UnlitViewmodeMask);

        float3 Emissive = GetMaterialEmissive(PixelMaterialInputs);

        if(View_OutOfBoundsMask >0) {

                if(any(abs(MaterialParameters.AbsoluteWorldPosition                    -
GetPrimitiveData(MaterialParameters.PrimitiveId).ObjectWorldPositionAndRadius.xyz)                    >
GetPrimitiveData(MaterialParameters.PrimitiveId).ObjectBounds +1))
                {
                        floatGradient = frac(dot(MaterialParameters.AbsoluteWorldPosition,
float3(.577f,.577f,.577f)) /500.0f);
                        Emissive = lerp(float3(1,1,0), float3(0,1,1), Gradient.xxx >.5f); Opacity =1;

                }
        }

        Color    + = DiffuseColor;
        Color    + = Emissive;

        {
                FLightAccumulator LightAccumulator = (FLightAccumulator)0;

                Color = Color * Fogging.a + Fogging.rgb;

                LightAccumulator_Add(LightAccumulator, Color,0,1.0f,false);

                Out.MRT[0] = (LightAccumulator_GetResult(LightAccumulator) ) ;
        }

        GBuffer.IndirectIrradiance = IndirectIrradiance;

        floatQuantizationBias = PseudoRandom( MaterialParameters.SvPosition.xy ) -0.5f;

        EncodeGBuffer(GBuffer, Out.MRT[1], Out.MRT[2], Out.MRT[3], OutGBufferD, OutGBufferE, OutGBufferF,
OutVelocity, QuantizationBias);

        Out.MRT[0||0?5:4] = OutGBufferD; Out.MRT[0||0?6:5] =
        OutGBufferE; Out.MRT[0].rgba *= View_PreExposure;

}
```

This way, there are fewer macro definitions. Isn't it much cleaner and more concise? From this, we can easily extract the final color output process under the DefaultLit shading model:

```
float3 Color =0;

GetPrecomputedIndirectLightingAndSkyLight(MaterialParameters, Interpolants, BasePassInterpolants,
LightmapVTPageTableResult, GBuffer, DiffuseDir, VolumetricLightmapBrickTextureUVs, DiffuseIndirectLighting,
SubsurfaceIndirectLighting, IndirectIrradiance);


DiffuseColor += (DiffuseIndirectLighting * DiffuseColorForIndirect + SubsurfaceIndirectLighting *
SubsurfaceColor) * AOMultiBounce( GBuffer.BaseColor, DiffOcclusion );


Color    + =DiffuseColor;
Color    + =Emissive;
Color = Color * Fogging.a + Fogging.rgb;

FLightAccumulator LightAccumulator = (FLightAccumulator)0;
LightAccumulator_Add(LightAccumulator, Color,0,1.0f,false); Out.MRT[0] =
(LightAccumulator_GetResult(LightAccumulator) ) ;

Out.MRT[0].rgba *= View_PreExposure;
```

If we break it down into a legend, it looks like this:

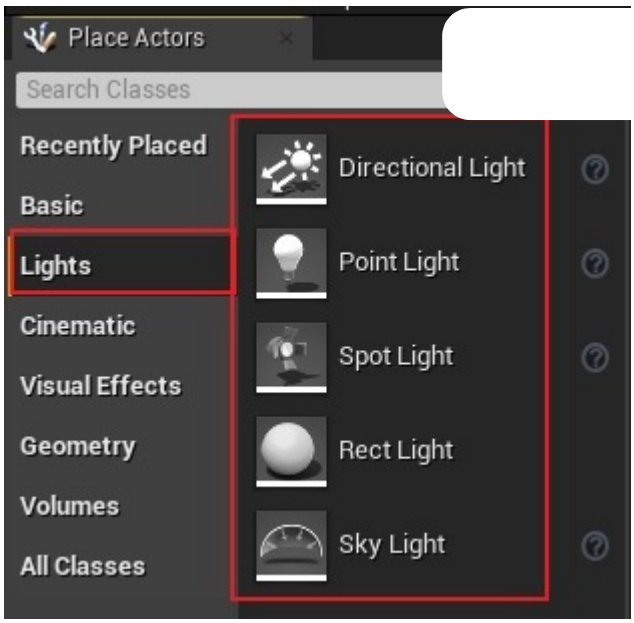


Therefore, under the default shading model (DefaultLit), the final output color of BasePass is affected by GBuffer.BaseColor, GBuffer.DiffuseColor, pre-calculated indirect light, sky light, subsurface scattering, AO, self-illumination, fog effect, pre -exposure, etc. (Ah!! It turns out that BasePass OutColor is a sea king, and it is ambiguous with so many color ladies at the same time O_O)

5.4 UE Light Source

5.4.1 Light source overview

UE's built-in light sources include Directional Light, Point Light, Spot Light, Rect Light, and Sky Light. (see the picture below)



The above light sources all have basic properties such as Transform and Mobility:

There are three modes of mobility: Static, Stationary, and Movable.

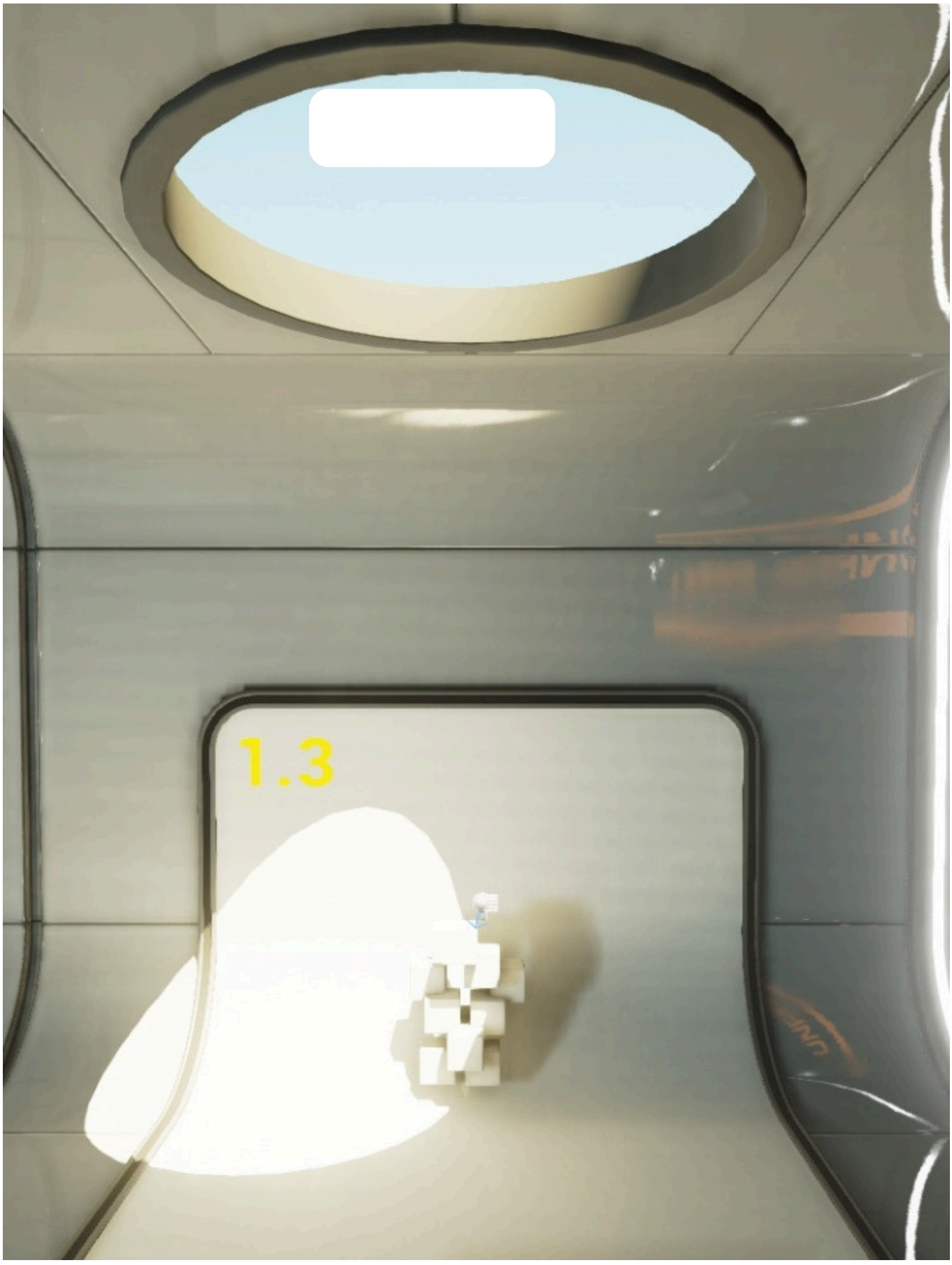
Static light sources can only be used as pre-calculated light sources, and Lightmap illumination maps are generated offline. Their properties cannot be changed at runtime. They have the lowest performance consumption, but the worst adjustability.

Stationary light sources bake shadows and indirect light offline to static geometry, so their shadows and indirect light cannot be changed at runtime, nor can their position be changed, but other properties can be changed. Moderate performance consumption and moderate adjustability.

Movable lights are fully dynamic lights and shadows, and any of their properties can be changed at runtime. They have the highest performance cost and the highest adjustability.

- **Directional Light**

Simulate light sources such as the sun, moon, and lasers that illuminate all objects in the scene from the same direction.



It has properties such as light intensity, color, color temperature, as well as property groups such as cascade shadows (CSM), light shaft, and light function.

- **Point Light**

Simulates a light source that spreads out in all directions from a single point, such as a light bulb.

In addition to basic properties, it also supports the setting of lighting functions.

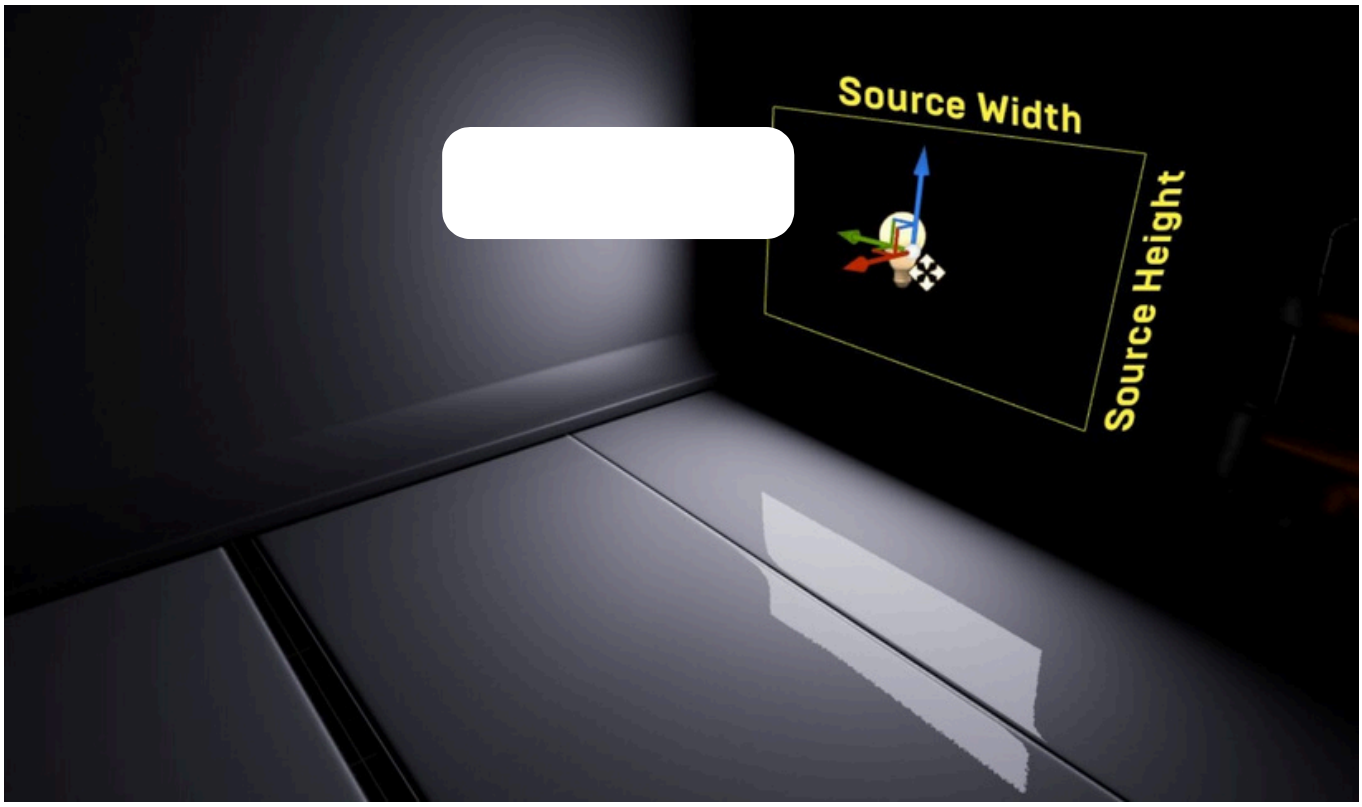
- **Spot Light**

A spotlight emits light in the shape of a cone, simulating a light source such as a flashlight.

It has special properties such as inner cone angle, outer cone angle, attenuation radius, and also supports lighting functions, IES, etc.

- **Rect Light**

Rectangular light simulates a surface light source with a rectangular area, such as LED, TV, monitor, etc.



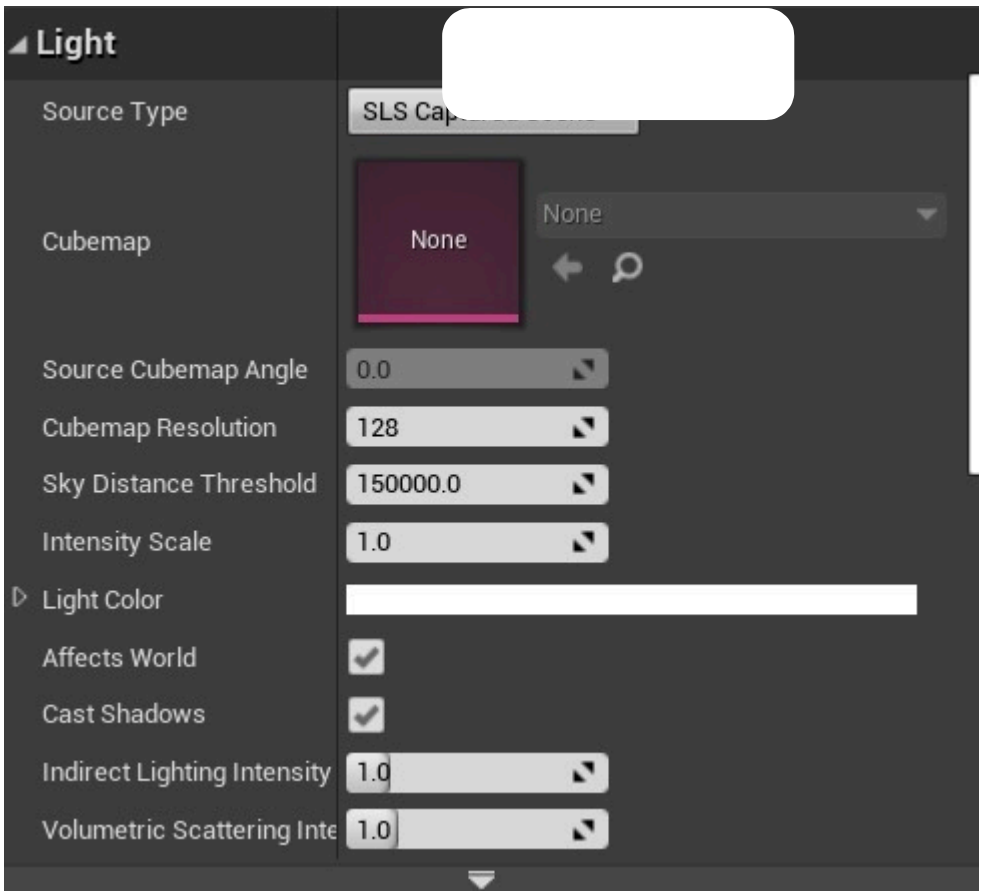
Its properties are a mixture of point light sources and spotlights, with special properties such as Source Width, Source Height, Barn Door Angle, Barn Door Length, as well as IES and lighting function property groups.

- **Sky Light**

Sky light, as its name suggests, simulates the lighting environment with sunlight or cloudy sky in the scene (Level). It can not only be used in outdoor environments, but also bring ambient light effects to indoor environments.

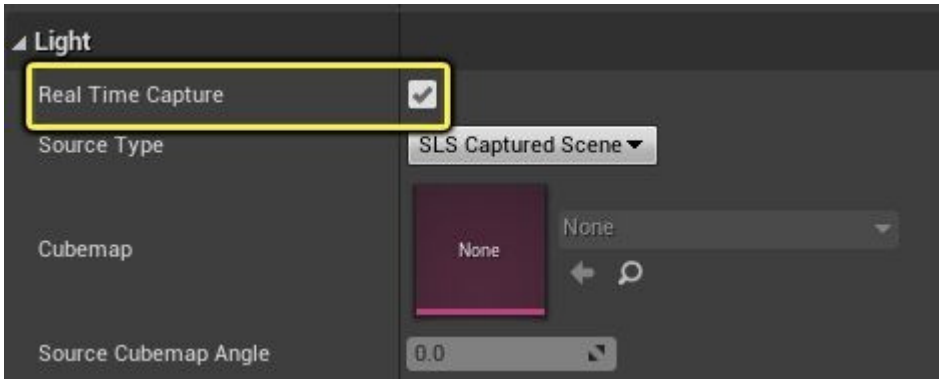
Simulates outdoor sky light with sun and clouds.

It also has three moving properties: Static, Fixed and Movable, and their meanings are similar to other types of light sources. Skylight has a Source Type property:



Source Type supports two modes:

- **Capture Scene**:Capture Cubemap from the scene. This mode can be subdivided into 3 behaviors based on mobility:
    - For Static Sky Lights, the Scene step is automatically triggered when the lighting for the scene is built.
    - For fixed or movable skylights, the scene is captured when the component is loaded. There is also a special case, when real-time capture() is turned on (UE4.25 does not have this feature yet), the capture scene interface can be called at runtime.



- **Specified Cubemap**:You can use an existing cubemap instead of capturing it from the scene.

5.4.2 Light source algorithm

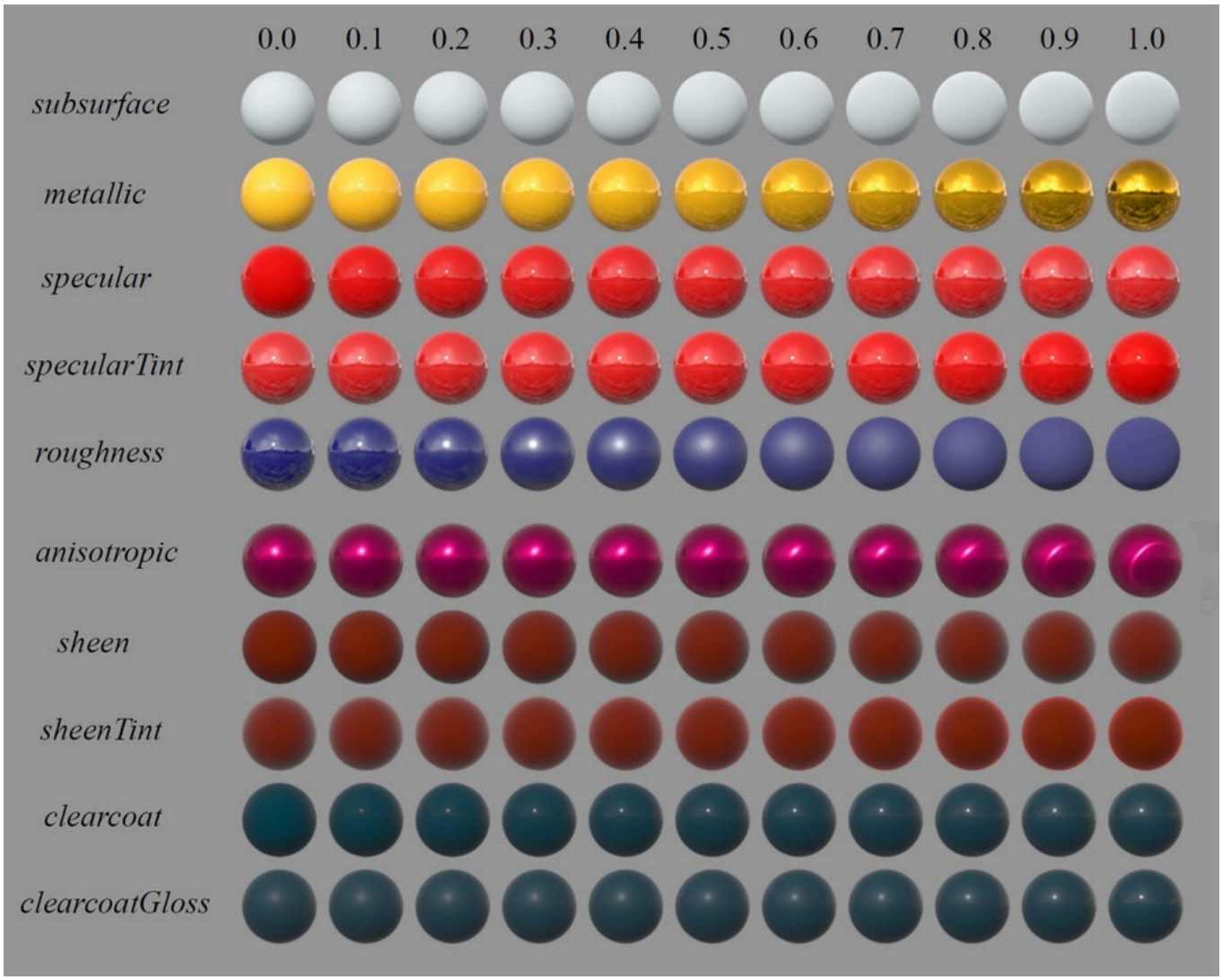
5.4.2.1 BRDF Overview

This section will analyze UE's lighting theory, model, and algorithm in combination with Real Shading in Unreal Engine 4 and Shader source code. For students who are weak in PBR lighting theory and knowledge, it is recommended that you read another article by the author to learn the principles and implementation of PBR from the shallower to the deeper .

In 1982,**Cook-Torrance**'s Specular BRDF was jointly proposed by Cook and Torrance. This model considers the relative brightness of different materials and different light sources in the same scene. It describes the distribution of reflected light in direction and the change of color when the reflection changes with the incident angle. It can also obtain the spectral energy distribution of light reflected from objects made of specific actual materials, and accurately reproduce the color based on this spectral energy distribution. Its formula is as follows:

$$fcthethek-ttherrancand=\frac{D(h)\cdot F(l,h)\cdot G(l,v,h)}{4(n\cdot l)(n\cdot v)}$$

In 2012,**Disney's BRDF**was proposed by Brent Burly of Disney Animation Studios, which greatly simplified the parameters and production process of the previously complex PBR with a small number of easy-to-understand parameters and a highly sophisticated art workflow. It is an art-oriented shading model, but not completely physically correct.

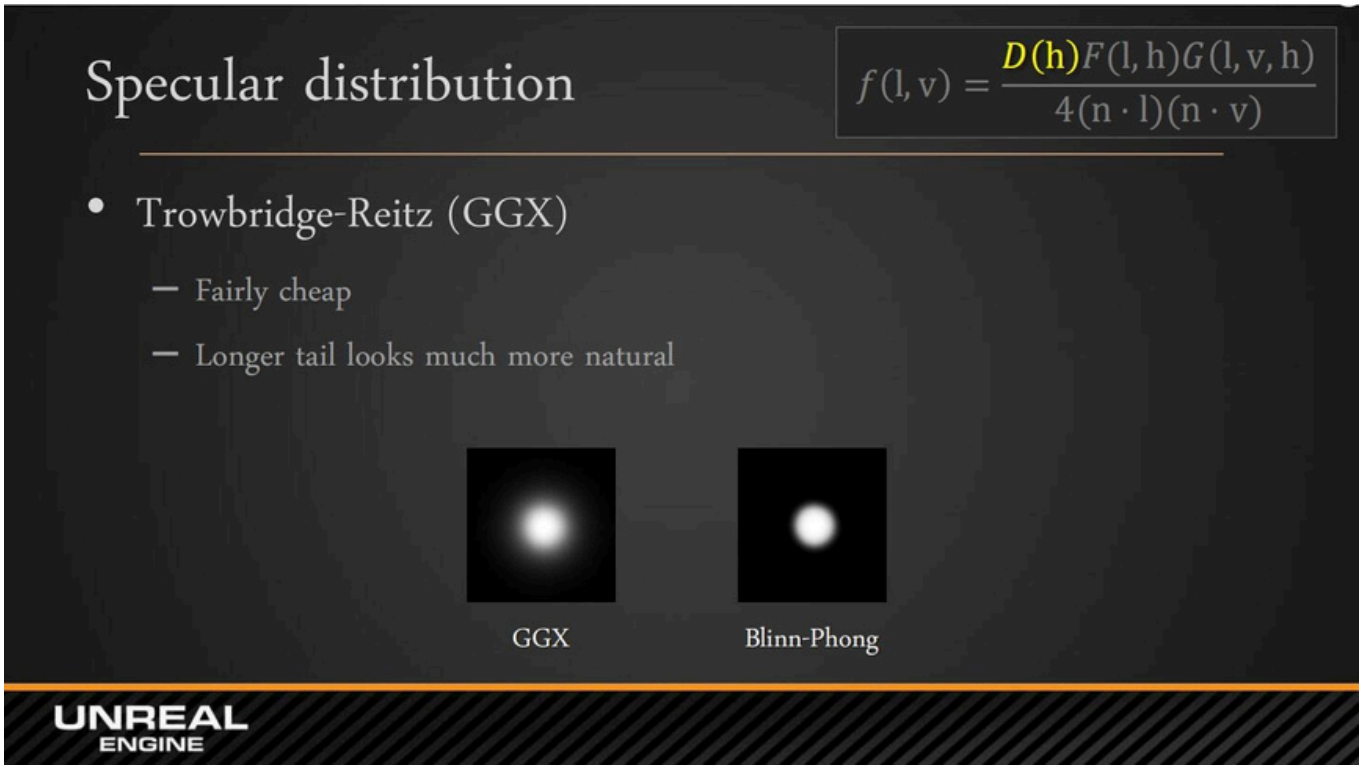


Disney principle BRDF abstracts various material parameters, which are mapped to values between 0 and 1.

The BRDF of the Disney principle laid the foundation for the direction and standards of PBR in the subsequent game and film industries. Subsequently, major commercial engines integrated the PBR rendering pipeline into their own engines based on the BRDF of the Disney principle. The following is the timeline for mainstream commercial engines to support PBR based on the Disney principle:

- **Unreal Engine 4**: "Real Shading in Unreal Engine 4", SIGGRAPH 2013 **Unity 5**:
- "Physically Based Shading in Unity", GDC 2014
- **Frostbite**: "Moving Frostbite to PBR", SIGGRAPH 2014 **Cry Engine 3.6**:
- "Physically Based Shading in Cry Engine", 2015

In 2013, the UE team led by Brian Karis was quick to take the lead and became the first to integrate Disney's principle BRDF into commercial engines. During the integration process, UE made tradeoffs, simplifications, improvements, and optimizations, and excellently integrated Disney's BRDF , which was originally used in the offline field, into UE's real-time rendering system.
UE made a multi-dimensional comparison on the selection of various sub-items of Cook-Torrance, and finally came up with the following model selection.
The first is the D term (normal distribution function). Compared with the Blinn-Phong model, the normal distribution function of GGX has a longer tail, is more in line with natural physical phenomena, and is quite efficient, and was eventually selected by UE:



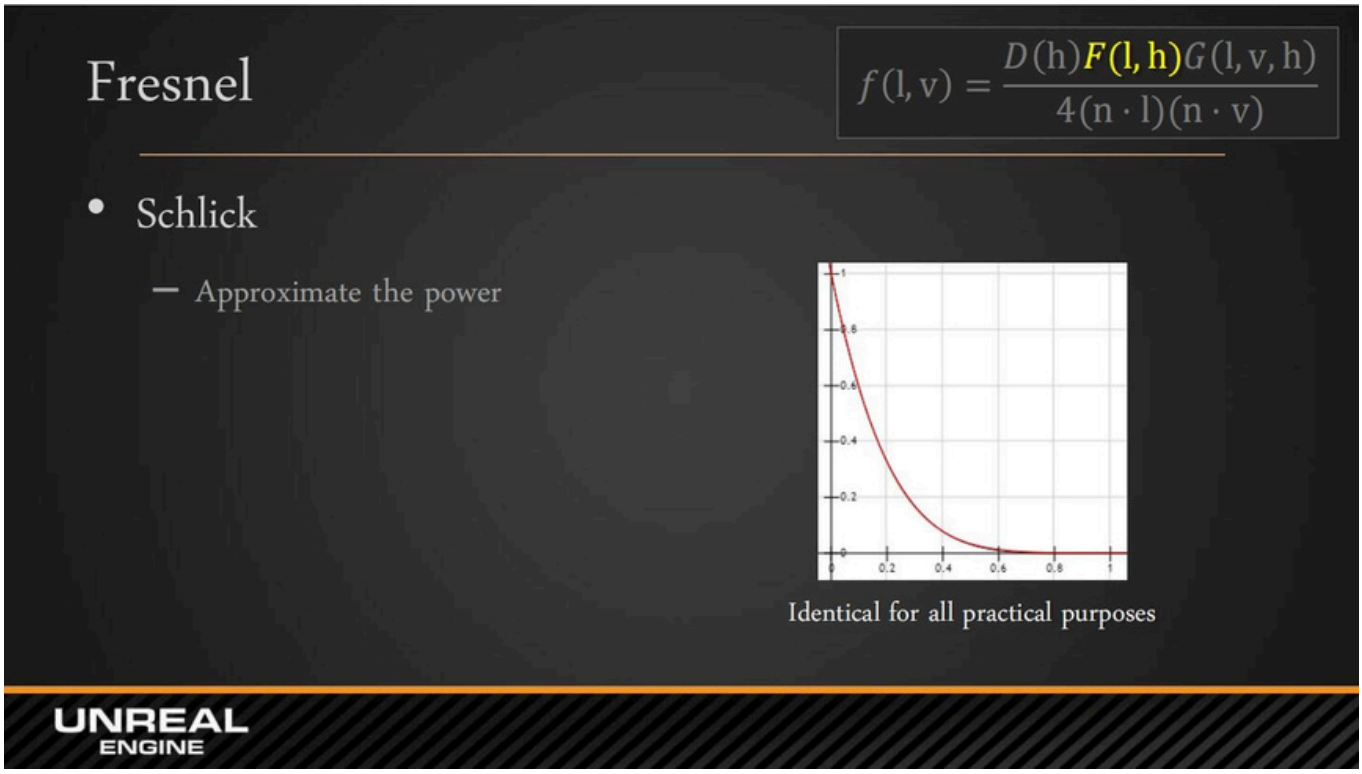
The formula of GGX normal distribution function is:

$$D_{GGX}(n,h,\alpha) = \frac{\alpha^2}{\pi((n\cdot h)^2(\alpha^2-1)+1)^2}$$

UE corresponding shader implementation code:

```
float D_GGX(floata2,floatNoH ) {

    floatd = (NoH * a2 - NoH ) * NoH +1; returna2 /          // 2 mad
    ( PI*d*d );                                              // 4 mul, 1 rcp
}
```

Next is the F term (Fresnel), which uses the Schlick approximation:



Schlick's Fresnel approximation classic formula:

$$F_{Schlick}(v, h) = F_0 + (1 - F_0)(1 - (h \cdot v))^5$$

The shader approximate implementation code corresponding to UE:

```
float3F_Schlick( float3 SpecularColor,floatVoH ) {

    floatFc = Pow5(1- VoH );                                 // 1 sub, 3 mul
    returnsaturate(50.0* SpecularColor.g ) * Fc + (1- Fc) * SpecularColor;
}
```

However, UE does not use the above formula completely, but uses the spherical Gaussian approximation method instead of the Pow operation for efficiency:

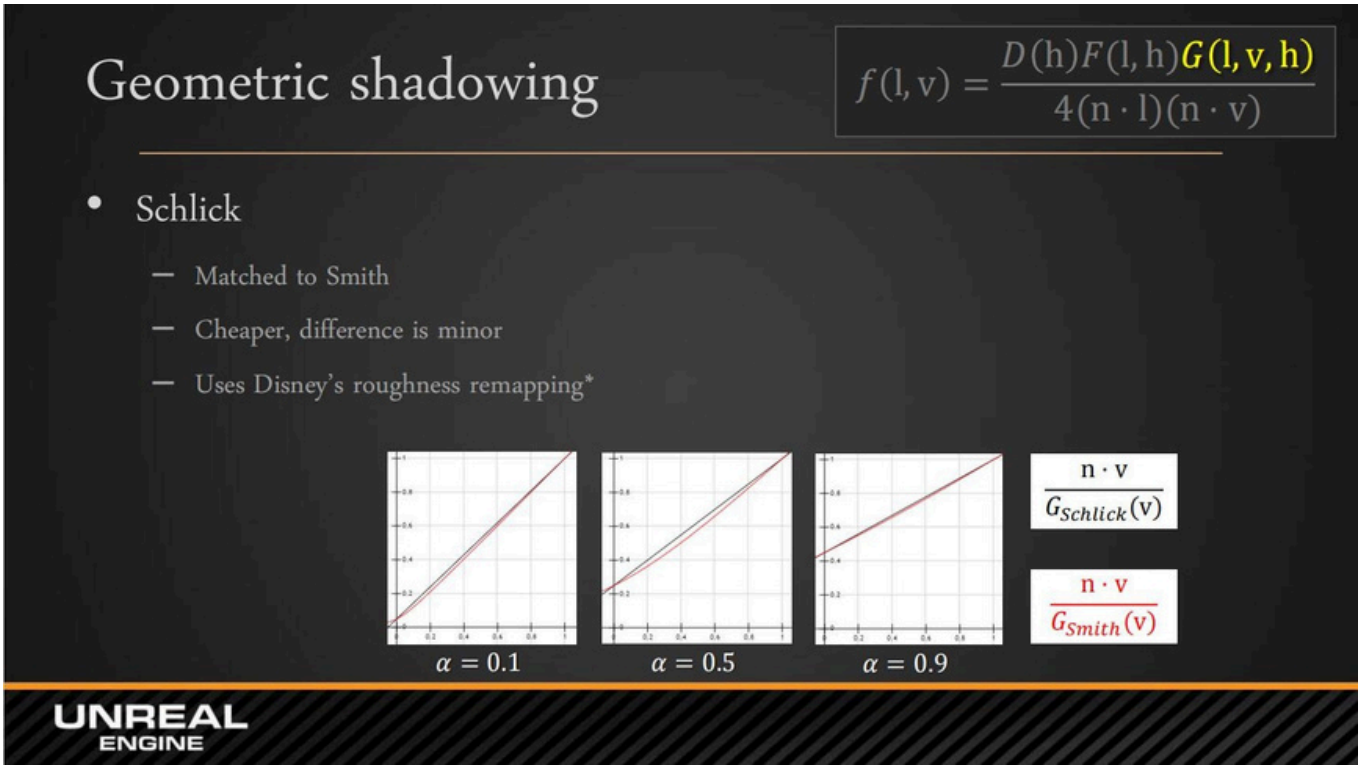
$$F(v,h) = F_0 + (1 - F_0)2^{(-5.55473 \cdot v - 6.98316) \cdot v \cdot h}$$

The implementation code corresponding to UE4.25 does not completely follow the above formula, and seems to have made a lot of approximations and optimizations, as follows:

```
float3F_Fresnel( float3 SpecularColor,floatVoH ) {
```

```
float3 SpecularColorSqrt =sqrt( clamp( float3(0,0,0), float3(0.99,0.99,0.99), SpecularColor ) );

float3 n = (1+ SpecularColorSqrt ) / (1- SpecularColorSqrt ); float3 g =sqrt( n*n + VoH*VoH
-1);
return0.5* Square( (g - VoH) / (g + VoH) ) * (1+ Square( ((g+VoH)*VoH -1) / ((g-VoH)*VoH +1) ) ); }
```

Finally, the G term (geometric masking) uses the Smith and Schlick joint masking function:



Their combined formula is as follows:

$$k = \frac{(\text{Roughness} + 1)2}{8}$$

$$G1(v) = \frac{n \cdot v}{(n \cdot v)(1-k) + k}$$

$$G(l,v,h) = G1(l)G1(v)$$

Corresponding UE implementation code:

```
float Vis_Schlick(floata2,floatNoV,floatNoL ) {

	floatk =sqrt(a2) *0.5;
	floatVis_SchlickV = NoV * (1- k) + k; floatVis_SchlickL = NoL * (1-
	k) + k; return0.25/ (Vis_SchlickV * Vis_SchlickL);

}
```

In addition to the above methods, UE also provides several other approximate methods for Cook-Torrance's lighting functions. For details, see section**5.2.3.6 BRDF.ush**.

UE's IBL approximates the illumination integral using the Riemann sum method. The illumination function combines Monte Carlo and importance sampling. The formula is as follows:

$$\int_H L_i(l) f(l,v) \cos_i dl \approx \frac{1}{N} \sum_{k=1}^{N} \frac{L_i(l_k) f(l_k,v) \cos_{i_{lk}}}{p(l_k,v)}$$

The corresponding shader implementation code:

```
float4ImportanceSampleGGX( float2 E,floata2 ) {

	floatPhi =2*PI*Ex;
	floatCosTheta =sqrt( (1- Ey) / (1+ (a2 -1) * Ey ) ); floatSinTheta =sqrt(1- CosTheta *
	CosTheta );

	float3 H;
	Hx = SinTheta *cos( Phi );
	Hy = SinTheta *sin( Phi );
	Hz = CosTheta;

	floatd = (CosTheta * a2 - CosTheta ) * CosTheta +1; floatD = a2 / ( PI*d*d );
	floatPDF = D * CosTheta;


	returnfloat4( H, PDF );
}

float3SpecularIBL( uint2 Random, float3 SpecularColor,floatRoughness, float3 N, float3 V
   )
{
	float3 SpecularLighting =0;

	constuint NumSamples =32;
	for( uint i =0; i < NumSamples; i++ ) {

		float2 E = Hammersley( i, NumSamples, Random );
		float3 H = TangentToWorld( ImportanceSampleGGX( E, Pow4(Roughness) ).xyz, N ); float3 L =2* dot( V, H )
		* H - V;

		floatNoV = saturate( dot( N, V ) ); floatNoL =
		saturate( dot( N, L ) ); floatNoH = saturate( dot( N,
		H ) ); floatVoH = saturate( dot( V, H ) );


		if( NoL >0)
		{
			float3 SampleColor = AmbientCubemap.SampleLevel( AmbientCubemapSampler, L,0
) . rgb;

			floatVis = Vis_SmithJointApprox( Pow4(Roughness), NoV, NoL ); floatFc =pow(1- VoH,
			5);
			float3 F = (1- Fc) * SpecularColor + Fc;

			// Incident light = SampleColor * NoL
			// Microfacet specular = D*G*F / (4*NoL*NoV) = D*Vis*F // pdf = D *
			NoH / (4 * VoH)
```

