```
            HairCategorizationTexture.Bind(Initializer.ParameterMap, TEXT(
"HairCategorizationTexture"));
            HairVisibilityNodeOffsetAndCount.Bind(Initializer.ParameterMap, TEXT(
"HairVisibilityNodeOffsetAndCount"));
            HairVisibilityNodeCoords.Bind(Initializer.ParameterMap, TEXT(
"HairVisibilityNodeCoords"));
            HairVisibilityNodeData.Bind(Initializer.ParameterMap, TEXT(
"HairVisibilityNodeData"));
        }

        FDeferredLightPS()
        {}

public:
        //Set the light source parameters.
        void SetParameters(
            FRHICommandList& RHICmdList,
            const FSceneView&       View,
            const    FLightSceneInfo*       LightSceneInfo,
            IPooledRenderTarget*       ScreenShadowMaskTexture,
            FRenderLightParams*       RenderLightParams)
        {
            FRHIPixelShader* ShaderRHI = RHICmdList.GetBoundPixelShader(); //Set the basic parameters
            of the light source.
            SetParametersBase(RHICmdList, ShaderRHI, View, ScreenShadowMaskTexture, LightSceneInfo-
>Proxy->GetIESTextureResource(),                                RenderLightParams);
            //Sets the parameters of a delayed light.
            SetDeferredLightParameters(RHICmdList,                ShaderRHI,
            GetUniformBufferParameter<FDeferredLightUniformStruct>(),       LightSceneInfo, View);
        }

        //Set simple light parameters.
        void SetParametersSimpleLight(FRHICommandList& RHICmdList,constFSceneView& View, const
FSimpleLightEntry& SimpleLight,constFSimpleLightPerViewEntry& SimpleLightPerViewData)

        {
            FRHIPixelShader* ShaderRHI = RHICmdList.GetBoundPixelShader(); SetParametersBase(RHICmdList,
            ShaderRHI, View, nullptr, nullptr, nullptr); SetSimpleDeferredLightParameters(RHICmdList,
            ShaderRHI,
GetUniformBufferParameter<FDeferredLightUniformStruct>(), SimpleLight,
SimpleLightPerViewData, View);
        }

private:
        void   SetParametersBase(
            FRHICommandList& RHICmdList,
            FRHIPixelShader*       ShaderRHI,
            constFSceneView&       View,
            IPooledRenderTarget*       ScreenShadowMaskTexture,
            FTexture* IESTextureResource,
            FRenderLightParams* RenderLightParams)
        {
            FGlobalShader::SetParameters<FViewUniformShaderParameters>(RHICmdList,
            ShaderRHI,View.ViewUniformBuffer);
            SceneTextureParameters.Set(RHICmdList, ShaderRHI, View.FeatureLevel,
            ESceneTextureSetupMode::All);

            FSceneRenderTargets& SceneRenderTargets = FSceneRenderTargets::Get(RHICmdList);
```

```
            //Light source attenuation graph.
            if(LightAttenuationTexture.IsBound()) {

                SetTexturePameter(
                    RHICmdList,
                    ShaderRHI,
                    LightAttenuationTexture, LightAttenuationTextureSampler,
                    TStaticSamplerState<SF_Point,AM_Wrap,AM_Wrap,AM_Wrap>::GetRHI(),
                    ScreenShadowMaskTexture? ScreenShadowMaskTexture-

> GetRenderTargetItem().ShaderResourceTexture : GWhiteTexture->TextureRHI
                    ) ;
            }

            //Area LightLCTTexture.
            SetTexturePameter(
                RHICmdList,
                ShaderRHI,
                LTCMatTexture,
                LTCMatSampler,
                TStaticSamplerState<SF_Bilinear,AM_Clamp,AM_Clamp,AM_Clamp>::GetRHI(),
                GSystemTextures.LTCMat->GetRenderTargetItem().ShaderResourceTexture );


            SetTexturePameter(
                RHICmdList,
                ShaderRHI,
                LTCAmpTexture,
                LTCAmpSampler,
                TStaticSamplerState<SF_Bilinear,AM_Clamp,AM_Clamp,AM_Clamp>::GetRHI(),
                GSystemTextures.LTCAmp->GetRenderTargetItem().ShaderResourceTexture );



            {
                FRHITexture* TextureRHI = IESTextureResource ? IESTextureResource->TextureRHI
: GSystemTextures.WhiteDummy->GetRenderTargetItem().TargetableTexture;

                SetTexturePameter(
                    RHICmdList,          ShaderRHI,          IESTexture,          IESTextureSampler,
                    TStaticSamplerState<SF_Bilinear,AM_Clamp,AM_Clamp,AM_Clamp>::GetRHI(), TextureRHI




                    ) ;
            }

            //Light channel texture.
            if( LightingChannelsTexture.IsBound() ) {

                FRHITexture* LightingChannelsTextureRHI = SceneRenderTargets.LightingChannels
? SceneRenderTargets.LightingChannels->GetRenderTargetItem().ShaderResourceTexture:
GSystemTextures.WhiteDummy->GetRenderTargetItem().TargetableTexture;

                SetTexturePameter(
                    RHICmdList,
                    ShaderRHI,
```

```
                LightingChannelsTexture, LightingChannelsSampler,
                TStaticSamplerState<SF_Point,AM_Clamp,AM_Clamp,AM_Clamp>::GetRHI(),
                LightingChannelsTextureRHI

                ) ;
        }

        if( TransmissionProfilesTexture.IsBound() ) {

                FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList); const
                IPooledRenderTarget* PooledRT =
GetSubsufaceProfileTexture_RT((FRHICommandListImmediate&)RHICmdList);

                if(!PooledRT)
                {
                        // no subsurface profile was used yet PooledRT =
                        GSystemTextures.BlackDummy;
                }

                constFSceneRenderTargetItem& Item = PooledRT->GetRenderTargetItem();

                SetTextureParameter(RHICmdList,
                        ShaderRHI,
                        TransmissionProfilesTexture,
                        TransmissionProfilesLinearSampler,
                        TStaticSamplerState<SF_Bilinear, AM_Clamp, AM_Clamp, AM_Clamp>::GetRHI(),
                        Item.ShaderResourceTexture);
        }

        if(HairTransmittanceBuffer.IsBound()) {

                constuint32 TransmittanceBufferMaxCount = RenderLightParams?
RenderLightParams->DeepShadow_TransmittanceMaskBufferMaxCount:0;
                SetShaderValue(
                        RHICmdList,
                        ShaderRHI,
                        HairTransmittanceBufferMaxCount,
                        TransmittanceBufferMaxCount);
                if(RenderLightParams && RenderLightParams-
> DeepShadow_TransmittanceMaskBuffer)
                {
                        SetSRVParameter(RHICmdList, ShaderRHI, HairTransmittanceBuffer,
RenderLightParams->DeepShadow_TransmittanceMaskBuffer);
                }
        }

        if(ScreenShadowMaskSubPixelTexture.IsBound()) {

                if(RenderLightParams)
                {
                        SetTextureParameter(
RHICmdList,
ShaderRHI,
ScreenShadowMaskSubPixelTexture,
LightAttenuationTextureSampler,
TStaticSamplerState<SF_Point, AM_Clamp, AM_Clamp, AM_Clamp>::GetRHI(),
(RenderLightParams && RenderLightParams-
> ScreenShadowMaskSubPixelTexture) ? RenderLightParams->ScreenShadowMaskSubPixelTexture-
```

```
> GetRenderTargetItem().ShaderResourceTexture : GWhiteTexture->TextureRHI);

                uint32 InHairShadowMaskValid = RenderLightParams-
> ScreenShadowMaskSubPixelTexture?1:0;
                SetShaderValue(
                    RHICmdList,
                    ShaderRHI,
                    HairShadowMaskValid,
                    InHairShadowMaskValid);
            }
        }

        if(HairCategorizationTexture.IsBound()) {

            if(RenderLightParams && RenderLightParams->HairCategorizationTexture) {

                SetTextureParameter(
                    RHICmdList,
                    ShaderRHI,
                    HairCategorizationTexture,
                    LightAttenuationTextureSampler,
                    TStaticSamplerState<SF_Point, AM_Clamp, AM_Clamp, AM_Clamp>::GetRHI(),
                    RenderLightParams->HairCategorizationTexture-
> GetRenderTargetItem().TargetableTexture);
            }
        }

        if(HairVisibilityNodeOffsetAndCount.IsBound()) {

            if(RenderLightParams && RenderLightParams->HairVisibilityNodeOffsetAndCount) {

                SetTextureParameter(
                    RHICmdList,
                    ShaderRHI,
                    HairVisibilityNodeOffsetAndCount,
                    LightAttenuationTextureSampler,
                    TStaticSamplerState<SF_Point, AM_Clamp, AM_Clamp, AM_Clamp>::GetRHI(),
                    RenderLightParams->HairVisibilityNodeOffsetAndCount-
> GetRenderTargetItem().TargetableTexture);
            }
        }

        if(HairVisibilityNodeCoords.IsBound()) {

            if(RenderLightParams && RenderLightParams->HairVisibilityNodeCoordsSRV) {

                FShaderResourceViewRHIRef SRV = RenderLightParams-
> HairVisibilityNodeCoordsSRV;
                SetSRVParameter(
                    RHICmdList,
                    ShaderRHI,
                    HairVisibilityNodeCoords, SRV);

            }
        }

        if(HairVisibilityNodeData.IsBound()) {
```

```
                if(RenderLightParams && RenderLightParams->HairVisibilityNodeDataSRV) {

                    FShaderResourceViewRHIRef SRV = RenderLightParams-
> HairVisibilityNodeDataSRV;
                    SetSRVParameter(
                        RHICmdList,
                        ShaderRHI,
                        HairVisibilityNodeData, SRV);

                }
            }

            if(HairLUTTexture.IsBound()) {

                IPooledRenderTarget* HairLUTTextureResource = GSystemTextures.HairLUT0;
                SetTextureParameter(
                    RHICmdList,
                    ShaderRHI,
                    HairLUTTexture,
                    HairLUTSampler,
                    TStaticSamplerState<SF_Bilinear, AM_Clamp, AM_Clamp, AM_Clamp>::GetRHI(),
                    HairLUTTextureResource ? HairLUTTextureResource-
> GetRenderTargetItem().ShaderResourceTexture : GBlackVolumeTexture->TextureRHI); }


            if(HairComponents.IsBound()) {

                uint32 InHairComponents = ToBitfield(GetHairComponents()); SetShaderValue(

                    RHICmdList,
                    ShaderRHI,
                    HairComponents,
                    InHairComponents);
            }

            if(HairDualScatteringRoughnessOverride.IsBound()) {

                const floatDualScatteringRoughness =
GetHairDualScatteringRoughnessOverride();
                SetShaderValue(
                    RHICmdList,
                    ShaderRHI,
                    HairDualScatteringRoughnessOverride,
                    DualScatteringRoughness);
            }

        }

        LAYOUT_FIELD(FSceneTextureShaderParameters,              SceneTextureParameters);
        LAYOUT_FIELD(FShaderResourceParameter,          LightAttenuationTexture);
        LAYOUT_FIELD(FShaderResourceParameter,          LightAttenuationTextureSampler);
        LAYOUT_FIELD(FShaderResourceParameter,          LTCMatTexture);   LTCMatSampler);
        LAYOUT_FIELD(FShaderResourceParameter,          LTCAmpTexture);
        LAYOUT_FIELD(FShaderResourceParameter,          LTCAmpSampler);       IESTexture);
        LAYOUT_FIELD(FShaderResourceParameter,          IESTextureSampler);
        LAYOUT_FIELD(FShaderResourceParameter,          LightingChannelsTexture);
        LAYOUT_FIELD(FShaderResourceParameter,
        LAYOUT_FIELD(FShaderResourceParameter,
```

```
    LAYOUT_FIELD(FShaderResourceParameter,          LightingChannelsSampler);
    LAYOUT_FIELD(FShaderResourceParameter,          TransmissionProfilesTexture);
    LAYOUT_FIELD(FShaderResourceParameter,          TransmissionProfilesLinearSampler);

    LAYOUT_FIELD(FShaderParameter, HairTransmittanceBufferMaxCount);
    LAYOUT_FIELD(FShaderResourceParameter,          HairTransmittanceBuffer);
    LAYOUT_FIELD(FShaderResourceParameter,          HairCategorizationTexture);
    LAYOUT_FIELD(FShaderResourceParameter,          HairVisibilityNodeOffsetAndCount);
    LAYOUT_FIELD(FShaderResourceParameter,          HairVisibilityNodeCoords);
    LAYOUT_FIELD(FShaderResourceParameter,          HairVisibilityNodeData);
    LAYOUT_FIELD(FShaderResourceParameter,          ScreenShadowMaskSubPixelTexture);

    LAYOUT_FIELD(FShaderResourceParameter,          HairLUTTexture);
    LAYOUT_FIELD(FShaderResourceParameter,          HairLUTSampler);
    LAYOUT_FIELD(FShaderParameter,          HairComponents);
    LAYOUT_FIELD(FShaderParameter,          HairShadowMaskValid);
    LAYOUT_FIELD(FShaderParameter,          HairDualScatteringRoughnessOverride);
};
```