```
                SpecularLighting += SampleColor * F * ( NoL * Vis * (4* VoH / NoH) );
            }
        }

        returnSpecularLighting/NumSamples;
}
```

Later, the diffuse reflection term will be separated into two terms. This part will not be elaborated. Those who are interested can refer tothe principle and implementation of PBR from the shallow to the deep .

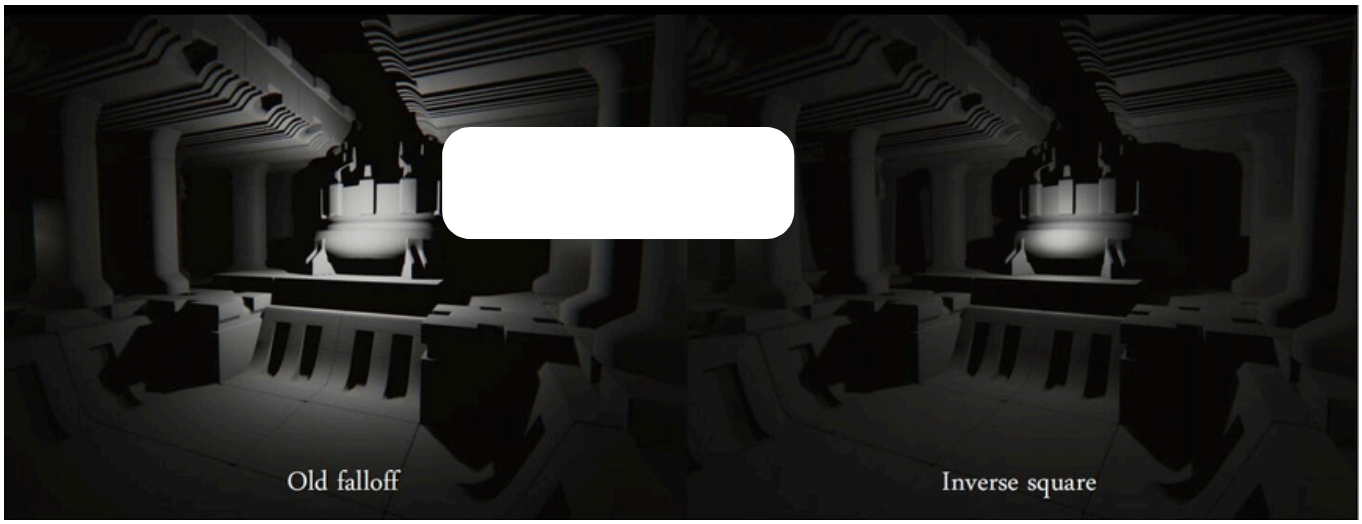
The following is an explanation of the UE's illumination model. For the distance attenuation function of the light source, the UE uses the following physical approximation:

$$\text{Fall-off} = \frac{\text{saturate}(1 - (\text{distance}/\text{lightRadius})^4)^2}{\text{distance}^2 + 1}$$

The corresponding implementation code:

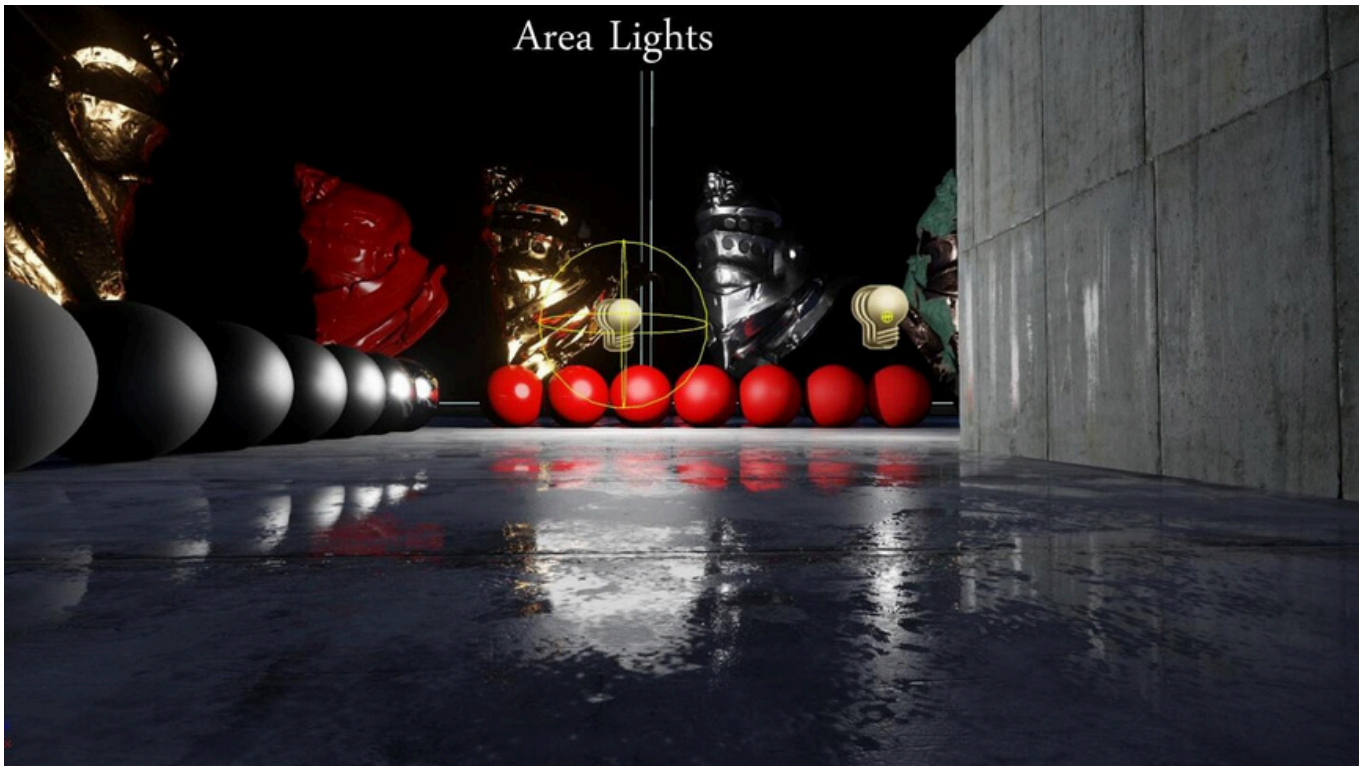
```
// Engine\Shaders\Private\DeferredLightingCommon.ush

float GetLocalLightAttenuation(float3 WorldPosition, FDeferredLightData LightData, inout float3 ToLight, inout float3 L)

{
        ToLight = LightData.Position - WorldPosition;

        floatDistanceSqr = dot( ToLight, ToLight ); L = ToLight *
        rsqrt( DistanceSqr );

        floatLightMask;
        if(LightData.bInverseSquared) {

            LightMask = Square( saturate(1- Square( DistanceSqr *
            Square(LightData.InvRadius) ) ) );
        }

        (.....)

        returnLightMask;
}
```

Compared to the old distance attenuation function, the new one is more physically realistic: it has a more reasonable radiation range and smaller contrast brightness changes:



5.4.2.2 Special light sources

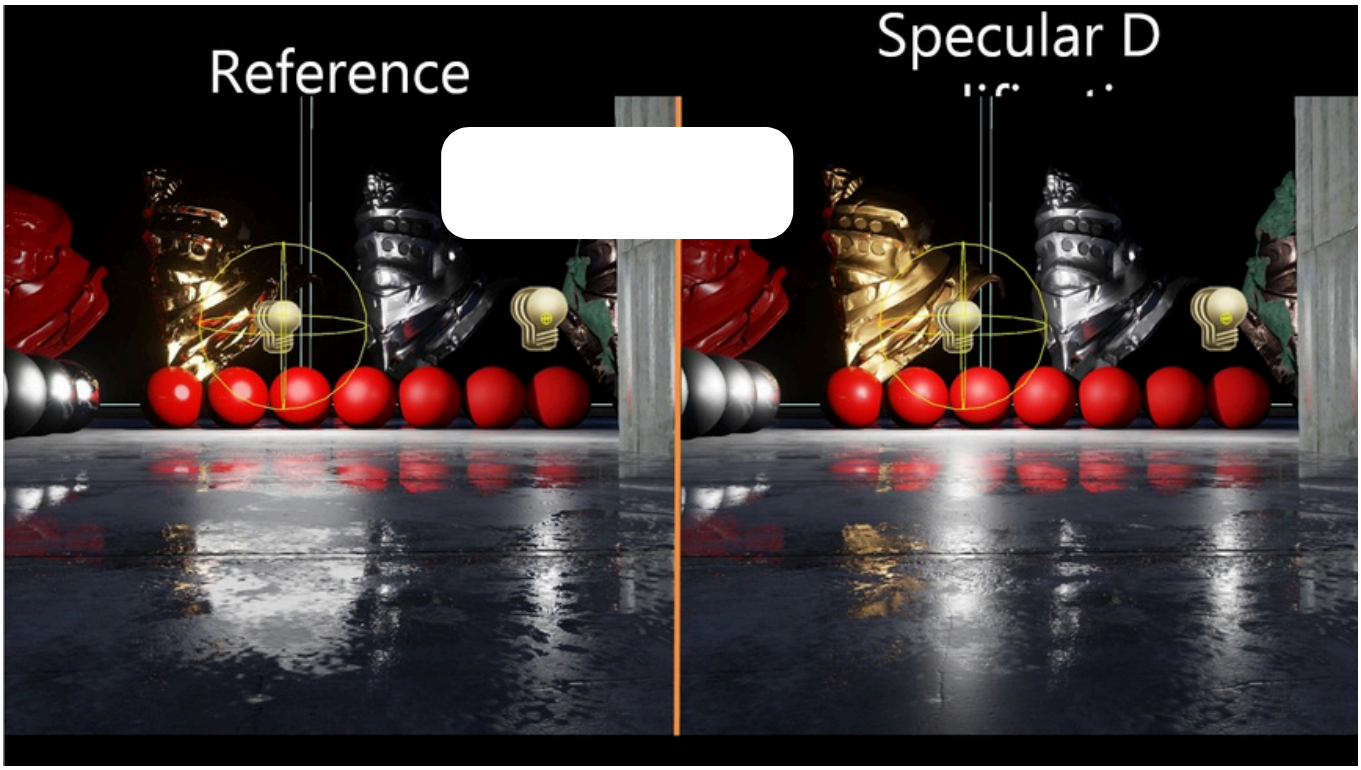
For**Area Light**,UE strives to achieve a consistent material appearance, and tries to keep the energy calculation of the diffuse and specular BRDFs as matched as possible. When the solid angle is close to 0, point light sources can be used to approximate area light sources, and finally ensure that they are efficient enough to be widely used.



UE has modified the D term of the highlight, based on a wider highlight distribution of the solid angle. The formula for changing the roughness is as follows:

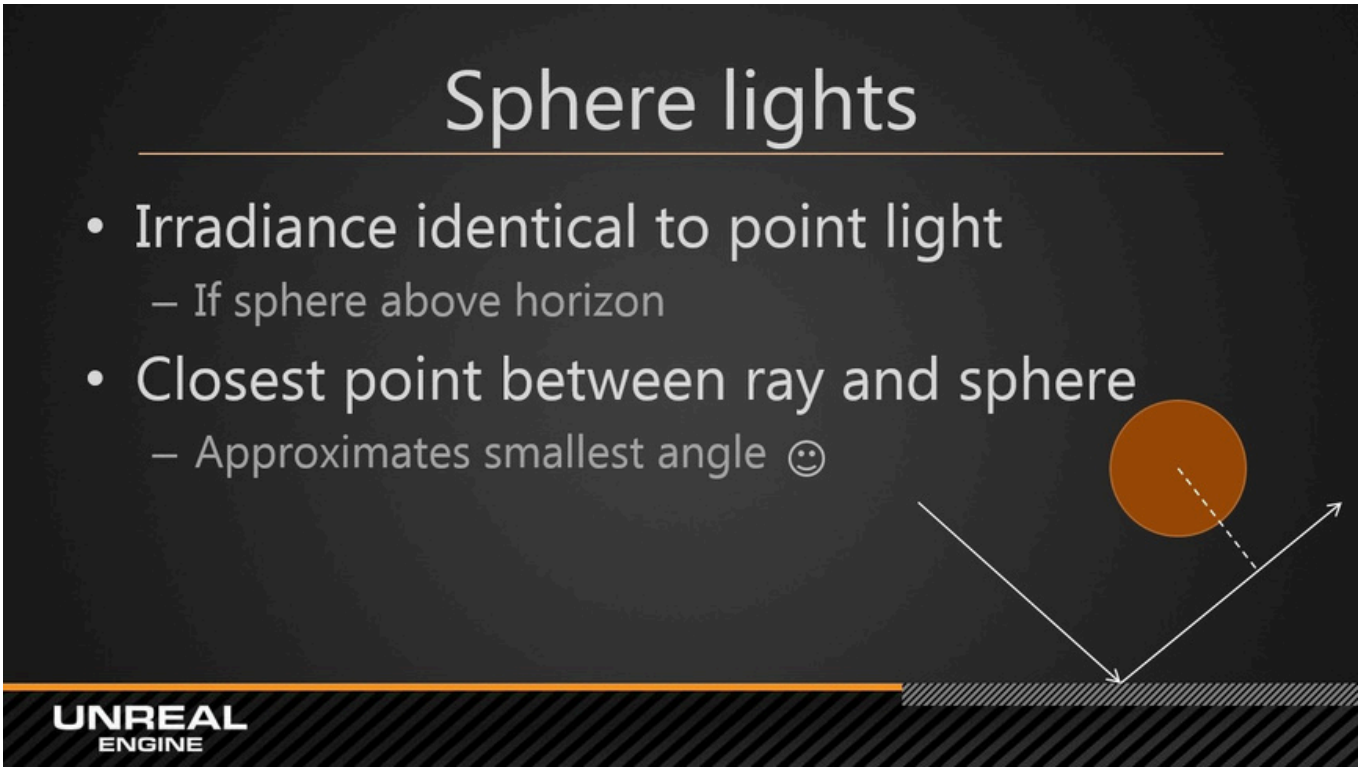
$$\alpha' = \text{saturate}(\alpha + \frac{\text{sourceRadius}}{2 * \text{distance}})$$

This also leads to a new problem: smooth objects no longer look smooth (below).



At the same time, a representative point in the area light is selected to simulate the entire area, which can be used directly for lighting and shading. It is a good choice for points with a large distribution range, and a sufficiently small angle can be used to approximate the reflected light.

For**a Sphere Light**,the irradiance is treated as a point light source (assuming the sphere is on the horizontal plane of the reflected light), and then the reflected light and the sphere are used to find the closest point:



The formula for calculating the minimum distance between the sphere and the reflected light is as follows:

$$\text{CenterToRay} = (L \cdot r)r - L$$

$$\text{ClosetPoint} = L + \text{CenterToRay} \cdot \text{saturate}\left(\frac{\text{SourceRadius}}{|\text{CenterToRay}|}\right)$$

$$l = ||\text{ClosetPoint}||$$

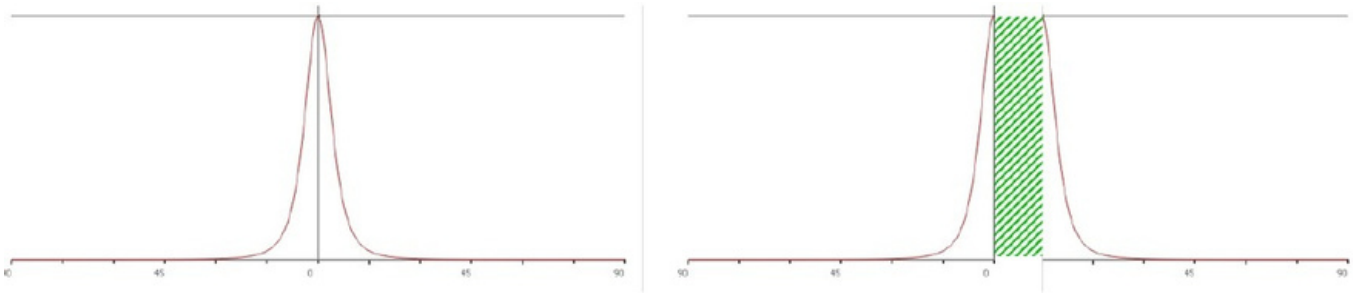
in L is the vector pointing from the shading point to the center of the sphere light source, SourceRadius is the radius of the light source sphere, r yes L The reflection vector at the shad point.

Then the energy integral of the shading point can be calculated according to the following formula:

$$Iptheint = \frac{p+2}{2p}\cos p\phi r$$

$$Isphandrand = \begin{cases} \frac{p+2}{2p} & if\ \phi r < \phi s \\ \frac{p+2}{2p}\cos p(\phi r - \phi s) & if\ \phi r \geq \phi s \end{cases}$$

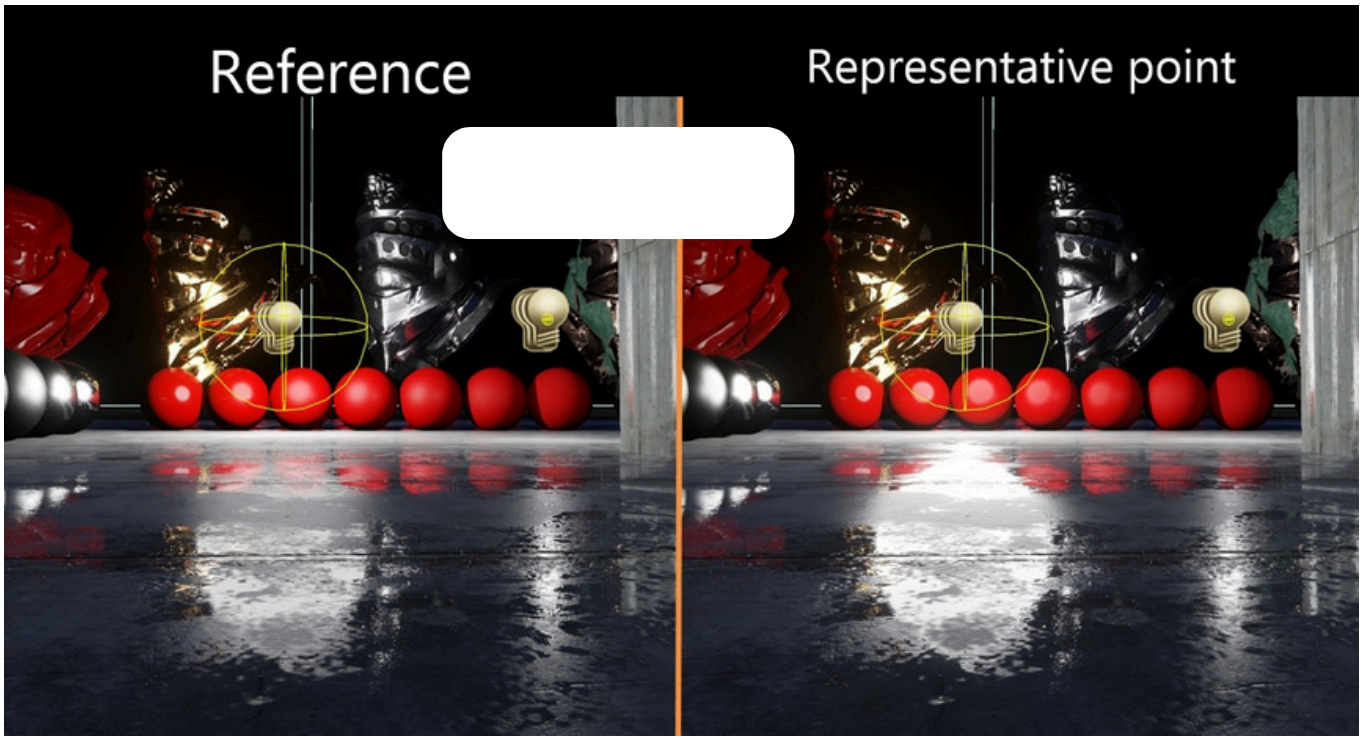
in ϕr ye r san L d The angle between ϕs is half the subtended angle of the sphere, $ptheint$ It is norm meaning that its integral over the hemisphere is 1. $Isphandrand$ is no longer normalized and dependent p the integral can become very large:



Since the normalization factor of GGX is $\frac{1}{\pi a_2}$ in order to obtain the approximate normalization o the representative points, the following formula is assumed:

$$\text{SphereNormalization} = \left(\frac{a}{a'}\right)^2$$

This results in a very nice spherical lighting effect:



The following is an explanation of the algorithm for**tube light (Tube Light, Capsule Light)**.For the vectors pointing from the shading point to the two ends of the tube light$L0$and$L1$,the analytical so irradiance integral of the line segment between them is:

$$\int_{L0}^{L1} \frac{n \cdot L}{|L|^3} dL = \frac{\frac{n \cdot L0}{L0} + \frac{n \cdot L1}{L1}}{|L0||L1| + (L0 \cdot L1)}$$

UE has made a lot of simplifications and approximations for this purpose, the most important of which is similar to a spherical light source, approximating the minimum angle and replacing it with the shortest distance:

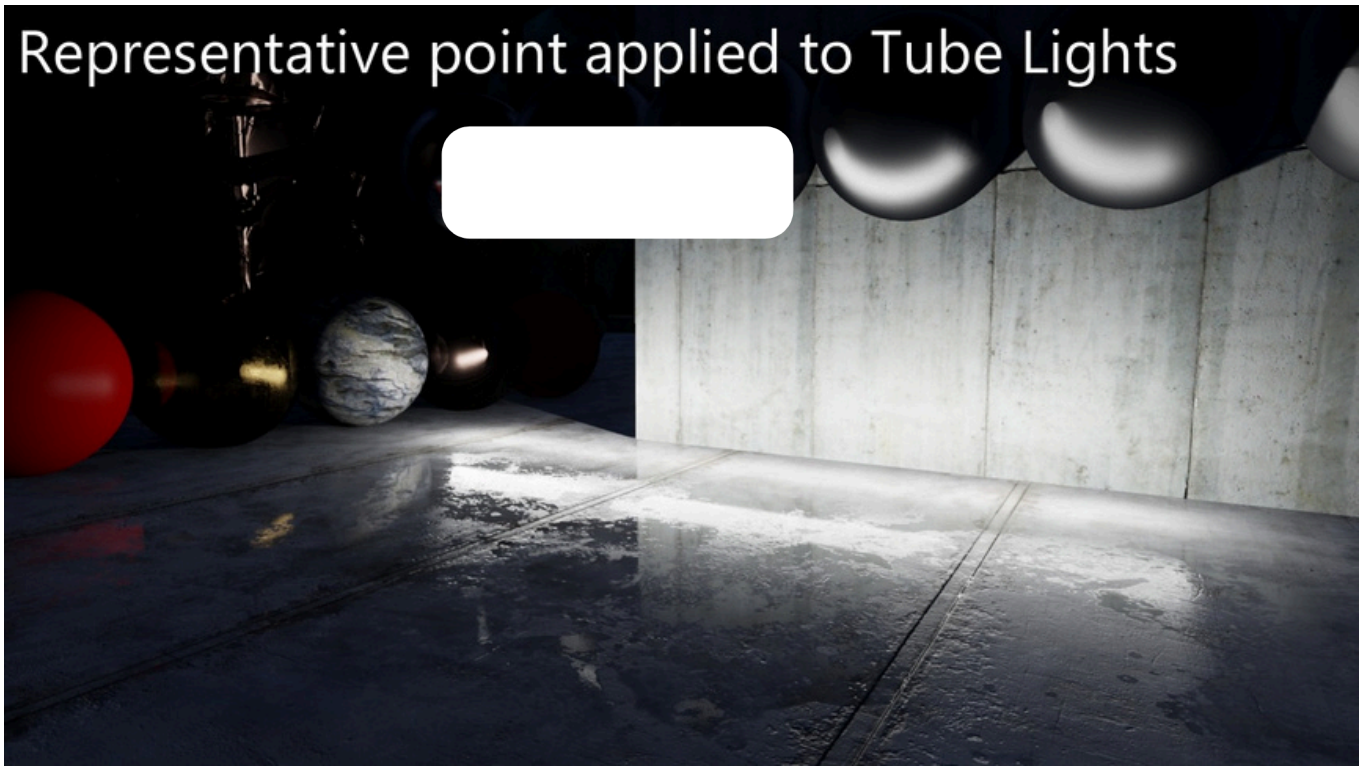
$$Ld = L1 - L0$$

$$t = \frac{(r \cdot L0)(r \cdot Ld) - (L0 \cdot Ld)}{|Ld|^2 - (r \cdot Ld)^2}$$

And the normalization factor according to the anisotropy GGX is $\frac{1}{pow(a_{xand}}$(Assumption$x$and$a$), thus we get the linear normalization formula:

$$\text{LineNormalizetion} = \frac{a}{a'}$$

Since only the light origin needs to be changed and energy conservation terms applied, these operations can be accumulated. So the approximate convolution for a tubular light source works very well:



5.4.3 Light source allocation and clipping

5.4.3.1 Basic concepts of light sources

The light source allocation of the UE is determined by the variables

FDeferredShadingSceneRenderer::RenderwithinbComputeLightGrid:

```
void FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList) {

	(……)

	boolbComputeLightGrid =false;
	if(!IsSimpleForwardShadingEnabled(ShaderPlatform)) {

		if(bUseGBuffer)
		{
			bComputeLightGrid	=	bRenderDeferredLighting;
		}
		else
		{
			bComputeLightGrid	=	ViewFamily.EngineShowFlags.Lighting;
		}

		bComputeLightGrid |= (
			ShouldRenderVolumetricFog()	||
			ViewFamily.ViewMode != VMI_Lit);
	}

	(……)
}
```

Whether to enable the light distribution task is affected by several factors: it is not a simple forward shading, and GBuffer's deferred rendering is used, or ViewFamily's lighting calculation is enabled, or volumetric fog needs to be rendered, etc. Then call the following logic to perform light distribution:

```
void FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList) {

    (……)

    //An ordered collection of light sources.
    FSortedLightSetSceneInfo {      SortedLightSet;

        //Collect and sort light sources.
        GatherAndSortLights(SortedLightSet); //Calculate the
        light grid (assign light sources).
        ComputeLightGrid(RHICmdList, bComputeLightGrid,              SortedLightSet);
    }

    (……)
}
```

FSortedLightSetSceneInfoIt is a very important basic concept of light source, and all subsequent light source processing will involve it. Let's take a look at its definition and other light source related concepts:

```
// Engine\Source\Runtime\Engine\Public\PrimitiveSceneProxy.h

//Data for simple dynamic light sources.
classFSimpleLightEntry {

public:
    FVector    Color;
    float    Radius;
    float    Exponent;
    float    VolumetricScatteringIntensity;
    bool    bAffectTranslucency;
};

//View-dependent data for simple dynamic lights.
classFSimpleLightPerViewEntry {

public:
    FVector Position;
};

//    Pointing to a simple light instanceviewRelated data.
//    The class name should be wrong.InstaceIt should beInstance.
class    FSimpleLightInstacePerViewIndexData
{
public:
    uint32 PerViewIndex :31;//occupy31Bit uint32
    bHasPerViewData;                    //occupy1Bit
};

//Simple dynamic light list.
```

```
classFSimpleLightArray {

public:
	//Simple dynamic light list, independent of view.
	TArray<FSimpleLightEntry, TMemStackAllocator<>> //Simple	InstanceData;
	dynamic light list, view dependent.
	TArray<FSimpleLightPerViewEntry, TMemStackAllocator<>> //Pointing to a PseimrVpielew liDgahtta ;
	instanceviewList of related data. TArray<FSimpleLightInstacePerViewIndexData,
	TMemStackAllocator<>> InstancePerViewDataIndices;



public:
	//Gets view-related data for a simple light source.
	inlineconstFSimpleLightPerViewEntry&GetViewDependentData(int32 LightIndex, int32 ViewIndex, int32
NumViews)const
	{
		if(InstanceData.Num() == PerViewData.Num()) {

			returnPerViewData[LightIndex];
		}
		else
		{
			//Computes the view index.
			FSimpleLightInstacePerViewIndexData PerViewIndex =
InstancePerViewDataIndices[LightIndex];
			constint32 PerViewDataIndex = PerViewIndex.PerViewIndex + (
PerViewIndex.bHasPerViewData ? ViewIndex:0);
			returnPerViewData[PerViewDataIndex];
		}
	}
};


// Engine\Source\Runtime\Renderer\Private\LightSceneInfo.h


//Information about the light source used for sorting.
structFSortedLightSceneInfo {

	union
	{
		//Similar toFMeshDrawCommandThe sort key	The order of the following data members controls the ordering importance of the lights.
		value, struct
		{
			//Light source type.
			uint32 LightType : LightType_NumBits; //Whether there
			is texture configuration
			uint32 bTextureProfile :1; //Whether to
			have a lighting function.
			uint32 bLightFunction :1; //Whether to
			calculate shadows.
			uint32 bShadowed :1; //Whether to
			use the lighting channel.
			uint32 bUsesLightingChannels:1; //Whether it is a non-
			simple light source.
			uint32 bIsNotSimpleLight :1; //Whether to
			not support delayed tile rendering.
			uint32 bTiledDeferredNotSupported:1; //Whether to not
			support delayed cluster rendering.
```

```
                uint32 bClusteredDeferredNotSupported:1; Fields;
            }
            //    Packed sort key values.
            int32    Packed;
        }  SortKey;

        //The corresponding light source scene information.
        constFLightSceneInfo*            LightSceneInfo;
        //Simple light index.
        int32 SimpleLightIndex;


        //Non-simple light constructor.
        explicitFSortedLightSceneInfo(constFLightSceneInfo* InLightSceneInfo)
            :LightSceneInfo(InLightSceneInfo),
            SimpleLightIndex(-1)
        {
            SortKey.Packed =0;
            SortKey.Fields.bIsNotSimpleLight =1;
        }

        //Simple light source constructor.
        explicitFSortedLightSceneInfo(int32                InSimpleLightIndex)
            :LightSceneInfo(nullptr), SimpleLightIndex
            (InSimpleLightIndex)
        {
            SortKey.Packed =0;
            SortKey.Fields.bIsNotSimpleLight =0;
        }
};

struct    FSortedLightSetSceneInfo
{
        //    Simple light end index.
        int    SimpleLightsEnd;
        //    The end index of the tile lighting.
        int    TiledSupportedEnd;
        //    Cluster lighting end index.
        int    ClusteredSupportedEnd;

        //    Index of the first light with a shadow map.
        int    AttenuationLightStart;

        //List of simple light sources.
        FSimpleLightArray SimpleLights; //An
        ordered list of non-simple lights.
        TArray<FSortedLightSceneInfo, SceneRenderingAllocator>                SortedLights;
};
```

What you can pay attention to isFSortedLightSceneInfothe light source sorting key value. The highest priority attributes are related to clustering, blocking, and simple light sources, followed by light channels, shadows, lighting functions, and lighting configurations. The lowest priority is the
light source type.

The basis for this sorting is: clustering and blocking can greatly reduce shading consumption and the number of instructions through operations such as early light source clipping, so they are placed at

the highest position; the lighting logic of simple light sources is much simpler, and the number of instructions is much less. All simple light sources can be rendered in clusters and blocks, so they are placed right after clustering and blocking; the rest are the characteristics of non-simple lighting.
Whether the light channel is turned on directly affects a lot of logic, turning on the light channel will greatly affect the rendering efficiency, so it is placed first after non-simple light sources; whether shadows are turned on also has a great impact on rendering performance. If shadows are turned on, several passes will be added to draw shadow maps and attenuation textures, and rendering texture switching will be increased, so the ranking is reasonable; the following key values are deduced by analogy.

5.4.3.2 GatherAndSortLights

This section analyzes the interface for collecting and sorting light sourcesGatherAndSortLights:

```
// Engine\Source\Runtime\Renderer\Private\LightRendering.cpp

void FDeferredShadingSceneRenderer::GatherAndSortLights(FSortedLightSetSceneInfo& OutSortedLights)

{
	if(bAllowSimpleLights) {

		//Collect simple light sources, detailed analysis is shown below.
		GatherSimpleLights(ViewFamily, Views, OutSortedLights.SimpleLights);
	}

	FSimpleLightArray &SimpleLights = OutSortedLights.SimpleLights;
	TArray<FSortedLightSceneInfo, SceneRenderingAllocator> &SortedLights =
	OutSortedLights.SortedLights;

	// SortedLightsContains simple light sources and non-simple	This will facilitate sorting later.
	light sources, SortedLights.Empty(Scene->Lights.Num()		+ SimpleLights.InstanceData.Num());

	boolbDynamicShadows = ViewFamily.EngineShowFlags.DynamicShadows && GetShadowQuality()
> 0;

	//	Build a list of visible light sources.
	for	(TSparseArray<FLightSceneInfoCompact>::TConstIterator		LightIt(Scene->Lights);
LightIt;	+ + LightIt)
	{
		constFLightSceneInfoCompact& LightSceneInfoCompact = *LightIt; const
		FLightSceneInfo*constLightSceneInfo = LightSceneInfoCompact.LightSceneInfo;


		if(LightSceneInfo->ShouldRenderLightViewIndependent()
			&& !ViewFamily.EngineShowFlags.ReflectionOverride)
		{
			//	Check if the light source is in a certainvieVisible inside.
			for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

				if(LightSceneInfo->ShouldRenderLight(Views[ViewIndex])) {

					//createFSortedLightSceneInfoExamples. FSortedLightSceneInfo*
					SortedLightInfo = new(SortedLights)
FSortedLightSceneInfo(LightSceneInfo);
```

```
                    //Detect shadow or lighting functions.
                    SortedLightInfo->SortKey.Fields.LightType                    =
LightSceneInfoCompact.LightType;
                    SortedLightInfo->SortKey.Fields.bTextureProfile                    =
ViewFamily.EngineShowFlags.TexturedLightProfiles && LightSceneInfo->Proxy-
> GetIESTextureResource();
                    SortedLightInfo->SortKey.Fields.bShadowed = bDynamicShadows &&
CheckForProjectedShadows(LightSceneInfo);
                    SortedLightInfo->SortKey.Fields.bLightFunction =
ViewFamily.EngineShowFlags.LightFunctions && CheckForLightFunction(LightSceneInfo);
                    SortedLightInfo->SortKey.Fields.bUsesLightingChannels =
Views[ViewIndex].bUsesLightingChannels && LightSceneInfo->Proxy->GetLightingChannelMask() !=
GetDefaultLightingChannelMask();

                    //Not a simple light source.
                    SortedLightInfo->SortKey.Fields.bIsNotSimpleLight =1;

                    //Specific conditions for supporting blocking or clustering: No additional features of the light source are used, and it is not a
                    parallel light or a rectangular light. const boolbTiledOrClusteredDeferredSupported =
                        !SortedLightInfo->SortKey.Fields.bTextureProfile !SortedLightInfo-&&
                        >SortKey.Fields.bShadowed && !SortedLightInfo-
                        >SortKey.Fields.bLightFunction !SortedLightInfo-                    &&
                        >SortKey.Fields.bUsesLightingChannels && LightSceneInfoCompact.LightType !=
                        LightType_ Directional && LightSceneInfoCompact.LightType != LightType_Rect;


                    //Whether tile rendering is not supported.
                    SortedLightInfo->SortKey.Fields.bTiledDeferredNotSupported                    = !
(bTiledOrClusteredDeferredSupported && LightSceneInfo->Proxy-
>IsTiledDeferredLightingSupported());

                    //Whether cluster rendering is not supported.
                    SortedLightInfo->SortKey.Fields.bClusteredDeferredNotSupported                    =
!bTiledOrClusteredDeferredSupported;
                    break;
                }
            }
        }
    }

    //Added simple light source.
    for(int32 SimpleLightIndex =0; SimpleLightIndex < SimpleLights.InstanceData.Num(); SimpleLightIndex++)

    {
        FSortedLightSceneInfo* SortedLightInfo = new(SortedLights)
        FSortedLightSceneInfo(SimpleLightIndex);
        SortedLightInfo->SortKey.Fields.LightType = LightType_Point; SortedLightInfo-
        >SortKey.Fields.bTextureProfile                    = 0;
        SortedLightInfo->SortKey.Fields.bShadowed =0; SortedLightInfo-
        >SortKey.Fields.bLightFunction SortedLightInfo-                    =  0;
        >SortKey.Fields.bUsesLightingChannels                    =0;

        //Simple light source.
        SortedLightInfo->SortKey.Fields.bIsNotSimpleLight =0;

        //Simple lights can be rendered in blocks or clusters.
        SortedLightInfo->SortKey.Fields.bTiledDeferredNotSupported =0;
```

```
            SortedLightInfo->SortKey.Fields.bClusteredDeferredNotSupported =0;
        }

        //Light source sorting, light sources without shadows and lighting functions are prioritized to avoid
        rendering texture switching. structFCompareFSortedLightSceneInfo {

            FORCEINLINEbool operator()(constFSortedLightSceneInfo& A,const
FSortedLightSceneInfo& B )const
            {
                returnA.SortKey.Packed < B.SortKey.Packed;
            }
        };
        SortedLights.Sort( FCompareFSortedLightSceneInfo() );

        //Initialize the index.
        OutSortedLights.SimpleLightsEnd =           SortedLights.Num();
        OutSortedLights.TiledSupportedEnd           = SortedLights.Num();
        OutSortedLights.ClusteredSupportedEnd           = SortedLights.Num();
        OutSortedLights.AttenuationLightStart           = SortedLights.Num();

        //Traverse all light sources to be rendered, and construct the range of clustered, block, and shadow-free light sources.
        for(int32 LightIndex =0; LightIndex < SortedLights.Num(); LightIndex++) {

            constFSortedLightSceneInfo& SortedLightInfo = SortedLights[LightIndex]; const bool
            bDrawShadows = SortedLightInfo.SortKey.Fields.bShadowed;
            const boolbDrawLightFunction = SortedLightInfo.SortKey.Fields.bLightFunction; const bool
            bTextureLightProfile = SortedLightInfo.SortKey.Fields.bTextureProfile; const boolbLightingChannels =
            SortedLightInfo.SortKey.Fields.bUsesLightingChannels;



            if(SortedLightInfo.SortKey.Fields.bIsNotSimpleLight &&
OutSortedLights.SimpleLightsEnd == SortedLights.Num())
            {
                //The ending index of the simple light.
                OutSortedLights.SimpleLightsEnd = LightIndex;
            }

            if(SortedLightInfo.SortKey.Fields.bTiledDeferredNotSupported &&
OutSortedLights.TiledSupportedEnd == SortedLights.Num())
            {
                //End index of the block light.
                OutSortedLights.TiledSupportedEnd = LightIndex;
            }

            if(SortedLightInfo.SortKey.Fields.bClusteredDeferredNotSupported &&
OutSortedLights.ClusteredSupportedEnd == SortedLights.Num())
            {
                //End index of the clustered light source.
                OutSortedLights.ClusteredSupportedEnd = LightIndex;
            }

            if(bDrawShadows || bDrawLightFunction || bLightingChannels) {

                //Index of the first shadowed light.
                OutSortedLights.AttenuationLightStart = LightIndex; break;

            }
```

```
    }
}
```

As can be seen from the above code, simple light sources can be rendered in blocks or clusters, but for non-simple light sources, only light sources that meet the following conditions can be rendered in blocks or clusters:

- Additional properties of the light source (TextureProfile, LightFunction, LightingChannel) are not used.
- Shadows are not enabled. Non-
- parallel or rectangular light.

In addition, whether block rendering is supported requires the light source scene proxy to IsTiledDeferredLightingSupportedreturn true:

```
// Engine\Source\Runtime\Engine\Public\SceneManagement.h

bool FLightSceneProxy::IsTiledDeferredLightingSupported()const {

    returnbTiledDeferredLightingSupported;
}
```

In the entire UE project source code, only the following sentence can bTiledDeferredLightingSupportedbe true:

```
// GEngine\Source\Runtime\Engine\Private\PointLightSceneProxy.h

classFPointLightSceneProxy: public FLocalLightSceneProxy {

    FPointLightSceneProxyconstUPointLightComponent* Component) ,
    SourceLength(Component->SourceLength)
    (.....)
    {
        //Whether to support tile rendering.
        bTiledDeferredLightingSupported = (SourceLength ==0.0f);
    }
}
```

In other words, only point lights with a length of 0 support block rendering, while other types and situations of light sources do not support it.

> Point light sources with a light source length greater than 0 become spherical lighting models, which are no longer applicable to simple lighting models and cannot be rendered in blocks.

GatherSimpleLightsThe purpose of is to collect simple light sources from the visible primitives in the scene:

```
void FSceneRenderer::GatherSimpleLights(constFSceneViewFamily& ViewFamily,const TArray<FViewInfo>&
Views, FSimpleLightArray& SimpleLights) {
```

```
    TArray<constFPrimitiveSceneInfo*, SceneRenderingAllocator>
    PrimitivesWithSimpleLights;

    //Gather visible primitives from all views where simple light sources may exist.
    for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        constFViewInfo& View = Views[ViewIndex]; for(int32 PrimitiveIndex =0;
        PrimitiveIndex < View.VisibleDynamicPrimitivesWithSimpleLights.Num();
PrimitiveIndex++)
        {
            constFPrimitiveSceneInfo* PrimitiveSceneInfo =
View.VisibleDynamicPrimitivesWithSimpleLights[PrimitiveIndex];

            //Note that adding elements usesAddUnique,This prevents multiple copies of the same light
            source from appearing in the list. PrimitivesWithSimpleLights.AddUnique(PrimitiveSceneInfo);
        }
    }

    //Collect simple light sources from primitives.
    for(int32 PrimitiveIndex =0; PrimitiveIndex < PrimitivesWithSimpleLights.Num(); PrimitiveIndex++)

    {
        constFPrimitiveSceneInfo* Primitive = PrimitivesWithSimpleLights[PrimitiveIndex]; Primitive->Proxy-
        >GatherSimpleLights(ViewFamily, SimpleLights);
    }
}
```

When collecting simple light sources, the interface of each primitive scene representative will be called GatherSimpleLights.Currently, the primitive scene representatives that implement this interface are mainly:

- FNiagaraSceneProxy
- FNiagaraRendererLights
- FDynamicSpriteEmitterData
- FParticleSystemSceneProxy

They are all agents related to Cascade Particles and Niagara Particles, so it can be inferred that simple light sources are only used in particle effects modules.

5.4.3.3 ComputeLightGrid

The role of ComputeLightGrid is to crop local light sources and reflection probes into a 3D grid in frustum space, and build a list of light sources and grids related to each view.

```
// Engine\Source\Runtime\Renderer\Private\LightGridInjection.cpp

void FDeferredShadingSceneRenderer::ComputeLightGrid(FRHICommandListImmediate& RHICmdList, bool
bNeedLightGrid, FSortedLightSetSceneInfo &SortedLightSet) {

    //No light source grid is required or supportedSM5And above, return directly.
    if(!bNeedLightGrid || FeatureLevel < ERHIFeatureLevel::SM5) {

        for(auto& View : Views) {
```

```
                View.ForwardLightingResources = GetMinimalDummyForwardLightingResources();
            }
            return;
        }

        {
            SCOPED_DRAW_EVENT(RHICmdList, ComputeLightGrid);

            static constautoAllowStaticLightingVar = IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT(
"r.AllowStaticLighting"));
            const boolbAllowStaticLighting = (!AllowStaticLightingVar ||
AllowStaticLightingVar->GetValueOnRenderThread() !=0);
            const boolbAllowFormatConversion =
            RHISupportsBufferLoadTypeConversion(GMaxRHIShaderPlatform);

            //Is thereviewUse forward rendering.
            boolbAnyViewUsesForwardLighting =false;
            for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                constFViewInfo& View = Views[ViewIndex]; bAnyViewUsesForwardLighting |=
                View.bTranslucentSurfaceLighting ||
ShouldRenderVolumetricFog() || View.bHasSingleLayerWaterMaterial;
            }

            //Whether to clip the light source to the grid.
            const boolbCullLightsToGrid = GLightCullingQuality
                && (ViewFamily.EngineShowFlags.DirectLighting
                && (IsForwardShadingEnabled(ShaderPlatform) || bAnyViewUsesForwardLighting ||
IsRayTracingEnabled() || ShouldUseClusteredDeferredShading()));

            // Store this flag if lights are injected in the grids, check with
            'AreClusteredLightsInLightGrid()'
            bClusteredShadingLightsInLightGrid = bCullLightsToGrid;

            for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                FViewInfo& View = Views[ViewIndex];
                FForwardLightData& ForwardLightData = View.ForwardLightingResources-
> ForwardLightData;
                ForwardLightData = FForwardLightData();

                TArray<FForwardLocalLightData, SceneRenderingAllocator> ForwardLocalLightData;

                floatFurthestLight=1000;

                // Track the end markers for different types int32
                SimpleLightsEnd =0; int32 ClusteredSupportedEnd =0;


                //  Clip light sources.
                if  (bCullLightsToGrid)
                {
                    SimpleLightsEnd = SortedLightSet.SimpleLightsEnd; // 1.Insert a
                    simple light source.
                    if(SimpleLightsEnd >0) {

                        constFSimpleLightArray &SimpleLights = SortedLightSet.SimpleLights;
```

```
                const FFloat16 SimpleLightSourceLength16f = FFloat16(0); FLightingChannels
                SimpleLightLightingChannels;
                //Apply simple lighting to all channels.
                SimpleLightLightingChannels.bChannel0 =
SimpleLightLightingChannels.bChannel1 = SimpleLightLightingChannels.bChannel2 =true;
                const uint32 SimpleLightLightingChannelMask =
GetLightingChannelMaskForStruct(SimpleLightLightingChannels);

                //Use an ordered list of lights, keeping track of their ranges.
                for(int SortedIndex =0; SortedIndex <
SortedLightSet.SimpleLightsEnd; ++SortedIndex)
                {
                    int32 SimpleLightIndex =
SortedLightSet.SortedLights[SortedIndex].SimpleLightIndex;

                    ForwardLocalLightData.AddUninitialized(1); FForwardLocalLightData& LightData =
                    ForwardLocalLightData.Last();

                    const FSimpleLightEntry& SimpleLight =
SimpleLights.InstanceData[SimpleLightIndex];
                    const FSimpleLightPerViewEntry& SimpleLightPerViewData =
SimpleLights.GetViewDependentData(SimpleLightIndex, ViewIndex, Views.Num());
                    LightData.LightPositionAndInvRadius =
FVector4(SimpleLightPerViewData.Position,1.0f/ FMath::Max(SimpleLight.Radius, KINDA_SMALL_NUMBER));

                    LightData.LightColorAndFalloffExponent =
FVector4(SimpleLight.Color, SimpleLight.Exponent);

                    //Simple light sources have no shadows.
                    uint32 ShadowMapChannelMask =0;
                    ShadowMapChannelMask |= SimpleLightLightingChannelMask <<8;

                    LightData.LightDirectionAndShadowMapChannelMask =
FVector4(FVector(1,0,0), *((float*)&ShadowMapChannelMask));

                    const FFloat16 VolumetricScatteringIntensity16f =
FFloat16(SimpleLight.VolumetricScatteringIntensity);
                    const uint32 PackedWInt =
((uint32)SimpleLightSourceLength16f.Encoded) |
((uint32)VolumetricScatteringIntensity16f.Encoded <<16);

                    LightData.SpotAnglesAndSourceRadiusPacked = FVector4(-2,1,0, *
(float*)&PackedWInt);
                    LightData.LightTangentAndSoftSourceRadius = FVector4(1.0f,0.0f,
0.0f,0.0f);
                    LightData.RectBarnDoor = FVector4(0,-2,0,0);
                }
            }

            const TArray<FSortedLightSceneInfo, SceneRenderingAllocator>& SortedLights
=  SortedLightSet.SortedLights;
            ClusteredSupportedEnd = SimpleLightsEnd; // 2.Added other types of light
            sources, and tracked the end index of clustered rendering support.
            for(int SortedIndex = SimpleLightsEnd; SortedIndex < SortedLights.Num();
+ + SortedIndex)
            {
                const FSortedLightSceneInfo& SortedLightInfo =
SortedLights[SortedIndex];
```

```
                    constFLightSceneInfo*constLightSceneInfo =
SortedLightInfo.LightSceneInfo;
                    constFLightSceneProxy* LightProxy = LightSceneInfo->Proxy;

                    if(LightSceneInfo->ShouldRenderLight(View)
                        && !ViewFamily.EngineShowFlags.ReflectionOverride)
                    {
                        FLightShaderParameters LightParameters; LightProxy-
                        >GetLightShaderParameters(LightParameters);

                        if(LightProxy->IsInverseSquared()) {

                            LightParameters.FalloffExponent =0;
                        }

                        if(View.bIsReflectionCapture) {

                            LightParameters.Color *= LightProxy-
> GetIndirectLightingScale();
                        }

                        int32 ShadowMapChannel = LightProxy->GetShadowMapChannel(); int32
                        DynamicShadowMapChannel = LightSceneInfo-
> GetDynamicShadowMapChannel();

                        if(!bAllowStaticLighting) {

                            ShadowMapChannel = INDEX_NONE;
                        }

                        //Static shadow usageShadowMapChannel,Dynamic shadows are packed into the light attenuation texture
DynamicShadowMapChannel.
                        uint32 LightTypeAndShadowMapChannelMaskPacked =
                            (ShadowMapChannel  ==0?1:0)  |  (ShadowMapChannel
                            ==1?2:0)      |      (ShadowMapChannel      ==2?4:0)      |
                            (ShadowMapChannel                  ==3?8:0)                |
                            (DynamicShadowMapChannel           ==0?16:0)           |
                            (DynamicShadowMapChannel           ==1?32:0)           |
                            (DynamicShadowMapChannel           ==2?64:0)           |
                            (DynamicShadowMapChannel ==3?128:0);



                        LightTypeAndShadowMapChannelMaskPacked          |= LightProxy-
> GetLightingChannelMask() <<8;
                        LightTypeAndShadowMapChannelMaskPacked          |=
SortedLightInfo.SortKey.Fields.LightType <<16;

                        //Handling local light sources.
                        if((SortedLightInfo.SortKey.Fields.LightType == LightType_Point
&&ViewFamily.EngineShowFlags.PointLights) ||
                            (SortedLightInfo.SortKey.Fields.LightType == LightType_Spot &&
ViewFamily.EngineShowFlags.SpotLights) ||
                            (SortedLightInfo.SortKey.Fields.LightType == LightType_Rect &&
ViewFamily.EngineShowFlags.RectLights))
                        {
                            ForwardLocalLightData.AddUninitialized(1);
                            FForwardLocalLightData& LightData =
ForwardLocalLightData.Last();
```

```
                                        // Track the last one to support clustered deferred if
(!SortedLightInfo.SortKey.Fields.bClusteredDeferredNotSupported)
                                        {
                                                ClusteredSupportedEnd = FMath::Max(ClusteredSupportedEnd,
ForwardLocalLightData.Num());
                                        }
                                        const floatLightFade = GetLightFadeFactor(View, LightProxy);
                                        LightParameters.Color *= LightFade;

                                        LightData.LightPositionAndInvRadius =
FVector4(LightParameters.Position, LightParameters.InvRadius);
                                        LightData.LightColorAndFalloffExponent                  =
FVector4(LightParameters.Color, LightParameters.FalloffExponent);
                                        LightData.LightDirectionAndShadowMapChannelMask                 =
FVector4(LightParameters.Direction, *((float*)&LightTypeAndShadowMapChannelMaskPacked));

                                        LightData.SpotAnglesAndSourceRadiusPacked =
FVector4(LightParameters.SpotAngles.X, LightParameters.SpotAngles.Y,
LightParameters.SourceRadius,0);

                                        LightData.LightTangentAndSoftSourceRadius =
FVector4(LightParameters.Tangent, LightParameters.SoftSourceRadius);

                                        LightData.RectBarnDoor =
FVector4(LightParameters.RectLightBarnCosAngle, LightParameters.RectLightBarnLength,0, 0);


                                        floatVolumetricScatteringIntensity = LightProxy-
> GetVolumetricScatteringIntensity();

                                        if
(LightNeedsSeparateInjectionIntoVolumetricFog(LightSceneInfo,
VisibleLightInfos[LightSceneInfo->Id]))
                                        {
                                                // Disable this lights forward shading volumetric
scattering contribution
                                                VolumetricScatteringIntensity =0;
                                        }

                                        // Pack both values into a single float to keep float4
alignment
                                        constFFloat16 SourceLength16f =
FFloat16(LightParameters.SourceLength);
                                        constFFloat16 VolumetricScatteringIntensity16f =
FFloat16(VolumetricScatteringIntensity);
                                        constuint32 PackedWInt = ((uint32)SourceLength16f.Encoded) |
((uint32)VolumetricScatteringIntensity16f.Encoded <<16);
                                        LightData.SpotAnglesAndSourceRadiusPacked.W = *
(float*)&PackedWInt;

                                        constFSphere BoundingSphere = LightProxy-
> GetBoundingSphere();
                                        const floatDistance =
View.ViewMatrices.GetViewMatrix().TransformPosition(BoundingSphere.Center).Z BoundingSphere.W;   +

                                        FurthestLight = FMath::Max(FurthestLight, Distance);
```

```
                                }
                                //Parallel light source.
                                else if(SortedLightInfo.SortKey.Fields.LightType ==
LightType_Directional           &&ViewFamily.EngineShowFlags.DirectionalLights)
                                {
                                        (......)
                                }
                            }
                        }
                    }

                    constint32 NumLocalLightsFinal = ForwardLocalLightData.Num(); if
                    (ForwardLocalLightData.Num() ==0) {

                        ForwardLocalLightData.AddZeroed();
                    }

                    //Update the light source data toGPU.
                    UpdateDynamicVector4BufferData(ForwardLocalLightData,
View.ForwardLightingResources->ForwardLocalLightBuffer);

                    constFIntPoint LightGridSizeXY =
FIntPoint::DivideAndRoundUp(View.ViewRect.Size(), GLightGridPixelSize);
                    ForwardLightData.ForwardLocalLightBuffer = View.ForwardLightingResources-
> ForwardLocalLightBuffer.SRV;
                    ForwardLightData.NumLocalLights = NumLocalLightsFinal;
                    ForwardLightData.NumReflectionCaptures = View.NumBoxReflectionCaptures +
View.NumSphereReflectionCaptures;
                    ForwardLightData.NumGridCells = LightGridSizeXY.X * LightGridSizeXY.Y *
GLightGridSizeZ;
                    ForwardLightData.CulledGridSize = FIntVector(LightGridSizeXY.X,
LightGridSizeXY.Y, GLightGridSizeZ);
                    ForwardLightData.MaxCulledLightsPerCell                =GMaxCulledLightsPerCell;
                    ForwardLightData.LightGridPixelSizeShift                =
FMath::FloorLog2(GLightGridPixelSize);
                    ForwardLightData.SimpleLightsEndIndex =                SimpleLightsEnd;
                    ForwardLightData.ClusteredDeferredSupportedEndIndex                = ClusteredSupportedEnd;

                    // Clamp far plane to something reasonable floatFarPlane =
                    FMath::Min(FMath::Max(FurthestLight,
View.FurthestReflectionCaptureDistance), (float)HALF_WORLD_MAX /5.0f);
                    FVector ZParams = GetLightGridZParams(View.NearClippingDistance, FarPlane +
10.f);
                    ForwardLightData.LightGridZParams = ZParams;

                    constuint64 NumIndexableLights = CHANGE_LIGHTINDEXTYPE_SIZE &&
!bAllowFormatConversion ? (1llu<< (sizeof(FLightIndexType32) *8llu)) : (1llu<< (sizeof(FLightIndexType) *8llu));


                    constint32 NumCells = LightGridSizeXY.X * LightGridSizeXY.Y * GLightGridSizeZ
* NumCulledGridPrimitiveTypes;

                    if(View.ForwardLightingResources->NumCulledLightsGrid.NumBytes != NumCells *
NumCulledLightsGridStride *sizeof(uint32))
                    {
                        View.ForwardLightingResources-
> NumCulledLightsGrid.Initialize(sizeof(uint32), NumCells * NumCulledLightsGridStride, PF_R32_UINT);
```

```
		}

		if(View.ForwardLightingResources->CulledLightDataGrid.NumBytes != NumCells *
GMaxCulledLightsPerCell * LightIndexTypeSize)
		{
			View.ForwardLightingResources-
> CulledLightDataGrid.Initialize(LightIndexTypeSize, NumCells * GMaxCulledLightsPerCell, LightIndexTypeSize ==
sizeof(uint16) ? PF_R16_UINT : PF_R32_UINT);
		}

		const boolbShouldCacheTemporaryBuffers = View.ViewState != nullptr;
		FForwardLightingCullingResources		LocalCullingResources;
		FForwardLightingCullingResources&		ForwardLightingCullingResources =
bShouldCacheTemporaryBuffers ? View.ViewState->ForwardLightingCullingResources :
LocalCullingResources;

		constuint32 CulledLightLinksElements = NumCells * GMaxCulledLightsPerCell *
LightLinkStride;

		ForwardLightData.DummyRectLightSourceTexture		= GWhiteTexture->TextureRHI;
		ForwardLightData.NumCulledLightsGrid		=		View.ForwardLightingResources-
> NumCulledLightsGrid.SRV;
		ForwardLightData.CulledLightDataGrid		=	View.ForwardLightingResources-
> CulledLightDataGrid.SRV;

		//Creating light source dataUniform Buffer. View.ForwardLightingResources-
		>ForwardLightDataUniformBuffer =
TUniformBufferRef<FForwardLightData>::CreateUniformBufferImmediate(ForwardLightData,
UniformBuffer_SingleFrame);

		//Use belowRDGandCompute ShaderTo clip local lights and reflection probes.

		//The number of thread groups.
		constFIntVector NumGroups =
FIntVector::DivideAndRoundUp(FIntVector(LightGridSizeXY.X, GLightGridSizeZ), LightGridSizeXY.Y,
LightGridInjectionGroupSize);

		TArray<FRHIUnorderedAccessView*, TInlineAllocator<2>> OutUAVs({
			View.ForwardLightingResources->NumCulledLightsGrid.UAV,
			View.ForwardLightingResources->CulledLightDataGrid.UAV });

		RHICmdList.TransitionResources(EResourceTransitionAccess::EWritable,
EResourceTransitionPipeline::EGfxToCompute, OutUAVs.GetData(), OutUAVs.Num());
		{
			FRDGBuilderGraphBuilder(RHICmdList);

				RDG_EVENT_SCOPE(GraphBuilder,"CullLights %ux%ux%u NumLights %u
NumCaptures %u",
					ForwardLightData.CulledGridSize.X,
					ForwardLightData.CulledGridSize.Y,
					ForwardLightData.CulledGridSize.Z,
					ForwardLightData.NumLocalLights,
					ForwardLightData.NumReflectionCaptures);

				FRDGBufferRef CulledLightLinksBuffer =
GraphBuilder.CreateBuffer(FRDGBufferDesc::CreateBufferDesc(sizeof(uint32),
CulledLightLinksElements), TEXT("CulledLightLinks"));
				FRDGBufferRef StartOffsetGridBuffer =
```

```
GraphBuilder.CreateBuffer(FRDGBufferDesc::CreateBufferDesc(sizeof(uint32), TEXT("StartOffsetGrid"NumCells),
));
                    FRDGBufferRef NextCulledLightLinkBuffer =
GraphBuilder.CreateBuffer(FRDGBufferDesc::CreateBufferDesc(sizeof(uint32), TEXT(          1),
"NextCulledLightLink"));
                    FRDGBufferRef NextCulledLightDataBuffer =
GraphBuilder.CreateBuffer(FRDGBufferDesc::CreateBufferDesc(sizeof(uint32), TEXT(          1),
"NextCulledLightData"));

                    //createPassParametersExamples.
                    FLightGridInjectionCS::FParameters *PassParameters          =
GraphBuilder.AllocParameters<FLightGridInjectionCS::FParameters>();

                    PassParameters->View = View.ViewUniformBuffer;
                    PassParameters->ReflectionCapture =
View.ReflectionCaptureUniformBuffer;
                    PassParameters->Forward = View.ForwardLightingResources-
> ForwardLightDataUniformBuffer;
                    PassParameters->RWNumCulledLightsGrid       = View.ForwardLightingResources-
>NumCulledLightsGrid.UAV;
                    PassParameters->RWCulledLightDataGrid       = View.ForwardLightingResources-
> CulledLightDataGrid.UAV;
                    PassParameters->RWNextCulledLightLink       =
GraphBuilder.CreateUAV(NextCulledLightLinkBuffer, PF_R32_UINT);
                    PassParameters->RWStartOffsetGrid =
GraphBuilder.CreateUAV(StartOffsetGridBuffer, PF_R32_UINT);
                    PassParameters->RWCulledLightLinks =
GraphBuilder.CreateUAV(CulledLightLinksBuffer, PF_R32_UINT);

                    FLightGridInjectionCS::FPermutationDomain PermutationVector;
                    PermutationVector.Set<FLightGridInjectionCS::FUseLinkedListDim>
(GLightLinkedListCulling !=0);
                    //Light source grid injectionshader.
                    TShaderMapRef<FLightGridInjectionCS>ComputeShader(View.ShaderMap,
PermutationVector);

                    //Processing light source grid.
                    if(GLightLinkedListCulling !=0) {

                        //CleaningUAV.
                        AddClearUAVPass(GraphBuilder,       PassParameters->RWStartOffsetGrid,
0xFFFFFFFF);
                        AddClearUAVPass(GraphBuilder,       PassParameters-
> RWNextCulledLightLink,0);
                        AddClearUAVPass(GraphBuilder,
GraphBuilder.CreateUAV(NextCulledLightDataBuffer, PF_R32_UINT),          0);
                        //Added light grid injectionPass.
                        FComputeShaderUtils::AddPass(GraphBuilder,
RDG_EVENT_NAME("LightGridInject:LinkedList"), ComputeShader, PassParameters, NumGroups);

                        {
                            //Light source grid compressionshader.
                            TShaderMapRef<FLightGridCompactCS>
ComputeShaderCompact(View.ShaderMap);
                            FLightGridCompactCS::FParameters *PassParametersCompact          =
GraphBuilder.AllocParameters<FLightGridCompactCS::FParameters>();
                            PassParametersCompact->View =       View.ViewUniformBuffer; =
                            PassParametersCompact->Forward
```

```
View.ForwardLightingResources->ForwardLightDataUniformBuffer;

                                    PassParametersCompact->CulledLightLinks =
GraphBuilder.CreateSRV(CulledLightLinksBuffer, PF_R32_UINT);
                                    PassParametersCompact->RWNumCulledLightsGrid        =
View.ForwardLightingResources->NumCulledLightsGrid.UAV;
                                    PassParametersCompact->RWCulledLightDataGrid        =
View.ForwardLightingResources->CulledLightDataGrid.UAV;
                                    PassParametersCompact->RWNextCulledLightData        =
GraphBuilder.CreateUAV(NextCulledLightDataBuffer, PF_R32_UINT);
                                    PassParametersCompact->StartOffsetGrid =
GraphBuilder.CreateSRV(StartOffsetGridBuffer, PF_R32_UINT);

                                    //Increase light grid compressionPass.
                                    FComputeShaderUtils::AddPass(GraphBuilder,
RDG_EVENT_NAME("CompactLinks"), ComputeShaderCompact, PassParametersCompact, NumGroups);
                                }
                            }
                            else
                            {
                                RHICmdList.ClearUAVUint(View.ForwardLightingResources-
> NumCulledLightsGrid.UAV, FUintVector4(0,0,0,0));
                                FComputeShaderUtils::AddPass(GraphBuilder,
RDG_EVENT_NAME("LightGridInject:NotLinkedList"), ComputeShader, PassParameters, NumGroups);

                            }
                        }
                        GraphBuilder.Execute();