5.5.3 LightingPass Shader

The previous article has pointed out the nested relationship between scenes, light sources, views, and shader calls:

```
foreach(scene in scenes) {

    foreach(light in          lights)
    {
        foreach(view      in views)
        {
            RenderLight();      // Executed once each time a render light is calledDeferredLightVertexShadersand
DeferredLightPixelShadersThe code.
        }
    }
}
```

This means that the number of times DeferredLightVertexShader and DeferredLightPixelShader are executed is:

$$Nscandnand Nlight_{viandin} N$$

This also indirectly illustrates the necessity of sorting light sources and shadows, which can reduce the data exchange 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 light sources.

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

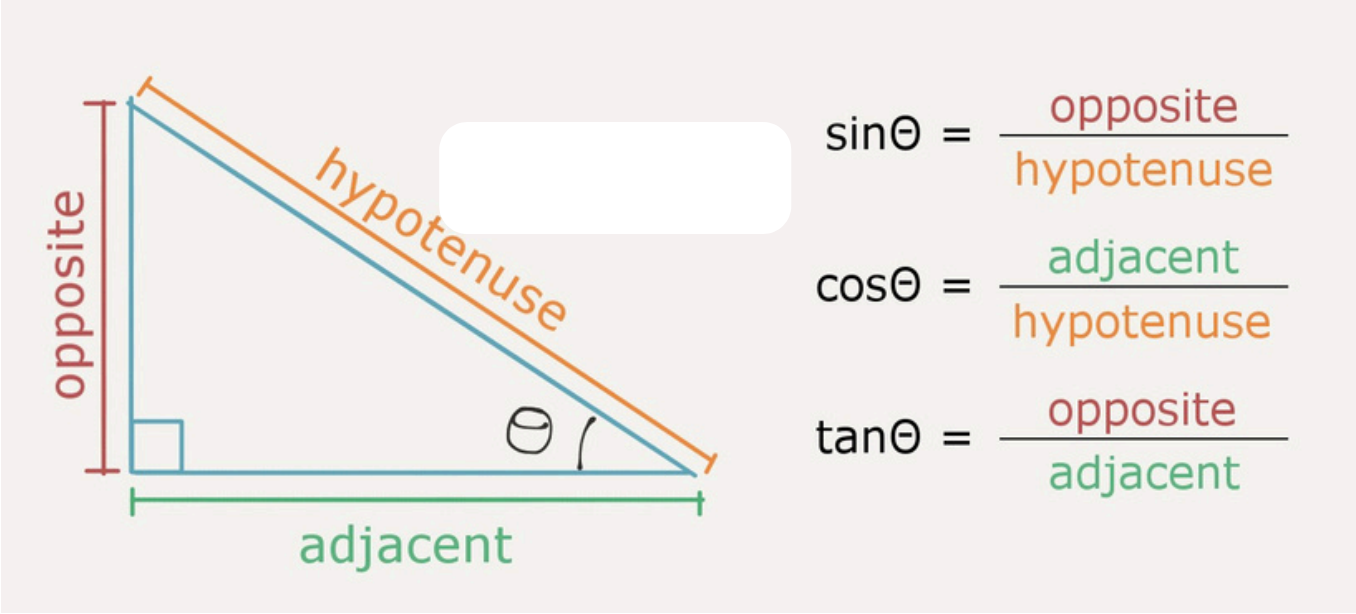
5.5.3.1 DeferredLightVertexShader

The entry of DeferredLightVertexShader is in DeferredLightVertexShaders.usf:

```
#include"Common.ush"

#ifdefined(SHADER_RADIAL_LIGHT)
float4 StencilingGeometryPosAndScale;
float4 StencilingConeParameters;                    // .x NumSides (0 if not cone), .y NumSlices, .z
ConeAngle,      . w ConeSphereRadius
float4x4        StencilingConeTransform;
float3          StencilingPreViewTranslation;
#endif

#ifdefined(SHADER_RADIAL_LIGHT) && SHADER_RADIAL_LIGHT == 0

//Rendering parallel light using full screen cubeVS.
void DirectionalVertexMain(
        in float2 InPosition : ATTRIBUTE0, in float2
        InUV                                : ATTRIBUTE1,
        out float2 OutTexCoord : TEXCOORD0, out float3
        OutScreenVector : TEXCOORD1, out float4
        OutPosition : SV_POSITION )

{
        //Draw a rectangle.
        DrawRectangle(float4(InPosition.xy,0,1), InUV, OutPosition, OutTexCoord); //Convert the output
        position to a screen vector.
        OutScreenVector = mul(float4(OutPosition.xy,1,0), View.ScreenToTranslatedWorld).xyz;
}
#endif

#ifFEATURE_LEVEL >= FEATURE_LEVEL_SM4 && SHADER_RADIAL_LIGHT == 1 //Draws a point
light or spotlight using approximate bounding geometryVS. void RadialVertexMain(

        in uint InVertexId : SV_VertexID, in float3 InPosition :
        ATTRIBUTE0, out float4 OutScreenPosition :
        TEXCOORD0, out float4 OutPosition : SV_POSITION )



{
        float3 WorldPosition;
        uint NumSides = StencilingConeParameters.x; //Cone
        shape
        if(NumSides !=0) {

                floatSphereRadius = StencilingConeParameters.w; float
                ConeAngle = StencilingConeParameters.z;

                //Cone vertex shading.
                const floatInvCosRadiansPerSide =1.0f/cos(PI / (float)NumSides); //useCos(Theta)=Adjacent edge(
                Adjacent)/hypotenuse(HypotenuseThe formula for finding the end of the cone along the coneZThe
                distance of the axes. const floatZRadius = SphereRadius *cos(ConeAngle); const floatTanConeAngle =
                tan(ConeAngle);

                uint NumSlices = StencilingConeParameters.y; uint
                CapIndexStart = NumSides * NumSlices; //Generate cone
                vertices
                if(InVertexId < CapIndexStart) {

                        uint SliceIndex = InVertexId / NumSides;
```

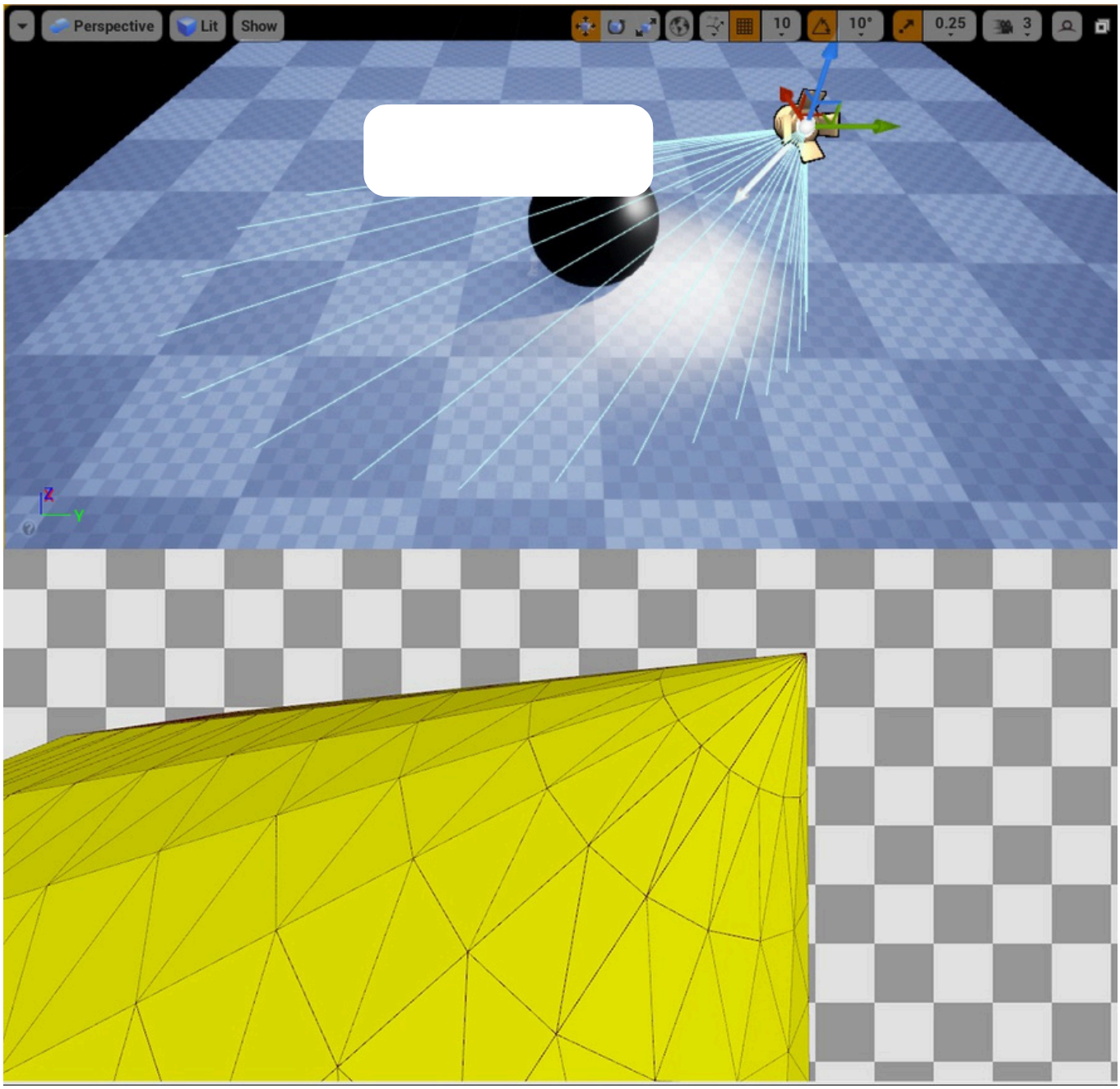
```
                uint SideIndex = InVertexId % NumSides;

                const floatCurrentAngle = SideIndex *2* PI / (float)NumSides; const float
                DistanceDownConeDirection = ZRadius * SliceIndex / (float)
(NumSlices  -  1);
                //  useTan(Theta)=Opposite side(Opposite)/Adjacent edge(Adjacent)to solve for the radius of this slice using the formula. Raise the
                //  effective radius so that the edge of the circle lies on the cone, replacing the vertex.
                const floatSliceRadius = DistanceDownConeDirection * TanConeAngle *
InvCosRadiansPerSide;
                //Create a position in the local space of the cone, atXYA circle is formed on the plane and alongZAxis offset.
                constfloat3 LocalPosition = float3(ZRadius * SliceIndex / (float)(NumSlices -
1), SliceRadius *sin(CurrentAngle), SliceRadius *cos(CurrentAngle));
                //Convert to world space and applypre-view translation, because these vertices will be associated with a deletedpre-view
translationUsed with shaders.
                WorldPosition = mul(float4(LocalPosition,1), StencilingConeTransform).xyz +
StencilingPreViewTranslation;
            }
            else
            {
                //Generate vertices for the ball cap.
                const floatCapRadius = ZRadius *tan(ConeAngle);

                uint VertexId = InVertexId - CapIndexStart; uint SliceIndex
                = VertexId / NumSides; uint SideIndex = VertexId %
                NumSides;

                const floatUnadjustedSliceRadius = CapRadius * SliceIndex / (float)(NumSlices
-  1);
                //Increase the effective radius so that the edge of the circle lies on the cone, instead of the vertex.
                const floatSliceRadius = UnadjustedSliceRadius * InvCosRadiansPerSide; //Using the
                Pythagorean theorem (Pythagorean theoremFind the value of this sliceZAxle distance. const
                floatZDistance =sqrt(SphereRadius * SphereRadius -
UnadjustedSliceRadius * UnadjustedSliceRadius);

                const floatCurrentAngle = SideIndex *2* PI / (float)NumSides; constfloat3 LocalPosition
                = float3(ZDistance, SliceRadius *
sin(CurrentAngle), SliceRadius *cos(CurrentAngle));
                WorldPosition = mul(float4(LocalPosition,1), StencilingConeTransform).xyz +
StencilingPreViewTranslation;
            }
        }
        else//spherical.
        {
            WorldPosition = InPosition * StencilingGeometryPosAndScale.w +
            StencilingGeometryPosAndScale.xyz;
        }

        OutScreenPosition = OutPosition = mul(float4(WorldPosition,1),
        View.TranslatedWorldToClip);
}
#endif

(.....)
```

The triangle composition theorem and various trigonometric function definitions are used above, as shown below:



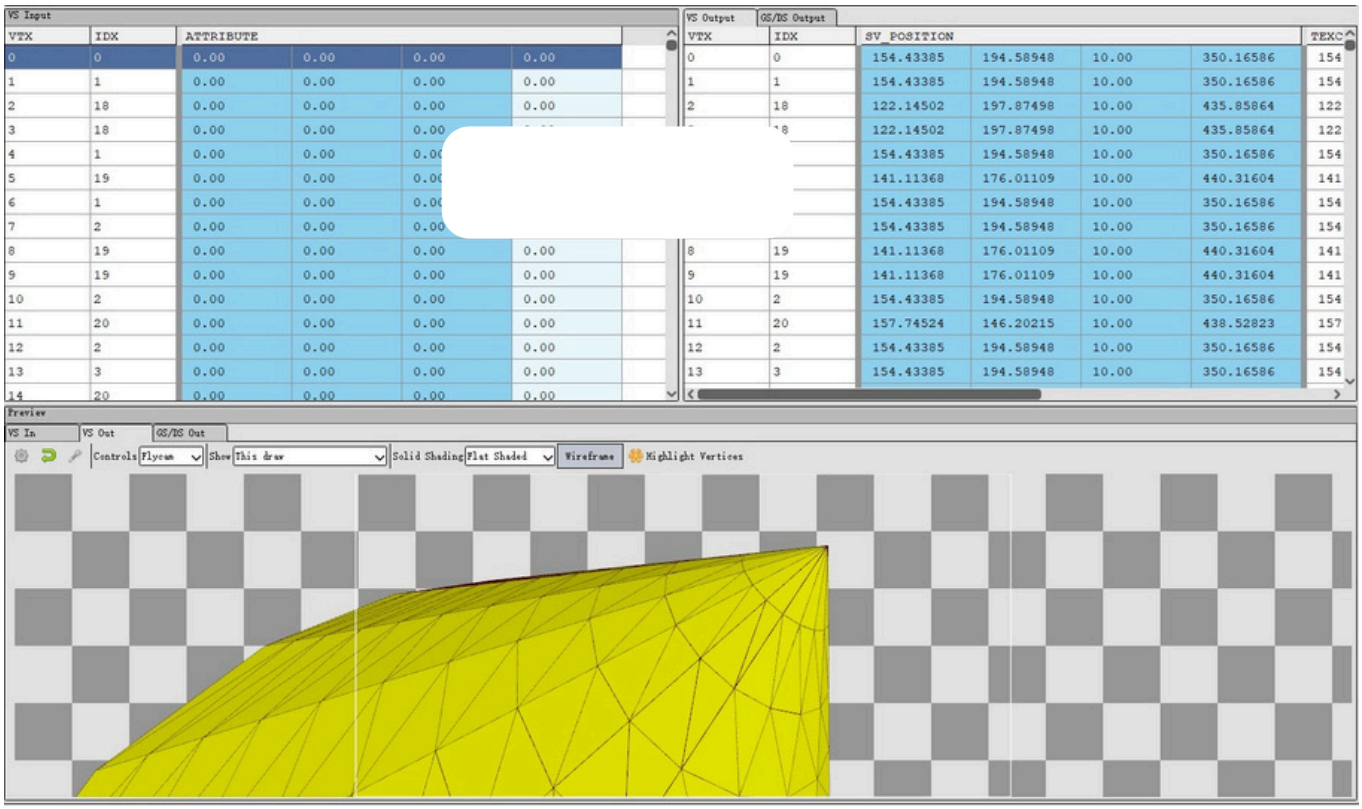
For parallel light, since it affects the surface of objects in the entire scene, it is directly drawn as a full-screen block.

Point light sources and spotlights require special processing. They are drawn using spheres and cones respectively to remove pixels outside the influence of the light source. Therefore, their vertex processing is more complicated than parallel light.



The spotlight in the scene uses cone drawing lighting in the VS stage of deferred lighting.

It should be noted that for point light sources or spotlights, the vertex data of VS only has vertex IDs but no vertex data. The output vertex data is generated by the vertex shader:



Note that the ATTRIBUTE values of the VS Input in the upper left corner are all 0, while the VS Output in the upper right corner has a normal value.