                        RHICmdList.TransitionResources(EResourceTransitionAccess::EReadable,
EResourceTransitionPipeline::EComputeToGfx, OutUAVs.GetData(), OutUAVs.Num());
                    }
                }
            }
}
```

It should be noted that only forward rendering, transparent channels using surface shading, and clustered delayed rendering are effective. RDG and Compute Shader are used to process the light grid, and the shaders used are FLightGridInjectionCS and FLightGridCompactCS:

```
// Engine\Source\Runtime\Renderer\Private\LightGridInjection.cpp

classFLightGridInjectionCS: public FGlobalShader {

    DECLARE_GLOBAL_SHADER(FLightGridInjectionCS);
    SHADER_USE_PARAMETER_STRUCT(FLightGridInjectionCS, public:     FGlobalShader)

    classFUseLinkedListDim: SHADER_PERMUTATION_BOOL("USE_LINKED_CULL_LIST"); using
    FPermutationDomain = TShaderPermutationDomain<FUseLinkedListDim>;

    BEGIN_SHADER_PARAMETER_STRUCT(FParameters, )
        //SHADER_PARAMETER_STRUCT_INCLUDE(FLightGridInjectionCommonParameters,
        CommonParameters)
        SHADER_PARAMETER_STRUCT_REF(FReflectionCaptureShaderData,            ReflectionCapture)
        SHADER_PARAMETER_STRUCT_REF(FForwardLightData, Forward)
        SHADER_PARAMETER_STRUCT_REF(FViewUniformShaderParameters,            View)
```

```
        SHADER_PARAMETER_UAV(RWBuffer<uint>,        RWNumCulledLightsGrid)
        SHADER_PARAMETER_UAV(RWBuffer<uint>,        RWCulledLightDataGrid)
        SHADER_PARAMETER_RDG_BUFFER_UAV(RWBuffer<uint>,     RWNextCulledLightLink)
        SHADER_PARAMETER_RDG_BUFFER_UAV(RWBuffer<uint>,     RWStartOffsetGrid)
        SHADER_PARAMETER_RDG_BUFFER_UAV(RWBuffer<uint>,     RWCulledLightLinks)
        SHADER_PARAMETER      _SRV(StrongTypedBuffer<float4>,  LightViewSpacePositionAndRadius)
        SHADER_PARAMETER_SRV(StrongTypedBuffer<float4>,  LightViewSpaceDirAndPreprocAngle)
    END_SHADER_PARAMETER_STRUCT ()

    static bool ShouldCompilePermutation(constFGlobalShaderPermutationParameters& Parameters)

    {
        returnIsFeatureLevelSupported(Parameters.Platform, ERHIFeatureLevel::SM5);
    }

    static void ModifyCompilationEnvironment(constFGlobalShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
    {
        FGlobalShader::ModifyCompilationEnvironment(Parameters, OutEnvironment);
        OutEnvironment.SetDefine(TEXT("THREADGROUP_SIZE"), LightGridInjectionGroupSize);
        FForwardLightingParameters::ModifyCompilationEnvironment(Parameters.Platform, OutEnvironment);

        OutEnvironment.SetDefine(TEXT("LIGHT_LINK_STRIDE"), LightLinkStride);
        OutEnvironment.SetDefine(TEXT("ENABLE_LIGHT_CULLING_VIEW_SPACE_BUILD_DATA"),
        ENABLE_LIGHT_CULLING_VIEW_SPACE_BUILD_DATA);
    }
};

classFLightGridCompactCS: public FGlobalShader {

    DECLARE_GLOBAL_SHADER(FLightGridCompactCS)
    SHADER_USE_PARAMETER_STRUCT(FLightGridCompactCS, public:FGlobalShader)

    BEGIN_SHADER_PARAMETER_STRUCT(FParameters, )
        SHADER_PARAMETER_STRUCT_REF(FForwardLightData,          Forward
        SHADER_PARAMETER_STRUCT_REF(FViewUniformShaderParameters,          View)
        SHADER_PARAMETER_UAV(RWBuffer<uint>,        RWNumCulledLightsGrid)
        SHADER_PARAMETER_UAV(RWBuffer<uint>,        RWCulledLightDataGrid)
        SHADER_PARAMETER_RDG_BUFFER_UAV(RWBuffer<uint>,     RWNextCulledLightData)
        SHADER_PARAMETER_RDG_BUFFER_SRV(Buffer<uint>,     StartOffsetGrid)
        SHADER_PARAMETER_RDG_BUFFER_SRV(Buffer<uint>,     CulledLightLinks)

    END_SHADER_PARAMETER_STRUCT()

    static bool ShouldCompilePermutation(constFGlobalShaderPermutationParameters& Parameters)

    {
        returnIsFeatureLevelSupported(Parameters.Platform, ERHIFeatureLevel::SM5);
    }

    static void ModifyCompilationEnvironment(constFGlobalShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
    {
        FGlobalShader::ModifyCompilationEnvironment(Parameters, OutEnvironment);
        OutEnvironment.SetDefine(TEXT("THREADGROUP_SIZE"), LightGridInjectionGroupSize);
        FForwardLightingParameters::ModifyCompilationEnvironment(Parameters.Platform, OutEnvironment);

        OutEnvironment.SetDefine(TEXT("LIGHT_LINK_STRIDE"),LightLinkStride);
```

```
        OutEnvironment.SetDefine(TEXT("MAX_CAPTURES"), GMaxNumReflectionCaptures);
        OutEnvironment.SetDefine(TEXT("ENABLE_LIGHT_CULLING_VIEW_SPACE_BUILD_DATA"),
        ENABLE_LIGHT_CULLING_VIEW_SPACE_BUILD_DATA);
    }
};
```

The corresponding shader code:

```
// Engine\Shaders\Private\LightGridInjection.usf

RWBuffer<uint>      RWNumCulledLightsGrid;
RWBuffer<uint>      RWCulledLightDataGrid;

RWBuffer<uint>      RWNextCulledLightLink;
RWBuffer<uint>      RWStartOffsetGrid;
RWBuffer<uint>      RWCulledLightLinks;

#defineREFINE_SPOTLIGHT_BOUNDS 1

//fromZThe slice number is converted to the nearest view-space depth value.
float ComputeCellNearViewDepthFromZSlice(uint ZSlice) {

    //Note that the depth of the slice is not threaded, but exponential, the farther away from the camera, the larger the depth range.
    floatSliceDepth = (exp2(ZSlice / ForwardLightData.LightGridZParams.z) -
ForwardLightData.LightGridZParams.y) / ForwardLightData.LightGridZParams.x;

    if(ZSlice == (uint)ForwardLightData.CulledGridSize.z) {

        // Extend the last slice depth max out to world max
        // This allows clamping the depth range to reasonable values,
        // But has the downside that any lights falling into the last depth slice will have very poor culling,

        // Since the view space AABB will be bloated in x and y SliceDepth =
        2000000.0f;
    }

    if(ZSlice ==0) {

        // The exponential distribution of z slices contains an offset, but some screen
pixels
        // may be nearer to the camera than this offset. To avoid false light rejection, the
we set
        // first depth slice to zero to ensure that the AABB includes the [0, offset] range.
depth
        SliceDepth =0.0f;
    }

    returnSliceDepth;
}

//Calculation based on gridAABB
void ComputeCellViewAABB(uint3 GridCoordinate, out float3 ViewTileMin, out float3 ViewTileMax)

{
    // Compute extent of tiles in clip-space. Note that the last tile may extend a bit outside of view if view size is not
evenly divisible tile size.
    constfloat2 InvCulledGridSizeF = (1<< ForwardLightData.LightGridPixelSizeShift) *
```

```
View.ViewSizeAndInvSize.zw;
    constfloat2 TileSize = float2(2.0f,-2.0f) * InvCulledGridSizeF.xy; constfloat2 UnitPlaneMin =
    float2(-1.0f,1.0f);

    float2 UnitPlaneTileMin = GridCoordinate.xy * TileSize + UnitPlaneMin; float2 UnitPlaneTileMax =
    (GridCoordinate.xy +1) * TileSize + UnitPlaneMin;

    floatMinTileZ = ComputeCellNearViewDepthFromZSlice(GridCoordinate.z); floatMaxTileZ =
    ComputeCellNearViewDepthFromZSlice(GridCoordinate.z +1);

    floatMinTileDeviceZ = ConvertToDeviceZ(MinTileZ);
    float4 MinDepthCorner0 = mul(float4(UnitPlaneTileMin.x, UnitPlaneTileMin.y, MinTileDeviceZ,1),
    View.ClipToView);
    float4 MinDepthCorner1 = mul(float4(UnitPlaneTileMax.x, UnitPlaneTileMax.y, MinTileDeviceZ,1),
    View.ClipToView);
    float4 MinDepthCorner2 = mul(float4(UnitPlaneTileMin.x, UnitPlaneTileMax.y, MinTileDeviceZ,1),
    View.ClipToView);
    float4 MinDepthCorner3 = mul(float4(UnitPlaneTileMax.x, UnitPlaneTileMin.y, MinTileDeviceZ,1),
    View.ClipToView);


    floatMaxTileDeviceZ = ConvertToDeviceZ(MaxTileZ);
    float4 MaxDepthCorner0 = mul(float4(UnitPlaneTileMin.x, UnitPlaneTileMin.y, MaxTileDeviceZ,1),
    View.ClipToView);
    float4 MaxDepthCorner1 = mul(float4(UnitPlaneTileMax.x, UnitPlaneTileMax.y, MaxTileDeviceZ,1),
    View.ClipToView);
    float4 MaxDepthCorner2 = mul(float4(UnitPlaneTileMin.x, UnitPlaneTileMax.y, MaxTileDeviceZ,1),
    View.ClipToView);
    float4 MaxDepthCorner3 = mul(float4(UnitPlaneTileMax.x, UnitPlaneTileMin.y, MaxTileDeviceZ,1),
    View.ClipToView);


    float2 ViewMinDepthCorner0 = MinDepthCorner0.xy / MinDepthCorner0.w; float2
    ViewMinDepthCorner1 = MinDepthCorner1.xy / MinDepthCorner1.w; float2
    ViewMinDepthCorner2 = MinDepthCorner2.xy / MinDepthCorner2.w; float2
    ViewMinDepthCorner3 = MinDepthCorner3.xy / MinDepthCor ner3.w; float2
    ViewMaxDepthCorner0 = MaxDepthCorner0.xy /MaxDepthCorner0.w; float2
    ViewMaxDepthCorner1 = MaxDepthCorner1.xy / MaxDepthCorner1.w; float2
    ViewMaxDepthCorner2 = MaxDepthCorner2.xy / MaxDepthCorner2.w; float2
    ViewMaxDepthCorner3 = MaxDepthCorner3.xy / MaxDepthCorner3.w;


    ViewTileMin.xy = min(ViewMinDepthCorner0, ViewMinDepthCorner1); ViewTileMin.xy =
    min(ViewTileMin.xy, ViewMinDepthCorner2); ViewTileMin.xy = min(ViewTileMin.xy,
    ViewMinDepthCorner3); ViewTileMin.xy = min(ViewTileMin.xy, ViewMaxDepthCorner0);
    ViewTileMin.xy = min(ViewTileMin.xy, ViewMaxDepthCorner1); ViewTileMin.xy =
    min(ViewTileMin.xy, ViewMaxDepthCorner2); ViewTileMin.xy = min(ViewTileMin.xy,
    ViewMaxDepthCorner3);



    ViewTileMax.xy = max(ViewMinDepthCorner0, ViewMinDepthCorner1);
    ViewTileMax.xy = max(ViewTileMax.xy, ViewMinDepthCorner2); ViewTileMax.xy =
    max(ViewTileMax.xy, ViewMinDepthCorner3); ViewTileMax.xy = max(ViewTileMax.xy,
    ViewMaxDepthCorner0); ViewTileMax.xy = max(ViewTileMax.xy,
    ViewMaxDepthCorner1); ViewTileMax.xy = max(ViewTileMax.xy,
    ViewMaxDepthCorner2); ViewTileMax.xy = max(ViewTileMax.xy,
    ViewMaxDepthCorner3);


    ViewTileMin.z = MinTileZ;
    ViewTileMax.z = MaxTileZ;
}
```

```
//Cone and sphere intersection test.
bool IntersectConeWithSphere(float3 ConeVertex, float3 ConeAxis,floatConeRadius, float2 CosSinAngle, float4
SphereToTest)
{
	float3 ConeVertexToSphereCenter = SphereToTest.xyz - ConeVertex; float
	ConeVertexToSphereCenterLengthSq = dot(ConeVertexToSphereCenter,
	ConeVertexToSphereCenter);
	floatSphereProjectedOntoConeAxis = dot(ConeVertexToSphereCenter, -ConeAxis); floatDistanceToClosestPoint =
	CosSinAngle.x *sqrt(ConeVertexToSphereCenterLengthSq - SphereProjectedOntoConeAxis *
SphereProjectedOntoConeAxis) - SphereProjectedOntoConeAxis * CosSinAngle.y;


	boolbSphereTooFarFromCone = DistanceToClosestPoint > SphereToTest.w;
	boolbSpherePastConeEnd = SphereProjectedOntoConeAxis > SphereToTest.w + ConeRadius; bool
	bSphereBehindVertex = SphereProjectedOntoConeAxis < -SphereToTest.w; return!(bSphereTooFarFromCone ||
	bSpherePastConeEnd || bSphereBehindVertex);
}

bool AabbOutsidePlane(float3 center, float3 extents, float4 plane) {

	floatdist = dot(float4(center,1.0), plane); floatradius =
	dot(extents,abs(plane.xyz));

	returndist > radius;
}

//Approximate cone andAABBCreate a test on the cone.AABBDedicated floor plan at the center.
bool IsAabbOutsideInfiniteAcuteConeApprox(float3 ConeVertex, float3 ConeAxis,float TanConeAngle, float3
AabbCentre, float3 AabbExt) {

	// 1. find plane (well, base) in which normal lies, and which is perpendicular to axis and center of aabb.

	float3 D = AabbCentre - ConeVertex;

	// perpendicular to cone axis in plane of cone axis and aabb center. float3 M =
	-normalize(cross(cross(D, ConeAxis), ConeAxis)); float3 N = -TanConeAngle * ConeAxis + M;

	float4 Plane = float4(N,0.0);

	returnAabbOutsidePlane(D, AabbExt, Plane);
}

//Light source grid is injected into the main entrance.
[numthreads(THREADGROUP_SIZE,	THREADGROUP_SIZE,	THREADGROUP_SIZE)]
void LightGridInjectionCS(
	uint3 GroupId : SV_GroupID,
	uint3 DispatchThreadId: SV_DispatchThreadID, uint3
	GroupThreadId: SV_GroupThreadID)
{
	uint3 GridCoordinate = DispatchThreadId;

	if(all(GridCoordinate < (uint3)ForwardLightData.CulledGridSize)) {

		//Grid index.
		uint GridIndex = (GridCoordinate.z * ForwardLightData.CulledGridSize.y + GridCoordinate.y) *
ForwardLightData.CulledGridSize.x + GridCoordinate.x;
```

```
#define   CULL_LIGHTS 1
      #if   CULL_LIGHTS
            float3      ViewTileMin;
            float3      ViewTileMax;
            //Compute the grid bounding box.
            ComputeCellViewAABB(GridCoordinate,             ViewTileMin,       ViewTileMax);

            float3 ViewTileCenter =.5f * (ViewTileMin + ViewTileMax); float3 ViewTileExtent
            = ViewTileMax - ViewTileCenter; float3 WorldTileCenter =
            mul(float4(ViewTileCenter,1), View.ViewToTranslatedWorld).xyz -
            View.PreViewTranslation;
            float4 WorldTileBoundingSphere = float4(WorldTileCenter, length(ViewTileExtent));

            uint NumAvailableLinks = ForwardLightData.NumGridCells *
ForwardLightData.MaxCulledLightsPerCell * NUM_CULLED_GRID_PRIMITIVE_TYPES;

            //Traverse all light sources and write the light sources that intersect with the grid into the list.
            LOOP
            for(uint LocalLightIndex =0; LocalLightIndex < ForwardLightData.NumLocalLights; LocalLightIndex++)

            {
                  uint LocalLightBaseIndex = LocalLightIndex * LOCAL_LIGHT_DATA_STRIDE;

#ifENABLE_LIGHT_CULLING_VIEW_SPACE_BUILD_DATA
                  float4 LightPositionAndRadius =
LightViewSpacePositionAndRadius[LocalLightIndex];
                  float3 ViewSpaceLightPosition = LightPositionAndRadius.xyz; floatLightRadius =
                  LightPositionAndRadius.w; float4 LightPositionAndInvRadius =

ForwardLightData.ForwardLocalLightBuffer[LocalLightBaseIndex                  +  0];
#else// !ENABLE_LIGHT_CULLING_VIEW_SPACE_BUILD_DATA
                  float4 LightPositionAndInvRadius =
ForwardLightData.ForwardLocalLightBuffer[LocalLightBaseIndex                  +  0];
                  floatLightRadius =1.0f/LightPositionAndInvRadius.w;

                  float3 ViewSpaceLightPosition = mul(float4(LightPositionAndInvRadius.xyz +
View.PreViewTranslation.xyz,1), View.TranslatedWorldToView).xyz;
#endif// ENABLE_LIGHT_CULLING_VIEW_SPACE_BUILD_DATA

                  floatBoxDistanceSq = ComputeSquaredDistanceFromBoxToPoint(ViewTileCenter,
ViewTileExtent, ViewSpaceLightPosition);

                  if(BoxDistanceSq < LightRadius * LightRadius) {

#if   REFINE_SPOTLIGHT_BOUNDS
                        boolbPassSpotlightTest=true; {

                              float4 ViewSpaceDirAndPreprocAngle =
LightViewSpaceDirAndPreprocAngle[LocalLightIndex];
                              floatTanConeAngle = ViewSpaceDirAndPreprocAngle.w;

                              // Set to 0 for non-acute cones, or non-spot lights. if(TanConeAngle >
                              0.0f) {

                                    float3 ViewSpaceLightDirection = -ViewSpaceDirAndPreprocAngle.xyz; //Light source cone
                                    and gridAABBIntersection test. bPassSpotlightTest=

!IsAabbOutsideInfiniteAcuteConeApprox(ViewSpaceLightPosition, ViewSpaceLightDirection,
```

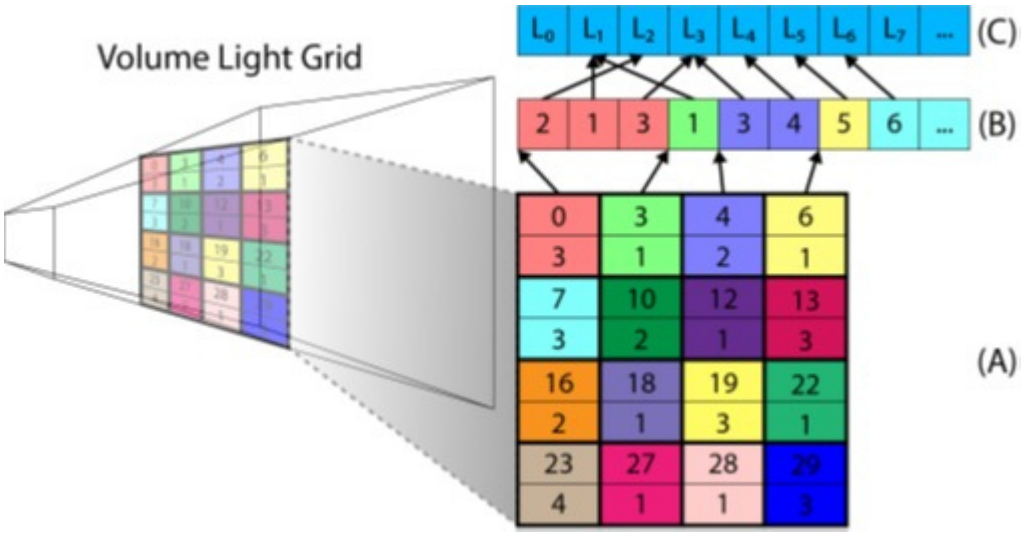
```
TanConeAngle,      ViewTileCenter, ViewTileExtent);
                        }
                    }
                    //    If they intersect, write the light source information.
                    if    (bPassSpotlightTest)
#endif// REFINE_SPOTLIGHT_BOUNDS
                    {
#ifUSE_LINKED_CULL_LIST
                        uint NextLink;
                        InterlockedAdd(RWNextCulledLightLink[0],              1U,   NextLink);

                        //Save the light source link.
                        if(NextLink < NumAvailableLinks) {

                            uint PreviousLink;
                            InterlockedExchange(RWStartOffsetGrid[GridIndex],                NextLink,
PreviousLink);
                            RWCulledLightLinks[NextLink *LIGHT_LINK_STRIDE +0] =
LocalLightIndex;
                            RWCulledLightLinks[NextLink *LIGHT_LINK_STRIDE +1] =
PreviousLink;
                        }