5.5.3.2 DeferredLightPixelShader

The entrance to DeferredLightPixelShader is in DeferredLightPixelShaders.usf:

```
#defineSUPPORT_CONTACT_SHADOWS 1

#include    "Common.ush"
#include    "DeferredShadingCommon.ush"
#include    "DeferredLightingCommon.ush"

(.....)

#ifUSE_ATMOSPHERE_TRANSMITTANCE
#include"/Engine/Private/SkyAtmosphereCommon.ush"
#endif

//Input parameters.
structFInputParams
{
    float2    PixelPos;
    float4    ScreenPosition;
    float2    ScreenUV;
    float3    ScreenVector;
};

//Derived parameters.
structFDerivedParams
{
    float3    CameraVector;
    float3    WorldPosition;
};
```

```
//Get derived parameters.
FDerivedParamsGetDerivedParams(in FInputParams Input, infloatSceneDepth) {

    FDerivedParams Out;
#ifLIGHT_SOURCE_SHAPE > 0
    // With a perspective projection, the clip space position is NDC * Clip.w // With an orthographic
    projection, clip space is the same as NDC float2 ClipPosition = Input.ScreenPosition.xy /
    Input.ScreenPosition.w * (View.ViewToClip[3][3] <1.0f? SceneDepth:1.0f);

    Out.WorldPosition = mul(float4(ClipPosition, SceneDepth,1), View.ScreenToWorld).xyz; Out.CameraVector =
    normalize(Out.WorldPosition - View.WorldCameraOrigin);
#else
    Out.WorldPosition = Input.ScreenVector * SceneDepth + View.WorldCameraOrigin; Out.CameraVector =
    normalize(Input.ScreenVector);
#endif
    returnOut;
}

//Create and set up the deferred lighting data structureFDeferredLightData.
FDeferredLightDataSetupLightDataForStandardDeferred() {

    FDeferredLightData LightData;

    LightData.Position          =DeferredLightUniforms.Position;
    LightData.InvRadius          =DeferredLightUniforms.InvRadius;
    LightData.Color = DeferredLightUniforms.Color;
    LightData.FalloffExponent              = DeferredLightUniforms.FalloffExponent; =
    LightData.Direction          DeferredLightUniforms.Direction;
    LightData.Tangent =          DeferredLightUniforms.Tangent;
    LightData.SpotAngles          =DeferredLightUniforms.SpotAngles; =
    LightData.SourceRadius              DeferredLightUniforms.SourceRadius;
    LightData.SourceLength            = DeferredLightUniforms.SourceLength;
    LightData.SoftSourceRadius              = DeferredLightUniforms.SoftSourceRadius;
    LightData.SpecularScale =             DeferredLightUniforms.SpecularScale;
    LightData.ContactShadowLength
    =abs(DeferredLightUniforms.ContactShadowLength);
    LightData.ContactShadowLengthInWS = DeferredLightUniforms.ContactShadowLength <0.0f;
    LightData.DistanceFadeMAD = DeferredLightUniforms.DistanceFadeMAD;
    LightData.ShadowMapChannelMask = DeferredLightUniforms.ShadowMapChannelMask;
    LightData.ShadowedBits = DeferredLightUniforms.ShadowedBits;

    LightData.bInverseSquared = INVERSE_SQUARED_FALLOFF;
    LightData.bRadialLight = LIGHT_SOURCE_SHAPE >0;
    LightData.bSpotLight = LIGHT_SOURCE_SHAPE >0;
    LightData.bRectLight = LIGHT_SOURCE_SHAPE ==2;

    LightData.RectLightBarnCosAngle              = DeferredLightUniforms.RectLightBarnCosAngle;
    LightData.RectLightBarnLength =              DeferredLightUniforms.RectLightBarnLength;

    LightData.HairTransmittance = InitHairTransmittanceData(); returnLightData;

}

//Light channel texture.
Texture2D<uint> LightingChannelsTexture; //Get the
light channel mask.
uintGetLightingChannelMask(float2 UV) {
```

```
    uint2 IntegerUV = UV * View.BufferSizeAndInvSize.xy; return
    LightingChannelsTexture.Load(uint3(IntegerUV,                    0)).x;
}

float GetExposure()
{
#ifUSE_PREEXPOSURE
    return    View.PreExposure;
#else
    return    1;
#endif
}

(.....)


#ifUSE_HAIR_LIGHTING == 0 || USE_HAIR_LIGHTING == 1

//Deferred light pixel shader main entry point.
void DeferredLightPixelMain(
#ifLIGHT_SOURCE_SHAPE >0
    float4 InScreenPosition : TEXCOORD0,
#else
    float2    ScreenUV                :  TEXCOORD0,
    float3    ScreenVector            :  TEXCOORD1,
# endif
    float4    SVP                  :  SV_POSITION,
    out float4    OutColor            : SV_Target0
    )
{
    constfloat2 PixelPos = SVPos.xy; OutColor =0;


    // Convert input data (directional/local light) FInputParams
    InputParams = (FInputParams)0; InputParams.PixelPos
                                    = SVPos.xy;
#ifLIGHT_SOURCE_SHAPE > 0
    InputParams.ScreenPosition //Calculating      = InScreenPosition;
    screens for special light sourcesUV.
    InputParams.ScreenUV            = InScreenPosition.xy / InScreenPosition.w *
 View.ScreenPositionScaleBias.xy    +  View.ScreenPositionScaleBias.wz; 0;
     InputParams.ScreenVector       =
#else
    InputParams.ScreenPosition          =0;
    InputParams.ScreenUV            =  ScreenUV;
    InputParams.ScreenVector        =  ScreenVector;
#endif

    //Get screen space data, includingGBufferandAO.
    FScreenSpaceData ScreenSpaceData = GetScreenSpaceData(InputParams.ScreenUV);

    //onlyShadingModelIDNot for0Pixels with deferred shading.
    BRANCHif( ScreenSpaceData.GBuffer.ShadingModelID >0
        //Check if the light channels overlap.
        #ifUSE_LIGHTING_CHANNELS
        && (GetLightingChannelMask(InputParams.ScreenUV) &
        DeferredLightUniforms.LightingChannelMask)
        # endif
```

```
        )
    {
        const floatOpaqueVisibility =1.0f;

        //Calculates scene depth value.
        const floatSceneDepth = CalcSceneDepth(InputParams.ScreenUV); //Get derived
        data.
        constFDerivedParams DerivedParams = GetDerivedParams(InputParams, SceneDepth);

        //Sets deferred light data.
        FDeferredLightData LightData = SetupLightDataForStandardDeferred();

        //Get the jitter.
        floatDither = InterleavedGradientNoise(InputParams.PixelPos,
        View.StateFrameIndexMod8 );

        //Gets a rectangle texture from the light's source texture.
        FRectTexture RectTexture = InitRectTexture(DeferredLightUniforms.SourceTexture); floatSurfaceShadow =
        1.0f; //Calculate dynamic lighting.

        constfloat4 Radiance = GetDynamicLighting(DerivedParams.WorldPosition,
DerivedParams.CameraVector, ScreenSpaceData.GBuffer,          ScreenSpaceData.AmbientOcclusion,
ScreenSpaceData.GBuffer.ShadingModelID, LightData,
GetPerPixelLightAttenuation(InputParams.ScreenUV), RectTexture, Dither,     uint2(InputParams.PixelPos),
SurfaceShadow);
        //Calculates additional attenuation coefficient for a light configuration.
        const floatAttenuation =
        ComputeLightProfileMultiplier(DerivedParams.WorldPosition,     DeferredLightUniforms.Position,
- DeferredLightUniforms.Direction, DeferredLightUniforms.Tangent);

        //Calculate the final color.
        OutColor += (Radiance * Attenuation) * OpaqueVisibility;

        //Atmospheric transmission.
    #if  USE_ATMOSPHERE_TRANSMITTANCE
        OutColor.rgb *= GetAtmosphericLightTransmittance(SVPos, InputParams.ScreenUV,
        DeferredLightUniforms.Direction.xyz);
    #endif
    }

    // RGB:SceneColor Specular and Diffuse // A:Non
    Specular SceneColor Luminance
    // So we need PreExposure for both color and alpha OutColor.rgba
    *= GetExposure();
}
#endif

(.....)
```

The above is the pixel shading process of delayed light source. Looks simple? No, there are two important interfaces that contain a lot of logic.

The first one is relatively simpleGetScreenSpaceData,tracing its logic stack:

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

```
//Get screen-space data.
FScreenSpaceDataGetScreenSpaceData(float2 UV,boolbGetNormalizedNormal =true) {

    FScreenSpaceData Out;

    //GetGBufferdata.
    Out.GBuffer = GetGBufferData(UV, bGetNormalizedNormal); //Get screen
    spaceAO.
    float4 ScreenSpaceAO = Texture2DSampleLevel(SceneTexturesStruct.ScreenSpaceAOTexture,
SceneTexturesStruct.ScreenSpaceAOTextureSampler, UV,0);

    //NoticeAOOnly takeraisle. Out.AmbientOcclusion =
    ScreenSpaceAO.r;

    returnOut;
}

//Get the specified screen spaceUVofGBufferdata.
FGBufferDataGetGBufferData(float2 UV,boolbGetNormalizedNormal =true) {

    (.....)

    //fromGBufferSample data from a texture.
    float4 GBufferA = Texture2DSampleLevel(SceneTexturesStruct.GBufferATexture,
SceneTexturesStruct.GBufferATextureSampler, UV,0);
    float4 GBufferB = Texture2DSampleLevel(SceneTexturesStruct.GBufferBTexture,
SceneTexturesStruct.GBufferBTextureSampler, UV,0);
    float4 GBufferC = Texture2DSampleLevel(SceneTexturesStruct.GBufferCTexture,
SceneTexturesStruct.GBufferCTextureSampler, UV,0);
    float4 GBufferD = Texture2DSampleLevel(SceneTexturesStruct.GBufferDTexture,
SceneTexturesStruct.GBufferDTextureSampler, UV,0);
    floatCustomNativeDepth = Texture2DSampleLevel(SceneTexturesStruct.CustomDepthTexture,
SceneTexturesStruct.CustomDepthTextureSampler, UV,0).r;

    int2 IntUV = (int2)trunc(UV * View.BufferSizeAndInvSize.xy); //Custom template
    values.
    uint CustomStencil = SceneTexturesStruct.CustomStencilTexture.Load(int3(IntUV,0))
    STENCIL_COMPONENT_SWIZZLE;

    (.....)

    //Static light.
    #if ALLOW_STATIC_LIGHTING
        float4 GBufferE = Texture2DSampleLevel(SceneTexturesStruct.GBufferETexture,
SceneTexturesStruct.GBufferETextureSampler, UV,0);
    #else
        float4 GBufferE =1;
    #endif

    //Tangent.
    #if GBUFFER_HAS_TANGENT
        float4 GBufferF = Texture2DSampleLevel(SceneTexturesStruct.GBufferFTexture,
SceneTexturesStruct.GBufferFTextureSampler, UV,0);
    #else
        float4 GBufferF =0.5f;
    #endif

    //speed.
```

```
	#if	WRITES_VELOCITY_TO_GBUFFER float4 GBufferVelocity =
		Texture2DSampleLevel(SceneTexturesStruct.GBufferVelocityTexture,
SceneTexturesStruct.GBufferVelocityTextureSampler, UV,0);

	#else
		float4 GBufferVelocity =0;
	#endif
#endif

	//depth.
	floatSceneDepth = CalcSceneDepth(UV);

	//Decode texture values into corresponding structures.
	returnDecodeGBufferData(GBufferA, GBufferB, GBufferC, GBufferD, GBufferE, GBufferF, GBufferVelocity,
CustomNativeDepth, CustomStencil, SceneDepth, bGetNormalizedNormal, CheckerFromSceneColorUV(UV));

}

//Decode texture values into corresponding structures.
FGBufferDataDecodeGBufferData(float4 InGBufferA, float4 InGBufferB, float4 InGBufferC, float4 InGBufferD, float4
InGBufferE, float4 InGBufferF, float4 InGBufferVelocity,float CustomNativeDepth, uint CustomStencil,floatSceneDepth,
boolbGetNormalizedNormal,bool bChecker)

{
	FGBufferData GBuffer;

	//Normal.
	GBuffer.WorldNormal = DecodeNormal(InGBufferA.xyz); if
	(bGetNormalizedNormal) {

		GBuffer.WorldNormal = normalize(GBuffer.WorldNormal);
	}

	//Per object, metalness, specularity,
	roughness. GBuffer.PerObjectGBufferData= InGBufferA.a;
	GBuffer.Metallic		=	InGBufferB.r;
	GBuffer.Specular		=	InGBufferB.g;
	GBuffer.Roughness		=	InGBufferB.b;
	//Shading ModelID.
	GBuffer.ShadingModelID		= DecodeShadingModelId(InGBufferB.a);
	GBuffer.SelectiveOutputMask		= DecodeSelectiveOutputMask(InGBufferB.a);

	//Basic color.
	GBuffer.BaseColor = DecodeBaseColor(InGBufferC.rgb);

	//AOand indirect light.
#if	ALLOW_STATIC_LIGHTING
	GBuffer.GBufferAO =1;
	GBuffer.IndirectIrradiance		=	DecodeIndirectIrradiance(InGBufferC.a);
#else
	GBuffer.GBufferAO = InGBufferC.a;
	GBuffer.IndirectIrradiance =1;
#endif

	//Custom data.
	GBuffer.CustomData = !(GBuffer.SelectiveOutputMask & SKIP_CUSTOMDATA_MASK) ? InGBufferD :0;
```

```
        //Scene or custom depth stencil.
        GBuffer.PrecomputedShadowFactors = !(GBuffer.SelectiveOutputMask &
SKIP_PRECSHADOW_MASK) ? InGBufferE :              ((GBuffer.SelectiveOutputMask &
ZERO_PRECSHADOW_MASK)?0:          1);
        GBuffer.CustomDepth =       ConvertFromDeviceZ(CustomNativeDepth); =
        GBuffer.CustomStencil       CustomStencil;
        GBuffer.Depth = SceneDepth;

        //Keep the initial base value.
        GBuffer.StoredBaseColor         =  GBuffer.BaseColor;
        GBuffer.StoredMetallic          =  GBuffer.Metallic;
        GBuffer.StoredSpecular          =  GBuffer.Specular;

        (......)

        //Derived fromBaseColor, Metalness, Specularof data. {

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


            if(UseSubsurfaceProfile(GBuffer.ShadingModelID)) {

                AdjustBaseColorAndSpecularColorForSubsurfaceProfileLighting(GBuffer.BaseColor,
GBuffer.SpecularColor, GBuffer.Specular, bChecker);
            }

            GBuffer.DiffuseColor = GBuffer.BaseColor - GBuffer.BaseColor * GBuffer.Metallic;

            (......)
        }

        //Tangent.
#if     GBUFFER_HAS_TANGENT
        GBuffer.WorldTangent        = DecodeNormal(InGBufferF.rgb);
        GBuffer.Anisotropy = InGBufferF.a *2.0f-1.0f;

        if(bGetNormalizedNormal) {

            GBuffer.WorldTangent        =  normalize(GBuffer.WorldTangent);
        }
#else
        GBuffer.WorldTangent        =0;
        GBuffer.Anisotropy =        0;
#endif

        //speed.
        GBuffer.Velocity = !(GBuffer.SelectiveOutputMask & SKIP_VELOCITY_MASK) ? InGBufferVelocity:0;


        returnGBuffer;
}
```

The process of decoding GBuffer is to sample data from GBuffer texture, perform secondary processing and store it in FGBufferData, and then store the instance of FGBufferData in FScreenSpaceData, so that it can be directly accessed in subsequent lighting calculations, and it can

also prevent multiple texture sampling from causing IO bottlenecks between GPU and video memory and reducing GPU Cache hit rate.

The first one is very complexGetDynamicLightingand goes into complex lighting calculation logic:

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

//Calculate dynamic lighting.
float4GetDynamicLighting(
	float3 WorldPosition, float3 CameraVector, FGBufferData GBuffer,float AmbientOcclusion, uint
	ShadingModelID,
	FDeferredLightData LightData, float4 LightAttenuation,floatDither, uint2 SVPos, FRectTexture SourceTexture,

	inoutfloatSurfaceShadow)
{
	FDeferredLightingSplit SplitLighting = GetDynamicLightingSplit(
		WorldPosition, CameraVector, GBuffer, AmbientOcclusion, ShadingModelID, LightData,
		LightAttenuation, Dither, SVPos, SourceTexture, SurfaceShadow);


	returnSplitLighting.SpecularLighting + SplitLighting.DiffuseLighting;
}

//Calculates dynamic lighting, separating diffuse and specular terms.
FDeferredLightingSplitGetDynamicLightingSplit(
	float3 WorldPosition, float3 CameraVector, FGBufferData GBuffer,float AmbientOcclusion, uint
	ShadingModelID,
	FDeferredLightData LightData, float4 LightAttenuation,floatDither, uint2 SVPos, FRectTexture SourceTexture,

	inoutfloatSurfaceShadow)
{
	FLightAccumulator LightAccumulator = (FLightAccumulator)0;

	float3 V = -CameraVector; float3 N =
	GBuffer.WorldNormal;
	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);
		N = OctahedronToUnitVector(oct1);
	}

	float3 L = LightData.Direction; float3		// Already normalized
	ToLight = L;

	floatLightMask =1; //
		Gets the attenuation of a radiant light source.
	if	(LightData.bRadialLight)
	{
		LightMask = GetLocalLightAttenuation( WorldPosition, LightData, ToLight, L );
	}

	LightAccumulator.EstimatedCost +=0.3f;				// running the PixelShader at all has a
cost

	BRANCH
```

```
//Only surfaces with significant intensities are calculated.
if( LightMask >0) {

    //Handles surface shadows.
    FShadowTerms Shadow;
    Shadow.SurfaceShadow      = AmbientOcclusion;
    Shadow.TransmissionShadow      =1;
    Shadow.TransmissionThickness          =1;
    Shadow.HairTransmittance.Transmittance                =1;
    Shadow.HairTransmittance.OpaqueVisibility //              =1;
    Computes surface shading data.
    GetShadowTerms(GBuffer, LightData, WorldPosition, L, LightAttenuation, Dither,
Shadow);
    SurfaceShadow = Shadow.SurfaceShadow;

    LightAccumulator.EstimatedCost +=0.3f; shadow                    // add the cost of getting the
    terms

    BRANCH
    //If the sum of the intensity of light after shadow and the intensity of transmission >0Then continue to calculate the lighting.
    if( Shadow.SurfaceShadow + Shadow.TransmissionShadow >0) {

        const boolbNeedsSeparateSubsurfaceLightAccumulation =
UseSubsurfaceProfile(GBuffer.ShadingModelID);
        float3 LightColor = LightData.Color;

    #ifNON_DIRECTIONAL_DIRECT_LIGHTING//Non-parallel direct light
        floatLighting;

        if( LightData.bRectLight )//Rectangular Light {

            FRect Rect = GetRect(ToLight, LightData);

            //Integral rectangular lighting.
            Lighting = IntegrateLight(Rect, SourceTexture);
        }
        else    //Capsule Light
        {
            FCapsuleLight Capsule = GetCapsule( ToLight, LightData );

            //Integral capsule lighting.
            Lighting = IntegrateLight( Capsule, LightData.bInverseSquared );
        }

        float3 LightingDiffuse = Diffuse_Lambert( GBuffer.DiffuseColor ) * Lighting;
        LightAccumulator_AddSplit(LightAccumulator, LightingDiffuse,0.0f,0,
LightColor * LightMask * Shadow.SurfaceShadow, bNeedsSeparateSubsurfaceLightAccumulation);
    #else//Parallel direct light
        FDirectLighting Lighting;

        if(LightData.bRectLight) {

            FRect Rect = GetRect(ToLight, LightData);

            #if    REFERENCE_QUALITY
                Lighting = IntegrateBxDF( GBuffer, N, V, Rect, Shadow, SourceTexture,
SVP      ) ;
            #else
```

```
                    Lighting = IntegrateBxDF(GBuffer, N, V, Rect, Shadow, SourceTexture);
                #endif
            }
            else
            {
                FCapsuleLight Capsule = GetCapsule( ToLight, LightData );

                #if  REFERENCE_QUALITY
                    Lighting = IntegrateBxDF( GBuffer, N, V, Capsule, Shadow, SVPos );
                #else
                    Lighting = IntegrateBxDF( GBuffer, N, V, Capsule, Shadow,
LightData.bInverseSquared );
                #endif
            }

            Lighting.Specular *= LightData.SpecularScale;

            LightAccumulator_AddSplit(LightAccumulator, Lighting.Diffuse,
Lighting.Specular, Lighting.Diffuse, LightColor * LightMask * Shadow.SurfaceShadow,
bNeedsSeparateSubsurfaceLightAccumulation );
            LightAccumulator_AddSplit(LightAccumulator, Lighting.Transmission,0.0f,
Lighting.Transmission, LightColor * LightMask * Shadow.TransmissionShadow,
bNeedsSeparateSubsurfaceLightAccumulation );

            LightAccumulator.EstimatedCost +=0.4f;                    // add the cost of the lighting
computations (should sum up to 1 form one light)
        #endif
        }
    }

    returnLightAccumulator_GetResultSplit(LightAccumulator);
}
```

In the above code, under the branch where NON_DIRECTIONAL_DIRECT_LIGHTING is 0, four different BxDF interfaces will be entered depending on whether it is rectangular light and quality reference (REFERENCE_QUALITY). The following is an analysis of the IntegrateBxDF version of non-rectangular light and non-quality reference:

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

//Integral capsule lighting.
FDirectLightingIntegrateBxDF(FGBufferData GBuffer, half3 N, half3 V, FCapsuleLight Capsule, FShadowTerms
Shadow,boolbInverseSquared ) {

    float    NoL;
    float    Falloff;
    floatLineCosSubtended =1;

    // Clip to horizon
    //float NoP0 = dot( N, Capsule.LightPos[0] ); //float NoP1 =
    dot( N, Capsule.LightPos[1] );
    //if( NoP0 < 0 ) Capsule.LightPos[0] = ( Capsule.LightPos[0] * NoP1 - Capsule.LightPos[1] *
NoP0 ) / ( NoP1 - NoP0);
    //if( NoP1 < 0 ) Capsule.LightPos[1] = ( -Capsule.LightPos[0] * NoP1 + Capsule.LightPos[1] *
NoP0 ) / (-NoP1 + NoP0);
```

```
    BRANCH
    //Processing attenuation,NandLThe dot product of .
    if( Capsule.Length >0)//If it is a valid capsule, only line segment integration is required (see the specific formula for5.4.2.2) {


        LineIrradiance( N, Capsule.LightPos[0], Capsule.LightPos[1], Capsule.DistBiasSqr, LineCosSubtended, Falloff,
        NoL );
    }
    else//The light source is considered as a point
    {
        floatDistSqr = dot( Capsule.LightPos[0], Capsule.LightPos[0] ); Falloff = rcp( DistSqr +
        Capsule.DistBiasSqr );

        float3 L = Capsule.LightPos[0] * rsqrt( DistSqr ); NoL = dot( N, L );

    }

    //Capsule Radius>0,When the ball cap is adjustedNandLThe
    dot product of . if( Capsule.Radius >0) {

        // TODO Use capsule area?
        floatSinAlphaSqr = saturate( Pow2( Capsule.Radius ) * Falloff ); NoL =
        SphereHorizonCosWrap( NoL, SinAlphaSqr );
    }

    NoL = saturate( NoL );
    Falloff = bInverseSquared ? Falloff :1;


    //Adjust the effective length of the light source direction.
    float3 ToLight = Capsule.LightPos[0]; if
    ( Capsule.Length >0) {

        float3 R = reflect( -V, N );

        ToLight = ClosestPointLineToRay( Capsule.LightPos[0], Capsule.LightPos[1], Capsule.Length, R );

    }

    floatDistSqr = dot(ToLight, ToLight); floatInvDist =
    rsqrt( DistSqr ); float3 L = ToLight * InvDist;


    GBuffer.Roughness = max( GBuffer.Roughness, View.MinRoughness ); floata =
    Pow2(GBuffer.Roughness);


    //Construct area light information based on the above information.
    FAreaLight AreaLight;
    AreaLight.SphereSinAlpha = saturate( Capsule.Radius * InvDist * (1- a) );
    AreaLight.SphereSinAlphaSoft = saturate( Capsule.SoftRadius * InvDist );
    AreaLight.LineCosSubtended          = LineCosSubtended;
    AreaLight.FalloffColor              =1;
    AreaLight.Rect =          (FRect)0;
    AreaLight.bIsRect          = false;
    AreaLight.Texture          = InitRectTexture(DummyRectLightTextureForCapsuleCompilerWarning);


    //Integral Area Light
    returnIntegrateBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow );
}
```