#else
                        uint CulledLightIndex;
                        InterlockedAdd(RWNumCulledLightsGrid[GridIndex              *
NUM_CULLED_LIGHTS_GRID_STRIDE +0],1U, CulledLightIndex);
                        RWNumCulledLightsGrid[GridIndex * NUM_CULLED_LIGHTS_GRID_STRIDE +1] =
GridIndex * ForwardLightData.MaxCulledLightsPerCell;

                        //Save the light index.
                        if(CulledLightIndex < ForwardLightData.MaxCulledLightsPerCell) {

                            RWCulledLightDataGrid[GridIndex *
ForwardLightData.MaxCulledLightsPerCell + CulledLightIndex] = LocalLightIndex;
                        }
#endif
                    }
                }
            }

            //Handle reflection catcher.
            LOOP
            for(uint ReflectionCaptureIndex =0; ReflectionCaptureIndex <
ForwardLightData.NumReflectionCaptures; ReflectionCaptureIndex++)
            {
                float4 CapturePositionAndRadius =
ReflectionCapture.PositionAndRadius[ReflectionCaptureIndex];
                float3 ViewSpaceCapturePosition = mul(float4(CapturePositionAndRadius.xyz +
View.PreViewTranslation.xyz,1), View.TranslatedWorldToView).xyz;

                floatBoxDistanceSq = ComputeSquaredDistanceFromBoxToPoint(ViewTileCenter,
ViewTileExtent, ViewSpaceCapturePosition);

                if(BoxDistanceSq < CapturePositionAndRadius.w * CapturePositionAndRadius.w) {

                    #ifUSE_LINKED_CULL_LIST uint
                      NextLink;
```

```
                        InterlockedAdd(RWNextCulledLightLink[0],1U, NextLink);

                        if(NextLink < NumAvailableLinks) {

                            uint PreviousLink;

InterlockedExchange(RWStartOffsetGrid[ForwardLightData.NumGridCells + GridIndex], NextLink,
PreviousLink);
                            RWCulledLightLinks[NextLink *LIGHT_LINK_STRIDE +0] =
ReflectionCaptureIndex;
                            RWCulledLightLinks[NextLink *LIGHT_LINK_STRIDE +1] =
PreviousLink;
                        }

                    #else
                        uint CulledCaptureIndex;
                        InterlockedAdd(RWNumCulledLightsGrid[(ForwardLightData.NumGridCells                    +
GridIndex) * NUM_CULLED_LIGHTS_GRID_STRIDE +0],1U, CulledCaptureIndex);
                        RWNumCulledLightsGrid[(ForwardLightData.NumGridCells + GridIndex) *
NUM_CULLED_LIGHTS_GRID_STRIDE +1] = (ForwardLightData.NumGridCells + GridIndex) *
ForwardLightData.MaxCulledLightsPerCell;

                        if(CulledCaptureIndex < ForwardLightData.MaxCulledLightsPerCell) {

                            RWCulledLightDataGrid[(ForwardLightData.NumGridCells + GridIndex)
* ForwardLightData.MaxCulledLightsPerCell + CulledCaptureIndex] = ReflectionCaptureIndex;
                        }
                    #endif
                }
            }
#else

            LOOP
            for(uint LocalLightIndex =0; LocalLightIndex < ForwardLightData.NumLocalLights; LocalLightIndex++)

            {
                if(LocalLightIndex < ForwardLightData.MaxCulledLightsPerCell) {

                    RWCulledLightDataGrid[GridIndex * ForwardLightData.MaxCulledLightsPerCell
+ LocalLightIndex] = LocalLightIndex;
                }
            }

            RWNumCulledLightsGrid[GridIndex * NUM_CULLED_LIGHTS_GRID_STRIDE +0] =
            ForwardLightData.NumLocalLights;
            RWNumCulledLightsGrid[GridIndex * NUM_CULLED_LIGHTS_GRID_STRIDE +1] = GridIndex *
            ForwardLightData.MaxCulledLightsPerCell;

            LOOP
            for(uint ReflectionCaptureIndex =0; ReflectionCaptureIndex <
ForwardLightData.NumReflectionCaptures; ReflectionCaptureIndex++)
            {
                if(ReflectionCaptureIndex < ForwardLightData.MaxCulledLightsPerCell) {

                    RWCulledLightDataGrid[(ForwardLightData.NumGridCells + GridIndex) *
ForwardLightData.MaxCulledLightsPerCell + ReflectionCaptureIndex] =
ReflectionCaptureIndex;
                }
```

```
            }

            RWNumCulledLightsGrid[(ForwardLightData.NumGridCells + GridIndex) *
NUM_CULLED_LIGHTS_GRID_STRIDE +0] = ForwardLightData.NumReflectionCaptures;
            RWNumCulledLightsGrid[(ForwardLightData.NumGridCells + GridIndex) *
NUM_CULLED_LIGHTS_GRID_STRIDE +1] = (ForwardLightData.NumGridCells + GridIndex) *
ForwardLightData.MaxCulledLightsPerCell;
#endif
        }
}

RWBuffer<uint>      RWNextCulledLightData;
Buffer<uint>       StartOffsetGrid;
Buffer<uint>       CulledLightLinks;

//Compresses the specified reverse linked list.
void CompactReverseLinkedList(uint GridIndex, uint SceneMax) {

        uint NumCulledLights =0;
        uint StartLinkOffset = StartOffsetGrid[GridIndex]; uint LinkOffset =
        StartLinkOffset;

        // Traverse the linked list to count how many culled indices we have while(LinkOffset !=
        0xFFFFFFFF&& NumCulledLights < SceneMax) {

            NumCulledLights++;
            LinkOffset = CulledLightLinks[LinkOffset * LIGHT_LINK_STRIDE +1];
        }

        uint CulledLightDataStart;
        InterlockedAdd(RWNextCulledLightData[0], NumCulledLights, CulledLightDataStart);
        RWNumCulledLightsGrid[GridIndex * NUM_CULLED_LIGHTS_GRID_STRIDE +0] = NumCulledLights;

        RWNumCulledLightsGrid[GridIndex * NUM_CULLED_LIGHTS_GRID_STRIDE +1] =
        CulledLightDataStart;

        LinkOffset = StartLinkOffset; uint
        CulledLightIndex =0;

        while(LinkOffset !=0xFFFFFFFF&& CulledLightIndex < NumCulledLights) {

            // Reverse the order as we write them out, which restores the original order before the reverse linked
list was built
            // Reflection captures are order dependent RWCulledLightDataGrid[CulledLightDataStart + NumCulledLights -
            CulledLightIndex - 1] = CulledLightLinks[LinkOffset * LIGHT_LINK_STRIDE +0];

            CulledLightIndex++;
            LinkOffset = CulledLightLinks[LinkOffset * LIGHT_LINK_STRIDE +1];
        }
}

//The light source grid compresses the main entrance.
[numthreads(THREADGROUP_SIZE,       THREADGROUP_SIZE,     THREADGROUP_SIZE)]
void LightGridCompactCS(
        uint3 GroupId : SV_GroupID,
        uint3 DispatchThreadId: SV_DispatchThreadID, uint3
        GroupThreadId: SV_GroupThreadID)
{
```

```
    uint3 GridCoordinate = DispatchThreadId;

    if(all(GridCoordinate < ForwardLightData.CulledGridSize)) {

        uint GridIndex = (GridCoordinate.z * ForwardLightData.CulledGridSize.y + GridCoordinate.y) *
ForwardLightData.CulledGridSize.x + GridCoordinate.x;

        //Compressed light source
        CompactReverseLinkedList(GridIndex, ForwardLightData.NumLocalLights);

        //Compressed reflection catcher.
        CompactReverseLinkedList(ForwardLightData.NumGridCells + GridIndex,
        ForwardLightData.NumReflectionCaptures);
    }
}
```

Storing light sources in a grid is somewhat similar tothe Volume Tiled Forward Rendering mentioned in the previous article:



Schematic diagram of the Volume Light Grid data structure.

5.5 LightingPass

This section mainly elaborates on the rendering process, rendering status, Shader logic, etc. of LightingPass.

5.5.1 LightingPass rendering process

The previous article also involved the rendering process and some core logic of LightingPass. This section briefly reviews it. LightingPass comes afterFDeferredShadingSceneRenderer::RenderBasePass:

```
void FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList) {

    (.....)

    //RenderingBase Pass.
```

```
    RenderBasePass(RHICmdList, ...);

    (......)

    //Rendering light sources.
    RenderLights(RHICmdList, ...);

    (......)
}
```

RenderLightsLet's briefly review the rendering logic again RenderLight:

```
// Engine\Source\Runtime\Renderer\Private\LightRendering.cpp

//Renders all lights.
void FDeferredShadingSceneRenderer::RenderLights(FRHICommandListImmediate& RHICmdList,
FSortedLightSetSceneInfo &SortedLightSet,constFHairStrandsDatas* HairDatas) {

    (......)

    boolbStencilBufferDirty =false;

    constFSimpleLightArray &SimpleLights = SortedLightSet.SimpleLights; const
    TArray<FSortedLightSceneInfo, SceneRenderingAllocator> &SortedLights =
    SortedLightSet.SortedLights;
    //The starting index of the light source with shadows.
    constint32 AttenuationLightStart = SortedLightSet.AttenuationLightStart; constint32
    SimpleLightsEnd = SortedLightSet.SimpleLightsEnd;


    //Direct Lighting
    {
        SCOPED_DRAW_EVENT(RHICmdList, DirectLighting);

        FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

        (......)

        //No shadow lighting
        if(ViewFamily.EngineShowFlags.DirectLighting) {

            SCOPED_DRAW_EVENT(RHICmdList, NonShadowLights);

            //The starting index of the light source without shadows.
            int32 StandardDeferredStart = SortedLightSet.SimpleLightsEnd; bool
            bRenderSimpleLightsStandardDeferred =
SortedLightSet.SimpleLights.InstanceData.Num() >0;

            //Clustered delayed lighting.
            if(ShouldUseClusteredDeferredShading() && AreClusteredLightsInLightGrid()) {

                StandardDeferredStart = SortedLightSet.ClusteredSupportedEnd;
                bRenderSimpleLightsStandardDeferred =false; //Added clustered delayed
                renderingPass.
                AddClusteredDeferredShadingPass(RHICmdList, SortedLightSet);
            }
            //Tiled deferred lighting.
            else if(CanUseTiledDeferred())
```

```
        {
            (……)

            if  (ShouldUseTiledDeferred(SortedLightSet.TiledSupportedEnd) &&
!bAnyViewIsStereo)
            {
StandardDeferredStart = SortedLightSet.TiledSupportedEnd;
bRenderSimpleLightsStandardDeferred =false; //Rendering tiled deferred
lighting.
RenderTiledDeferredLighting(RHICmdList, SortedLights,
SortedLightSet.SimpleLightsEnd, SortedLightSet.TiledSupportedEnd, SimpleLights);
            }
        }

        //  Simple lighting.
        if  (bRenderSimpleLightsStandardDeferred)
        {
            SceneContext.BeginRenderingSceneColor(RHICmdList,
ESimpleRenderTargetMode::EExistingColorAndDepth,
FExclusiveDepthStencil::DepthRead_StencilWrite);
            //Rendering simple lighting.
            RenderSimpleLightsStandardDeferred(RHICmdList,
SortedLightSet.SimpleLights);
            SceneContext.FinishRenderingSceneColor(RHICmdList);
        }

        //  Standard deferred lighting.
        if  (!bUseHairLighting)
        {
            SCOPED_DRAW_EVENT(RHICmdList,       StandardDeferredLighting);

            SceneContext.BeginRenderingSceneColor(RHICmdList,
ESimpleRenderTargetMode::EExistingColorAndDepth,
FExclusiveDepthStencil::DepthRead_StencilWrite,true);

            for(int32 LightIndex = StandardDeferredStart; LightIndex <
AttenuationLightStart; LightIndex++)
            {
                constFSortedLightSceneInfo& SortedLightInfo =
SortedLights[LightIndex];
                constFLightSceneInfo*constLightSceneInfo =
SortedLightInfo.LightSceneInfo;

                //Renders shadowless lighting.
                RenderLight(RHICmdList, LightSceneInfo, nullptr, nullptr,false,
false);
            }

            SceneContext.FinishRenderingSceneColor(RHICmdList);
        }

        (……)
    }

    (……)

    //Lighting with shadows
    {
```

```
SCOPED_DRAW_EVENT(RHICmdList, ShadowedLights);

constint32 DenoiserMode = CVarShadowUseDenoiser.GetValueOnRenderThread();

constIScreenSpaceDenoiser* DefaultDenoiser =
IScreenSpaceDenoiser::GetDefaultDenoiser();
constIScreenSpaceDenoiser* DenoiserToUse = DenoiserMode ==1?
DefaultDenoiser : GScreenSpaceDenoiser;

TArray<TRefCountPtr<IPooledRenderTarget>>            PreprocessedShadowMaskTextures;
TArray<TRefCountPtr<IPooledRenderTarget>>
PreprocessedShadowMaskSubPixelTextures;

constint32 MaxDenoisingBatchSize =
FMath::Clamp(CVarMaxShadowDenoisingBatchSize.GetValueOnRenderThread(),                1,
IScreenSpaceDenoiser::kMaxBatchSize);
constint32 MaxRTShadowBatchSize =
CVarMaxShadowRayTracingBatchSize.GetValueOnRenderThread();
const boolbDoShadowDenoisingBatching = DenoiserMode !=0&&
MaxDenoisingBatchSize >1;

(.....)

boolbDirectLighting = ViewFamily.EngineShowFlags.DirectLighting; bool
bShadowMaskReadable =false; TRefCountPtr<IPooledRenderTarget>
TRefCountPtr<IPooledRenderTarget>            ScreenShadowMaskTexture;
                                             ScreenShadowMaskSubPixelTexture;

//Renders shadowed lights and light function lights.
for(int32 LightIndex = AttenuationLightStart; LightIndex <
SortedLights.Num(); LightIndex++)
{
    constFSortedLightSceneInfo& SortedLightInfo = SortedLights[LightIndex]; constFLightSceneInfo&
    LightSceneInfo = *SortedLightInfo.LightSceneInfo;

    const boolbDrawShadows = SortedLightInfo.SortKey.Fields.bShadowed &&
!ShouldRenderRayTracingStochasticRectLight(LightSceneInfo);
    boolbDrawLightFunction = SortedLightInfo.SortKey.Fields.bLightFunction; bool
    bDrawPreviewIndicator =
ViewFamily.EngineShowFlags.PreviewShadowsIndicator && !
LightSceneInfo.IsPrecomputedLightingValid() && LightSceneInfo.Proxy-
> HasStaticShadowing();
    boolbInjectedTranslucentVolume =false; bool
    bUsedShadowMaskTexture =false;
    const boolbDrawHairShadow = bDrawShadows && bUseHairLighting; const bool
    bUseHairDeepShadow = bDrawShadows && bUseHairLighting &&
LightSceneInfo.Proxy->CastsHairStrandsDeepShadow();

    FScopeCycleCounterContext(LightSceneInfo.Proxy->GetStatId());

    if((bDrawShadows || bDrawLightFunction || bDrawPreviewIndicator) &&
!ScreenShadowMaskTexture.IsValid())
    {
        SceneContext.AllocateScreenShadowMask(RHICmdList,
ScreenShadowMaskTexture);
        bShadowMaskReadable =false; if
        (bUseHairLighting)
        {
```

```
                                SceneContext.AllocateScreenShadowMask(RHICmdList,
ScreenShadowMaskSubPixelTexture,true);
                        }
                    }

                    FString LightNameWithLevel;
                    GetLightNameForDrawEvent(LightSceneInfo.Proxy,          LightNameWithLevel);
                    SCOPED_DRAW_EVENTF(RHICmdList, EventLightPass,          * LightNameWithLevel);

                    if(bDrawShadows)
                    {
                        INC_DWORD_STAT(STAT_NumShadowedLights);

                        constFLightOcclusionType OcclusionType =
GetLightOcclusionType(*LightSceneInfo.Proxy);

                        (......)

                        //Processes shadow masking maps.
                        else// (OcclusionType == FOcclusionType::Shadowmap) {

                            for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                                constFViewInfo& View = Views[ViewIndex];
                                View.HeightfieldLightingViewInfo.ClearShadowing(View,
RHICmdList,     LightSceneInfo);
                            }

                            //Clean up shadow mask map.
                            autoClearShadowMask = [&](TRefCountPtr<IPooledRenderTarget>&
InScreenShadowMaskTexture)
                            {
                                const boolbClearLightScreenExtentsOnly =
CVarAllowClearLightSceneExtentsOnly.GetValueOnRenderThread() &&
SortedLightInfo.SortKey.Fields.LightType != LightType_Directional;
                                boolbClearToWhite = !bClearLightScreenExtentsOnly;

                                FRHIRenderPassInfoRPInfo(InScreenShadowMaskTexture-
> GetRenderTargetItem().TargetableTexture, ERenderTargetActions::Load_Store);
                                RPInfo.DepthStencilRenderTarget.Action =
MakeDepthStencilTargetActions(ERenderTargetActions::Load_DontStore,
ERenderTargetActions::Load_Store);
                                RPInfo.DepthStencilRenderTarget.DepthStencilTarget =
SceneContext.GetSceneDepthSurface();
                                RPInfo.DepthStencilRenderTarget.ExclusiveDepthStencil                    =
FExclusiveDepthStencil::DepthRead_StencilWrite;
                                if(bClearToWhite)
                                {
                                    RPInfo.ColorRenderTargets[0].Action =
ERenderTargetActions::Clear_Store;
                                }

                                TransitionRenderPassTargets(RHICmdList, RPInfo);
                                RHICmdList.BeginRenderPass(RPInfo,
TEXT("ClearScreenShadowMask"));
                                if(bClearLightScreenExtentsOnly) {

                                    SCOPED_DRAW_EVENT(RHICmdList, ClearQuad);
```

```
                                        for(int32 ViewIndex =0; ViewIndex < Views.Num();
ViewIndex++)
                                        {
                                            constFViewInfo& View = Views[ViewIndex]; FIntRect
                                            ScissorRect;

                                            if(!LightSceneInfo.Proxy->GetScissorRect(ScissorRect,
View,    View.ViewRect)
                                            {
                                                ScissorRect = View.ViewRect;
                                            }

                                            if(ScissorRect.Min.X < ScissorRect.Max.X &&
ScissorRect.Min.Y        <  ScissorRect.Max.Y)
                                            {
                                                RHICmdList.SetViewport(ScissorRect.Min.X,
ScissorRect.Min.Y,0.0f, ScissorRect.Max.X, ScissorRect.Max.Y,1.0f);
                                                DrawClearQuad(RHICmdList,true, FLinearColor(1,1,
1,1),false,0,false,0);
                                            }
                                            else
                                            {
                                                LightSceneInfo.Proxy->GetScissorRect(ScissorRect,
View, View.ViewRect);
                                            }
                                        }
                                    }
                                    RHICmdList.EndRenderPass();
                                };

                                ClearShadowMask(ScreenShadowMaskTexture); if
                                (ScreenShadowMaskSubPixelTexture) {

                                    ClearShadowMask(ScreenShadowMaskSubPixelTexture);
                                }

                                RenderShadowProjections(RHICmdList, &LightSceneInfo,
ScreenShadowMaskTexture, ScreenShadowMaskSubPixelTexture, HairDatas,
bInjectedTranslucentVolume);
                            }

                            bUsedShadowMaskTexture =true;
                        }

                        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                            constFViewInfo& View = Views[ViewIndex];
                            View.HeightfieldLightingViewInfo.ComputeLighting(View, RHICmdList,
LightSceneInfo);
                        }

                        //  Processing lighting functions (light
                        if  function). (bDirectLighting)
                        {
                            if(bDrawLightFunction) {

                                const boolbLightFunctionRendered =
```

```
RenderLightFunction(RHICmdList, &LightSceneInfo, ScreenShadowMaskTexture, bDrawShadows, false);

                        bUsedShadowMaskTexture |= bLightFunctionRendered;
                    }

                    (.....)
                }

                if(bUsedShadowMaskTexture) {

                    check(ScreenShadowMaskTexture);
                    RHICmdList.CopyToResolveTarget(ScreenShadowMaskTexture-
> GetRenderTargetItem().TargetableTexture, ScreenShadowMaskTexture-
> GetRenderTargetItem().ShaderResourceTexture, FResolveParams(FResolveRect()));
                    if(ScreenShadowMaskSubPixelTexture) {

                        RHICmdList.CopyToResolveTarget(ScreenShadowMaskSubPixelTexture-
> GetRenderTargetItem().TargetableTexture, ScreenShadowMaskSubPixelTexture-
> GetRenderTargetItem().ShaderResourceTexture, FResolveParams(FResolveRect()));
                    }

                    if(!bShadowMaskReadable) {


RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
ScreenShadowMaskTexture->GetRenderTargetItem().ShaderResourceTexture);
                        if(ScreenShadowMaskSubPixelTexture) {


RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
ScreenShadowMaskSubPixelTexture->GetRenderTargetItem().ShaderResourceTexture);
                        }
                        bShadowMaskReadable =true;
                    }

                    GVisualizeTexture.SetCheckPoint(RHICmdList, ScreenShadowMaskTexture); if
                    (ScreenShadowMaskSubPixelTexture) {

                        GVisualizeTexture.SetCheckPoint(RHICmdList,
ScreenShadowMaskSubPixelTexture);
                    }
                }

                (.....)