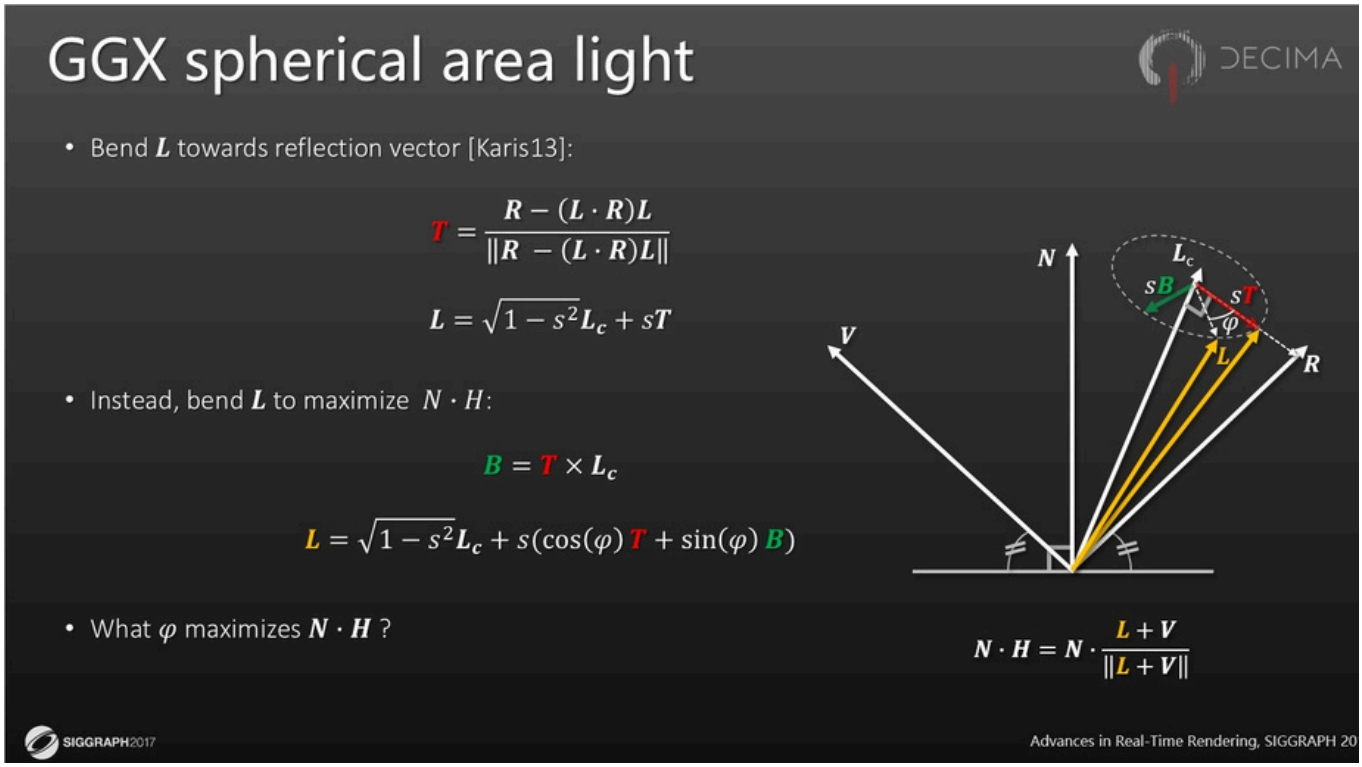
```
// Engine\Shaders\Private\ShadingModels.ush

FDirectLightingIntegrateBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L,float Falloff,floatNoL, FAreaLight
AreaLight, FShadowTerms Shadow ) {

    //Enter different lighting according to different shading modelsBxDF.
    switch(    GBuffer.ShadingModelID )
    {
        case   SHADINGMODELID_DEFAULT_LIT:
        case   SHADINGMODELID_SINGLELAYERWATER:
        case   SHADINGMODELID_THIN_TRANSLUCENT: //
            Default Lighting
            returnDefaultLitBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow );
        case SHADINGMODELID_SUBSURFACE:
            //Subsurface scattering
            returnSubsurfaceBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow );
            SHADINGMODELID_PREINTEGRATED_SKIN:
        case //Pre-integrated skin

            returnPreintegratedSkinBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight,
Shadow   ) ;
        case   SHADINGMODELID_CLEAR_COAT: //
            Varnish
            returnClearCoatBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow );
        case SHADINGMODELID_SUBSURFACE_PROFILE:
            //Subsurface scattering configuration

            returnSubsurfaceProfileBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight,
Shadow   ) ;
        case   SHADINGMODELID_TWOSIDED_FOLIAGE: //
            Double-Sided
            returnTwoSidedBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow ); SHADINGMODELID_HAIR:
        case
            //hair
            returnHairBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow );
        case SHADINGMODELID_CLOTH:
            //Fabric
            returnClothBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow ); SHADINGMODELID_EYE:
        case
            //Eye
            returnEyeBxDF( GBuffer, N, V, L, Falloff, NoL, AreaLight, Shadow ); default:

            return(FDirectLighting)0;
    }
}
```

From the above code, we can see that in the logic of the integrated area light, different shading BxDF interfaces will be entered according to the shading model of the surface. The most common ones are analyzed belowDefaultLitBxDF:

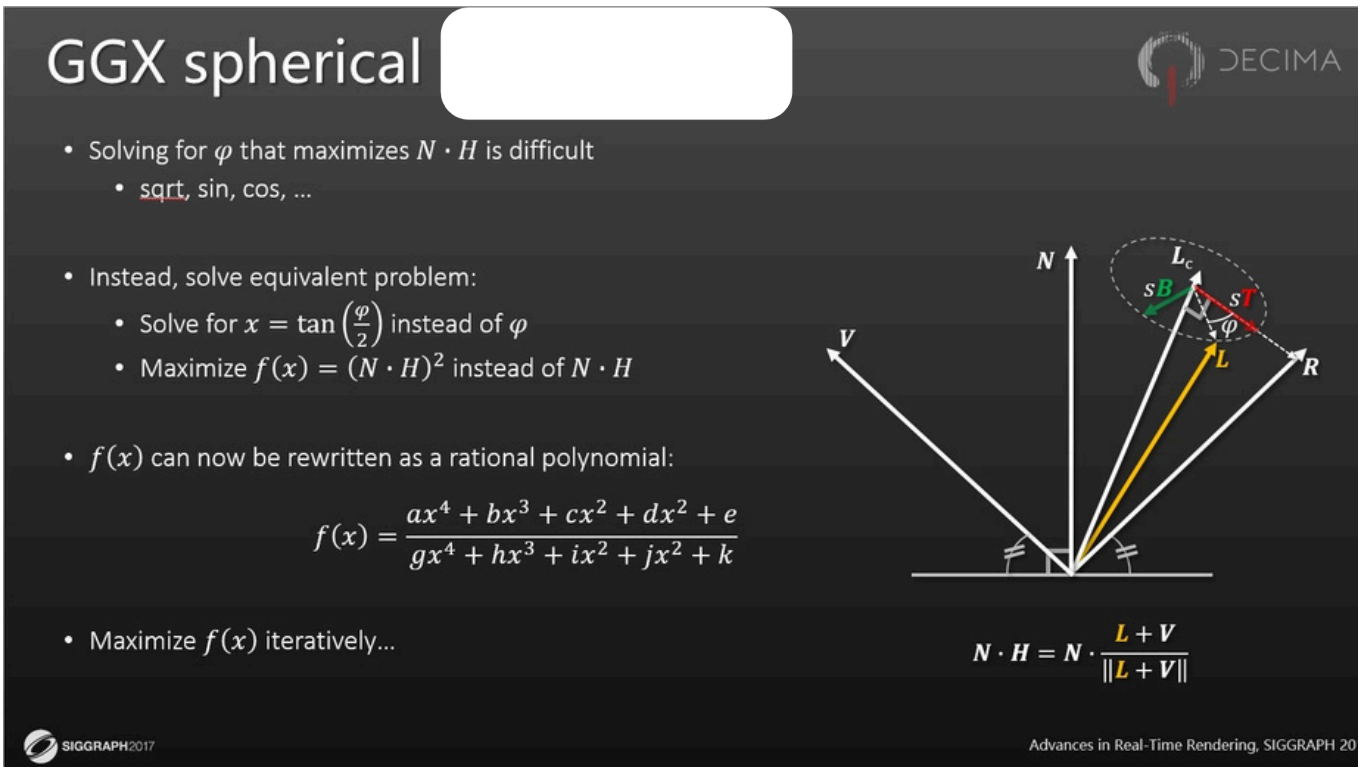
```
// Engine\Shaders\Private\ShadingModels.ush

//Default lighting modelBxDF.
FDirectLightingDefaultLitBxDF(FGBufferData GBuffer, half3 N, half3 V, half3 L,float Falloff,floatNoL, FAreaLight
AreaLight, FShadowTerms Shadow ) {

#ifGBUFFER_HAS_TANGENT
    half3 X = GBuffer.WorldTangent;
```

```
        half3 Y = normalize(cross(N, X));
#else
        half3 X =0;
        half3 Y =0;
#endif

        //Initial context, containing various dot products.
        BxDFContext Context;
        Init( Context, N, X, Y, V, L );
        SphereMaxNoH(Context, AreaLight.SphereSinAlpha,true); Context.NoV =
        saturate(abs( Context.NoV ) +1e-5);

        FDirectLighting Lighting; //Computes diffuse
        reflection of direct light using Lambert.
        Lighting.Diffuse             = AreaLight.FalloffColor * (Falloff * NoL) * Diffuse_Lambert(
GBuffer.DiffuseColor          ) ;

        //Use rectangularGGXApproximateLTCComputes specular reflections of direct light. if(
                AreaLight.bIsRect )
                Lighting.Specular = RectGGXApproxLTC( GBuffer.Roughness, GBuffer.SpecularColor, N,
V, AreaLight.Rect, AreaLight.Texture ); else

                Lighting.Specular = AreaLight.FalloffColor * (Falloff * NoL) * SpecularGGX( GBuffer.Roughness,
GBuffer.Anisotropy, GBuffer.SpecularColor, Context, NoL, AreaLight );

        Lighting.Transmission =0; return
        Lighting;
}
```

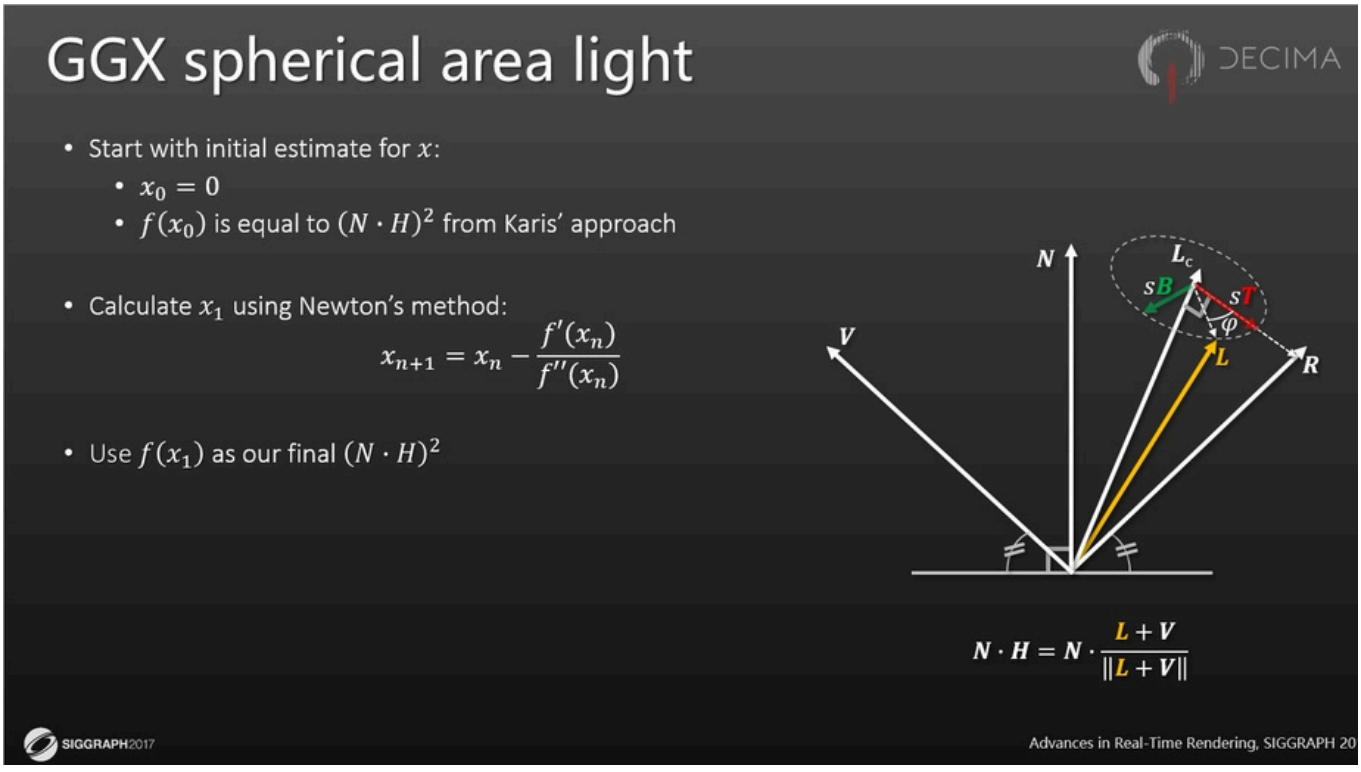
SphereMaxNoHIt is the dot product between various vectors after adjusting the simulated area light. It is proposed by the paperDECIMA ENGINE: ADVANCES IN LIGHTING AND AA. Its core idea is to first lengthen and adjust the tangent, bitangent, and light source direction:



Directly analyze to maximize$N \cdot H$It is more difficult, but we can assume some conditions and then pro
a rational polynomial simulation$(N \cdot H)2$:



First use the Karis approximation to estimate the first term, then use the Newton iteration method to find the second term, and then use it to simulate$(N \cdot H)2$:



SphereMaxNoHThe corresponding implementation code:

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

void SphereMaxNoH( inout BxDFContext Context,floatSinAlpha,boolbNewtonIteration ) {
```

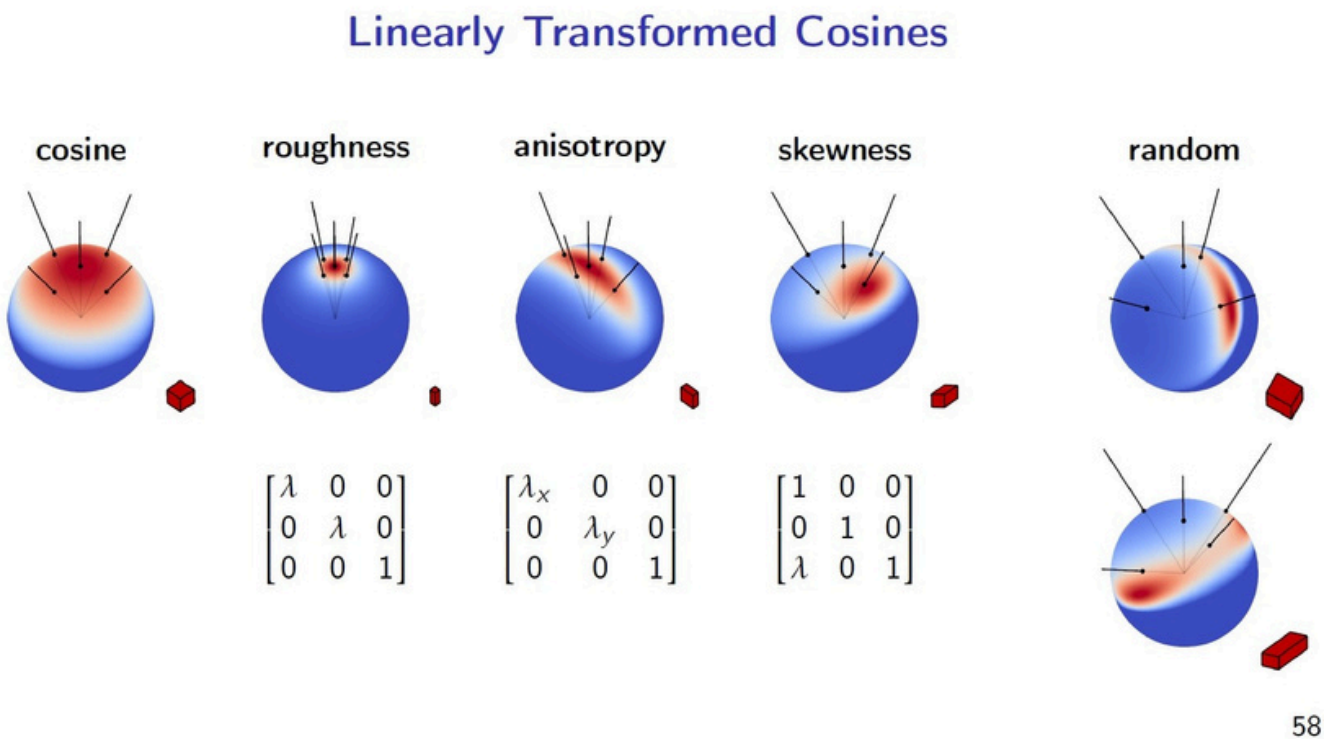
```
    if( SinAlpha >0) {

        floatCosAlpha =sqrt(1- Pow2( SinAlpha ) );

        floatRoL =2* Context.NoL * Context.NoV - Context.VoL; if( RoL >= CosAlpha )
        {

            Context.NoH    =  1;
            Context.XoH    =  0;
            Context.YoH    = 0;
            Context.VoH =abs( Context.NoV );
        }
        else
        {
            floatrInvLengthT = SinAlpha * rsqrt(1- RoL*RoL ); floatNoTr = rInvLengthT * ( Context.NoV -
            RoL * Context.NoL ); floatVoTr = rInvLengthT * (2* Context.NoV*Context.NoV -1-RoL*

Context.VoL );

            if(bNewtonIteration)
            {
                // dot( cross(N,L), V )
                floatNxLoV =sqrt( saturate(1- Pow2(Context.NoL) - Pow2(Context.NoV) -
Pow2(Context.VoL) +2* Context.NoL * Context.NoV * Context.VoL ) );

                floatNoBr = rInvLengthT * NxLoV;
                floatVoBr = rInvLengthT * NxLoV *2* Context.NoV;

                floatNoLVTr = Context.NoL * CosAlpha + Context.NoV + NoTr; floatVoLVTr =
                Context.VoL * CosAlpha +1                                    + VoTr;

                floatp = NoBr              * VoLVTr;
                floatq = NoLVTr * VoLVTr; floats =
                VoBr * NoLVTr;

                floatxNum = q * (-0.5* p +0.25* VoBr * NoLVTr );
                floatxDenom = p*p + s * (s -2*p) + NoLVTr * ( (Context.NoL * CosAlpha +
Context.NoV) * Pow2(VoLVTr) + q * (-0.5* (VoLVTr + Context.VoL * CosAlpha) -0.5) );
                floatTwoX1 =2* xNum / ( Pow2(xDenom) + Pow2(xNum) ); floatSinTheta =
                TwoX1 * xDenom; floatCosTheta =1.0- TwoX1 * xNum; NoTr = CosTheta *
                NoTr + SinTheta * NoBr; VoTr = CosTheta * VoTr + SinTheta * VoBr;



            }

            Context.NoL = Context.NoL * CosAlpha + NoTr;// dot(N, L * CosAlpha + T *
SinAlpha    )
            Context.VoL = Context.VoL * CosAlpha + VoTr;

            floatInvLenH = rsqrt(2+2* Context.VoL );
            Context.NoH = saturate( ( Context.NoL + Context.NoV ) * InvLenH ); Context.VoH =
            saturate( InvLenH + InvLenH * Context.VoL );
        }
    }
}
```

After adjusting the dot product of various surface vectors, the diffuse reflection is calculated using Lambert, and the specular reflection of direct light is calculated using LTC of the rectangular GGX approximation. The implementation code of the latter is as follows:

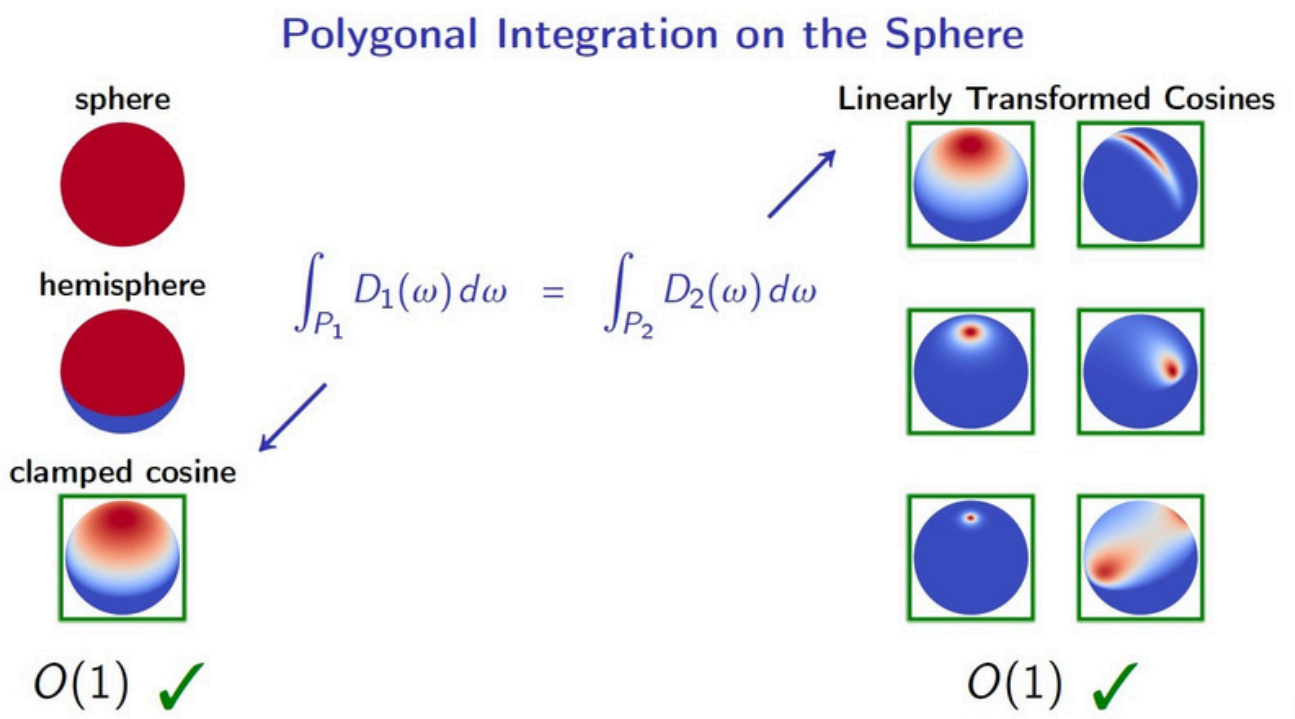
```
// Engine\Shaders\Private\RectLight.ush

float3RectGGXApproxLTC(floatRoughness, float3 SpecularColor, half3 N, float3 V, FRect Rect, FRectTexture RectTexture )