                //Rendering standard deferred lighting.
                {
                    SCOPED_DRAW_EVENT(RHICmdList, StandardDeferredLighting);
                    SceneContext.BeginRenderingSceneColor(RHICmdList,
ESimpleRenderTargetMode::EExistingColorAndDepth,
FExclusiveDepthStencil::DepthRead_StencilWrite,true);

                    IPooledRenderTarget* LightShadowMaskTexture = nullptr; IPooledRenderTarget*
                    LightShadowMaskSubPixelTexture = nullptr; if(bUsedShadowMaskTexture) {


                        LightShadowMaskTexture = ScreenShadowMaskTexture;
                        LightShadowMaskSubPixelTexture = ScreenShadowMaskSubPixelTexture;
```

```
                        }

                        if (bDirectLighting)
                        {
                            //Renders light sources with shadows.
                            RenderLight(RHICmdList, false,&LightSceneInfo, LightShadowMaskTexture,
InHairVisibilityViews, true);
                        }

                        SceneContext.FinishRenderingSceneColor(RHICmdList);

                        (.....)
                    }
                }
            }
        }
}

//Renders a single light source.
void FDeferredShadingSceneRenderer::RenderLight(FRHICommandList& RHICmdList,const FLightSceneInfo*
LightSceneInfo, IPooledRenderTarget* ScreenShadowMaskTexture,const FHairStrandsVisibilityViews*
InHairVisibilityViews,boolbRenderOverlap,bool bIssueDrawEvent)

{
    FGraphicsPipelineStateInitializer GraphicsPSOInit;
    RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);

    //Set the blending state to Overlay so that the light intensity of all lights is added to the same texture.
    GraphicsPSOInit.BlendState = TStaticBlendState<CW_RGBA, BO_Add, BF_One, BF_One, BO_Add, BF_One,
    BF_One>::GetRHI();

    GraphicsPSOInit.PrimitiveType = PT_TriangleList;

    constFSphere LightBounds = LightSceneInfo->Proxy->GetBoundingSphere(); const bool
    bTransmission = LightSceneInfo->Proxy->Transmission();

    //Traverse allview,Give this light source to eachviewDraw them all once.
    for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        FViewInfo& View = Views[ViewIndex];

        // Make sure the light source is valid in this view.
        if (!LightSceneInfo->ShouldRenderLight(View))
        {
            continue;
        }

        boolbUseIESTexture =false;

        if(View.Family->EngineShowFlags.TexturedLightProfiles) {

            bUseIESTexture = (LightSceneInfo->Proxy->GetIESTextureResource() !=0);
        }

        RHICmdList.SetViewport(View.ViewRect.Min.X, View.ViewRect.Min.Y,0.0f, View.ViewRect.Max.X,
View.ViewRect.Max.Y,1.0f);

        (.....)
```

```
//Draws a parallel directional light.
if(LightSceneInfo->Proxy->GetLightType() == LightType_Directional) {

    GraphicsPSOInit.bDepthBounds =false; TShaderMapRef<TDeferredLightVS<false> >
    VertexShader(View.ShaderMap);

    GraphicsPSOInit.RasterizerState =       TStaticRasterizerState<FM_Solid,
CM_None>::GetRHI();
    GraphicsPSOInit.DepthStencilState       = TStaticDepthStencilState<false,
CF_Always>::GetRHI();

    // Set the delayed light sourceshaderandpso
    if parameter. (bRenderOverlap)
    {
        TShaderMapRef<TDeferredLightOverlapPS<false>        >
PixelShader(View.ShaderMap);
        GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI       =
GFilterVertexDeclaration.VertexDeclarationRHI;
        GraphicsPSOInit.BoundShaderState.VertexShaderRHI        =
VertexShader.GetVertexShader();
        GraphicsPSOInit.BoundShaderState.PixelShaderRHI       =
PixelShader.GetPixelShader();
        SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
        >SetParameters(RHICmdList, View, LightSceneInfo);
    }
    else
    {
        const boolbAtmospherePerPixelTransmittance = LightSceneInfo->Proxy-
> IsUsedAsAtmosphereSunLight() && ShouldApplyAtmosphereLightPerPixelTransmittance(Scene, View.Family-
>EngineShowFlags);

        FDeferredLightPS::FPermutationDomain PermutationVector;
        PermutationVector.Set< FDeferredLightPS::FSourceShapeDim >(
ELightSourceShape::Directional );
        PermutationVector.Set< FDeferredLightPS::FIESProfileDim >(false); PermutationVector.Set<
        FDeferredLightPS::FInverseSquaredDim >(false); PermutationVector.Set<
        FDeferredLightPS::FVisualizeCullingDim                          > (
View.Family->EngineShowFlags.VisualizeLightCulling );
        PermutationVector.Set< FDeferredLightPS::FLightingChannelsDim       > (
View.bUsesLightingChannels );
        PermutationVector.Set< FDeferredLightPS::FTransmissionDim >( bTransmission
) ;
        PermutationVector.Set< FDeferredLightPS::FHairLighting>(bHairLighting?1
:0);
        PermutationVector.Set< FDeferredLightPS::FAtmosphereTransmittance >
(bAtmospherePerPixelTransmittance);

        TShaderMapRef< FDeferredLightPS >PixelShader( View.ShaderMap,
PermutationVector );
        GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI       =
GFilterVertexDeclaration.VertexDeclarationRHI;
        GraphicsPSOInit.BoundShaderState.VertexShaderRHI        =
VertexShader.GetVertexShader();
        GraphicsPSOInit.BoundShaderState.PixelShaderRHI       =
PixelShader.GetPixelShader();

        SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);
```

```
                PixelShader->SetParameters(RHICmdList, View, LightSceneInfo,
ScreenShadowMaskTexture, (bHairLighting) ? &RenderLightParams : nullptr);
            }

            VertexShader->SetParameters(RHICmdList, View, LightSceneInfo);

            //Since it is a parallel light, it will cover all areas of the screen space, so a full-screen rectangle is used for
            drawing. DrawRectangle(
                RHICmdList,
                0,0,
                View.ViewRect.Width(), View.ViewRect.Height(),
                View.ViewRect.Min.X, View.ViewRect.Min.Y,
                View.ViewRect.Width(), View.ViewRect.Height(), View.
                ViewRect.Size(),
                FSceneRenderTargets::Get(RHICmdList).GetBufferSizeXY(), VertexShader,


                EDRF_UseTriangleOptimization);
        }
        else  //  Local light source
        {
            //Whether to enable depth bounding box test (DBT).
            GraphicsPSOInit.bDepthBounds = GSupportsDepthBoundsTest &&
GAllowDepthBoundsTest !=0;

            TShaderMapRef<TDeferredLightVS<true> > VertexShader(View.ShaderMap);

            SetBoundingGeometryRasterizerAndDepthState(GraphicsPSOInit, View,
LightBounds);

            //  Set the delayed light sourceshaderandpso
            if  parameter. (bRenderOverlap)
            {
                TShaderMapRef<TDeferredLightOverlapPS<true>            >  PixelShader(View.ShaderMap);
                GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                =
GetVertexDeclarationFVector4();
                GraphicsPSOInit.BoundShaderState.VertexShaderRHI              =
VertexShader.GetVertexShader();
                GraphicsPSOInit.BoundShaderState.PixelShaderRHI               =
PixelShader.GetPixelShader();

                SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
                >SetParameters(RHICmdList, View, LightSceneInfo);
            }
            else
            {
                FDeferredLightPS::FPermutationDomain PermutatPioenrVmeuctaotri.oSneVt<e ctor;
                FDeferredLightPS::FSourceShapeDim                                    > (
LightSceneInfo->Proxy->IsRectLight() ? ELightSourceShape::Rect :
ELightSourceShape::Capsule );
                PermutationVector.Set< FDeferredLightPS::FSourceTextureDim                > (
LightSceneInfo->Proxy->IsRectLight() && LightSceneInfo->Proxy->HasSourceTexture() );
                PermutationVector.Set< FDeferredLightPS::FIESProfileDim >( bUseIESTexture
) ;
                PermutationVector.Set< FDeferredLightPS::FInverseSquaredDim >(
LightSceneInfo->Proxy->IsInverseSquared() );
                PermutationVector.Set< FDeferredLightPS::FVisualizeCullingDim             > (
View.Family->EngineShowFlags.VisualizeLightCulling );
                PermutationVector.Set< FDeferredLightPS::FLightingChannelsDim             > (
```

```
View.bUsesLightingChannels );
                    PermutationVector.Set< FDeferredLightPS::FTransmissionDim >( bTransmission
) ;
                    PermutationVector.Set< FDeferredLightPS::FHairLighting>(bHairLighting?1
:0);
                    PermutationVector.Set < FDeferredLightPS::FAtmosphereTransmittance >
(false);

                    TShaderMapRef< FDeferredLightPS >PixelShader( View.ShaderMap,
PermutationVector );
                    GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                =
GetVertexDeclarationFVector4();
                    GraphicsPSOInit.BoundShaderState.VertexShaderRHI                =
VertexShader.GetVertexShader();
                    GraphicsPSOInit.BoundShaderState.PixelShaderRHI                =
PixelShader.GetPixelShader();

                    SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
                    >SetParameters(RHICmdList, View, LightSceneInfo,
ScreenShadowMaskTexture, (bHairLighting) ? &RenderLightParams : nullptr);
                }

                VertexShader->SetParameters(RHICmdList, View, LightSceneInfo);

                //  Depth bounding box test (DBT)Only works in light sources with shadows, and can effectively cull pixels outside the
                if  depth range. (GraphicsPSOInit.bDepthBounds)
                {
                    //UE4Using the inverse depth (reversed depth),sofar<near. floatNearDepth =1.f; floatFarDepth =0.f;
                    CalculateLightNearFarDepthFromBounds(View,LightBounds,NearDepth,FarDepth);



                    if(NearDepth <= FarDepth) {

                        NearDepth    =  1.0f;
                        FarDepth    =  0.0f;
                    }

                    //Set the depth bounding box parameters.
                    RHICmdList.SetDepthBounds(FarDepth, NearDepth);
                }

                //Point or area lights are drawn using spheres.
                if(LightSceneInfo->Proxy->GetLightType() == LightType_Point ||
                    LightSceneInfo->Proxy->GetLightType() == LightType_Rect )
                {
                    StencilingGeometry::DrawSphere(RHICmdList);
                }
                //Spotlights are drawn using cones.
                else if(LightSceneInfo->Proxy->GetLightType() == LightType_Spot) {

                    StencilingGeometry::DrawCone(RHICmdList);
                }
            }
        }
}
```

5.5.2 LightingPass rendering status

This section will explain the various rendering states, Shader bindings, and drawing parameters used when rendering LightingPass.

First, let's look at the mixed state when rendering a single light source:

```
GraphicsPSOInit.BlendState = TStaticBlendState<CW_RGBA, BO_Add, BF_One, BF_One, BO_Add, BF_One, BF_One>::GetRHI();
```

This means that the color and Alpha of the light source are in a superposition state, and the formula is as follows:

$$\text{FinalColor} = 1 \cdot \text{SourceColor} + 1 \cdot \text{DestColor} = \text{SourceColor} + \text{DestColor}$$
$$\text{FinalAlpha} = 1 \cdot \text{SourceAlpha} + 1 \cdot \text{DestAlpha} = \text{SourceAlpha} + \text{DestAlpha}$$

This is also in line with common sense in physics. When a single surface object is illuminated by multiple light sources, both the color and intensity should be in a superposition state.

Other rendering states are as follows (taking parallel light as an example):

Rendering Status	value	describe
PrimitiveType	PT_TriangleList	The primitive type is triangle.
bDepthBounds	false	The depth boundary values of different types of light sources are different.
RasterizerState	StaticRasterizerState<FM_Solid, CM_None>	Solid rendering, with back or front face clipping turned off.
DepthStencilState	TStaticDepthStencilState<false, CF_Always>	Disable depth writing, depth comparison function always passes the test.
VertexShader	TDeferredLightVS	Vertex shader for non-radiative lighting.
PixelShader	FDeferredLightPS	Pixel shader.

The following is a breakdown of the VS and PS C++ code used to render the light source:

```
// Engine\Source\Runtime\Renderer\Private\LightRendering.h

//Vertex shader.
template<boolbRadialLight>
```

```
classTDeferredLightVS: public FGlobalShader {

	DECLARE_SHADER_TYPE(TDeferredLightVS,Global); public:


	static bool ShouldCompilePermutation(constFGlobalShaderPermutationParameters& Parameters)

	{
		returnbRadialLight ? IsFeatureLevelSupported(Parameters.Platform,
ERHIFeatureLevel::SM5):true;
	}

	static void ModifyCompilationEnvironment(constFGlobalShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
	{
		FGlobalShader::ModifyCompilationEnvironment(Parameters, OutEnvironment);
		OutEnvironment.SetDefine(TEXT("SHADER_RADIAL_LIGHT"), bRadialLight ?1:0);
	}

	TDeferredLightVS()		{}
	TDeferredLightVSconstShaderMetaType::CompiledShaderInitializerType&		Initializer):
		FGlobalShader(Initializer)
	{
		StencilingGeometryParameters.Bind(Initializer.ParameterMap);
	}

	//Set the parameters.
	void SetParameters(FRHICommandList& RHICmdList,constFViewInfo& View,const FLightSceneInfo*
	LightSceneInfo)
	{
		//Setting up the viewUniform Buffer.
		FGlobalShader::SetParameters<FViewUniformShaderParameters>(RHICmdList,
RHICmdList.GetBoundVertexShader(), View.ViewUniformBuffer);
		StencilingGeometryParameters.Set(RHICmdList, this, View, LightSceneInfo);
	}

	//Set simple light parameters.
	void SetSimpleLightParameters(FRHICommandList& RHICmdList,constFViewInfo& View, const
	   FSphere & LightBounds)
	{
		FGlobalShader::SetParameters<FViewUniformShaderParameters>(RHICmdList,
RHICmdList.GetBoundVertexShader(), View.ViewUniformBuffer);

		FVector4 StencilingSpherePosAndScale;

StencilingGeometry::GStencilSphereVertexBuffer.CalcTransform(StencilingSpherePosAndScale, LightBounds,
View.ViewMatrices.GetPreViewTranslation());
		StencilingGeometryParameters.Set(RHICmdList, this, StencilingSpherePosAndScale);
	}

private:

	LAYOUT_FIELD(FStencilingGeometryShaderParameters, StencilingGeometryParameters);
};

//	Engine\Source\Runtime\Renderer\Private\LightRendering.cpp

//	The pixel shader for the light.
```

```
classFDeferredLightPS: public FGlobalShader {

	DECLARE_SHADER_TYPE(FDeferredLightPS, Global)

	classFSourceShapeDim                : SHADER_PERMUTATION_ENUM_CLASS("LIGHT_SOURCE_SHAPE",
ELightSourceShape);
	class	FSourceTextureDim               : SHADER_PERMUTATION_BOOL("USE_SOURCE_TEXTURE"
	class	FIESProfileDim              : SHADER_PERMUTATION_BOOL("USE_IES_PROFILE"
	class	FInverseSquaredDim          :   SHADER_PERMUTATION_BOOL("INVERSE_SQUARED_FALLOFF"); :
	class	FVisualizeCullingDim                SHADER_PERMUTATION_BOOL("VISUALIZE_LIGHT_CULLING"
	class	FLightingChannelsDim            : SHADER_PERMUTATION_BOOL("USE_LIGHTING_CHANNELS"
	class	FTransmissionDim                :   SHADER_PERMUTATION_BOOL("USE_TRANSMISSION"); :
	class	FHairLighting                     SHADER_PERMUTATION_INT("USE_HAIR_LIGHTING",          3);
	class	FAtmosphereTransmittance:
SHADER_PERMUTATION_BOOL("USE_ATMOSPHERE_TRANSMITTANCE");

	using FPermutationDomain = TShaderPermutationDomain<
		FSourceShapeDim,
		FSourceTextureDim,
		FIESProfileDim,
		FInverseSquaredDim,
		FVisualizeCullingDim,
		FLightingChannelsDim,
		FTransmissionDim,
		FHairLighting,
		FAtmosphereTransmittance>;


	//reduceshadercombination of.
	static bool ShouldCompilePermutation(constFGlobalShaderPermutationParameters& Parameters)

	{
		FPermutationDomainPermutationVector(Parameters.PermutationId);

		if(PermutationVector.Get< FSourceShapeDim >() == ELightSourceShape::Directional
&&(
			PermutationVector.Get< FIESProfileDim >() ||
			PermutationVector.Get< FInverseSquaredDim >() ) )
		{
			returnfalse;
		}

		if(PermutationVector.Get< FSourceShapeDim >() != ELightSourceShape::Directional
&&		PermutationVector.Get<FAtmosphereTransmittance>())
		{
			returnfalse;
		}

		if(PermutationVector.Get< FSourceShapeDim >() == ELightSourceShape::Rect ) {

			if(		!PermutationVector.Get< FInverseSquaredDim >() )
			{
				returnfalse;
			}
		}
		else
		{
			if(PermutationVector.Get< FSourceTextureDim >() ) {
```

```
                returnfalse;
            }
        }

        if(PermutationVector.Get<FHairLighting>()  &&  !
        IsHairStrandsSupported(Parameters.Platform))
        {
            returnfalse;
        }

        if(PermutationVector.Get< FHairLighting >() ==2&& (
            PermutationVector.Get< FVisualizeCullingDim >() ||
            PermutationVector.Get< FTransmissionDim >()))
        {
            returnfalse;
        }

        returnIsFeatureLevelSupported(Parameters.Platform, ERHIFeatureLevel::SM5);
    }

    FDeferredLightPSconst :        ShaderMetaType::CompiledShaderInitializerType&                Initializer
        FGlobalShader(Initializer)
    {
        //Bindingshaderparameter. SceneTextureParameters.Bind(Initializer);
        LightAttenuationTexture.Bind(Initializer.ParameterMap, TEXT(
        "LightAttenuationTexture"));

        LightAttenuationTextureSampler.Bind(Initializer.ParameterMap, TEXT(
"LightAttenuationTextureSampler"));
        LTCMatTexture.Bind(Initializer.ParameterMap, LTCMatSampleTrE.XBTin("dL(TInCiMtialtiTzerx.tPuarrea"m));e TtEeXrMT(ap,
        LTCAmpTexture.Bind(Initializer.ParameterMap, LTCAmpSamp"lLeTrC.BMinadtS(lanmitipalleizre")r).;P TaErXaTm(eterMap,
        IESTexture.Bind(Initializer.ParameterMap, TEXT("IESTexture")");L TCAmpTexture")); TEXT(
        IESTextureSampler.Bind(Initializer.ParameterMap, TEXT("IEST"eLxTtCuAremSpaSmapmleprl"e)r);" ));
        LightingChannelsTexture.Bind(Initializer.ParameterMap,


TEXT("LightingChannelsTexture"));
        LightingChannelsSampler.Bind(Initializer.ParameterMap, TEXT(
"LightingChannelsSampler"));
        TransmissionProfilesTexture.Bind(Initializer.ParameterMap, TEXT(
"SSProfilesTexture"));
        TransmissionProfilesLinearSampler.Bind(Initializer.ParameterMap, TEXT(
"TransmissionProfilesLinearSampler"));

        HairTransmittanceBuffer.Bind(Initializer.ParameterMap, TEXT(
"HairTransmittanceBuffer"));
        HairTransmittanceBufferMaxCount.Bind(Initializer.ParameterMap, TEXT(
"HairTransmittanceBufferMaxCount"));
        ScreenShadowMaskSubPixelTexture.Bind(Initializer.ParameterMap, TEXT(
"ScreenShadowMaskSubPixelTexture"));// TODO hook the shader itself

        HairLUTTexture.Bind(Initializer.ParameterMap,          TEXT("HairLUTTexture")); TEXT(
        HairLUTSampler.Bind(Initializer.ParameterMap,          "HairLUTSampler")); TEXT(
        HairComponents.Bind(Initializer.ParameterMap,          "HairComponents"));
        HairShadowMaskValid.Bind(Initializer.ParameterMap,              TEXT("HairShadowMaskValid"));
        HairDualScatteringRoughnessOverride.Bind(Initializer.ParameterMap, TEXT(
"HairDualScatteringRoughnessOverride"));
```