{
	// No visible rect light due to barn door occlusion if(Rect.Extent.x ==0||
	Rect.Extent.y ==0)return0;

	floatNoV = saturate(abs( dot(N, V) ) +1e-5);

	float2 UV = float2(Roughness,sqrt(1- NoV ) ); UV = UV * (63.0/64.0)
	+ (0.5/64.0);

	float4 LTCMat = LTCMatTexture.SampleLevel( LTCMatSampler, UV,0); float4 LTCAmp =
	LTCAmpTexture.SampleLevel( LTCAmpSampler, UV,0);

	float3x3 LTC = {
		float3(LTCMat.x,0, LTCMat.z ), float3(
		0,1, 0),
		float3(LTCMat.y,0, LTCMat.w )
	};

	floatLTCDet = LTCMat.x * LTCMat.w - LTCMat.y * LTCMat.z;

	float4 InvLTCMat = LTCMat / LTCDet; float3x3
	InvLTC = {
		float3( InvLTCMat.w,0,-InvLTCMat.z ), float3(
		0,1, 0 ) ,
		float3(-InvLTCMat.y,0, InvLTCMat.x )
	};

	// Rotate to tangent space
	float3 T1 = normalize( V - N * dot( N, V ) ); float3 T2 =
	cross( N, T1 );
	float3x3 TangentBasis = float3x3( T1, T2, N );

	LTC = mul( LTC, TangentBasis );
	InvLTC = mul( transpose( TangentBasis ), InvLTC );

	float3 Poly[4];
	Poly[0] = mul( LTC, Rect.Origin - Rect.Axis[0] * Rect.Extent.x - Rect.Axis[1] * Rect.Extent.y );

	Poly[1] = mul( LTC, Rect.Origin + Rect.Axis[0] * Rect.Extent.x - Rect.Axis[1] * Rect.Extent.y );

	Poly[2] = mul( LTC, Rect.Origin + Rect.Axis[0] * Rect.Extent.x + Rect.Axis[1] * Rect.Extent.y );

	Poly[3] = mul( LTC, Rect.Origin - Rect.Axis[0] * Rect.Extent.x + Rect.Axis[1] * Rect.Extent.y );


	// Vector irradiance
	float3 L = PolygonIrradiance( Poly );
```

```
        floatLengthSqr = dot( L, L ); floatInvLength =
        rsqrt( LengthSqr ); floatLength = LengthSqr *
        InvLength;

        // Mean light direction L *=
        InvLength;

        // Solid angle of sphere // Cosine                = 2*PI * ( 1 - sqrt(1 - r^2 / d^2 ) )
        weighted integration // SinAlphaSqr =             = PI * r^2 / d^2
        r^2 / d^2; floatSinAlphaSqr = Length;


        floatNoL = SphereHorizonCosWrap( Lz, SinAlphaSqr ); floatIrradiance =
        SinAlphaSqr * NoL;

        // Kill negative and NaN Irradiance = -min(-
        Irradiance,0.0);

        SpecularColor = LTCAmp.y + ( LTCAmp.x - LTCAmp.y ) * SpecularColor;

        // Transform to world space L =
        mul( InvLTC, L );

        float3 LightColor = SampleSourceTexture( L, Rect, RectTexture );

        returnLightColor * Irradiance * SpecularColor;
}
```

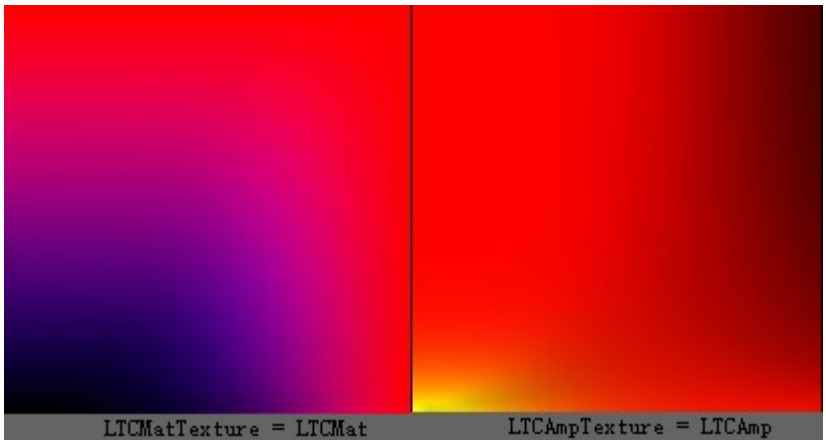
The above implementation comes from the paper[Real-Time Polygonal-Light Shading with Linearly Transformed Cosines]() . The core idea is to linearly pre-transform the roughness, anisotropy, and bevel characteristics of the BRDF of a surface light source of an arbitrary shape:



And it is as efficient as the sphere, hemisphere, and clamped cosine lighting models, and faster than the Feng lighting model:



Then the most time-consuming LTC matrix and scaling are pre-integrated into the textures, LTCMatTexture and LTCAmpTexture respectively:



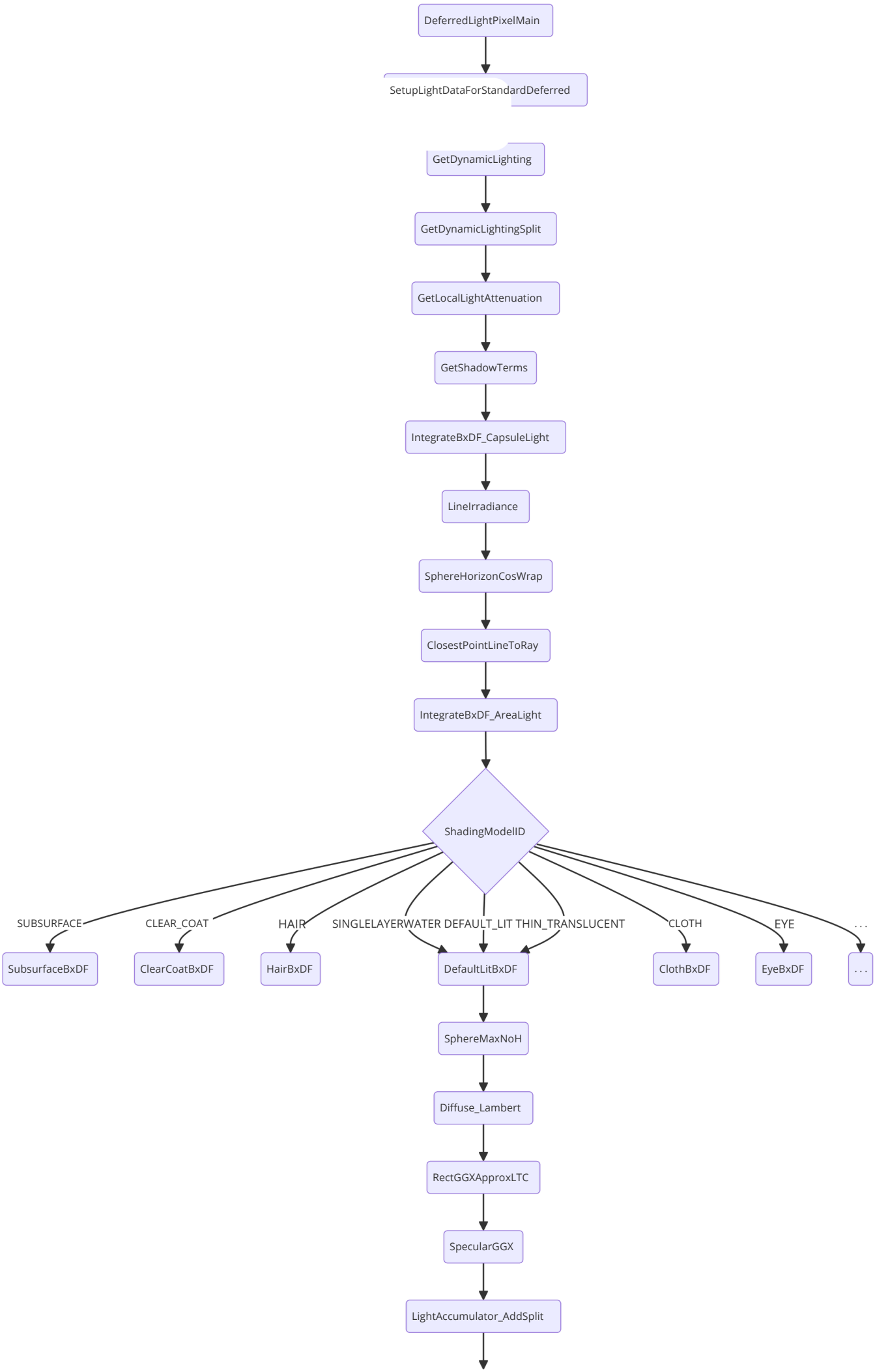
However, the above two textures are calculated at runtime when the engine is initialized, and then cached and used directly:

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

void FSystemTextures::InitializeFeatureLevelDependentTextures(FRHICommandListImmediate& RHICmdList,const
ERHIFeatureLevel::Type InFeatureLevel) {

	(......)

	//LTCMatTexture
	{
		FPooledRenderTargetDesc
Desc(FPooledRenderTargetDesc::Create2DDesc(FIntPoint(LTC_Size, FClearValueBindinLgT::CN_oSnize?)F_FloatRGBA,
TexCreate_FastVRAM, TexCreate_ShaderResource,false));
		Desc.AutoWritable =false;

		GRenderTargetPool.FindFreeElement(RHICmdList, Desc, LTCMat, TEXT("LTCMat"));
```

```
            // Write the contents of the texture. uint32
            DestStride;
            uint8* DestBuffer = (uint8*)RHICmdList.LockTexture2D((FTexture2DRHIRef&)LTCMat-
> GetRenderTargetItem().ShaderResourceTexture,0, RLM_WriteOnly, DestStride,false);

            for(int32 y =0; y < Desc.Extent.Y; ++y)
            {
                  for(int32 x =0; x < Desc.Extent.X; ++x) {

                        uint16* Dest = (uint16*)(DestBuffer + x *4*sizeof(uint16) + y *
DestStride);

                        for(intk =0; k <4; k++)
                              Dest[k] = FFloat16(LTC_Mat[4* (x + y * LTC_Size) + k]).Encoded;
                  }
            }
            RHICmdList.UnlockTexture2D((FTexture2DRHIRef&)LTCMat-
> GetRenderTargetItem().ShaderResourceTexture,0,false); }


      // LTCAmpTexture
      {
            FPooledRenderTargetDesc
Desc(FPooledRenderTargetDesc::Create2DDesc(FIntPoint(LTC_Size, FClearValueBindinLgT::CN_oSnize,PF_G16R16F,
TexCreate_FastVRAM, TexCreate_ShaderResource,false));
            Desc.AutoWritable =false;

            GRenderTargetPool.FindFreeElement(RHICmdList, Desc, LTCAmp, TEXT("LTCAmp")); // Write the
            contents of the texture. uint32 DestStride;

            uint8* DestBuffer = (uint8*)RHICmdList.LockTexture2D((FTexture2DRHIRef&)LTCAmp-
> GetRenderTargetItem().ShaderResourceTexture,0, RLM_WriteOnly, DestStride,false);

            for(int32 y =0; y < Desc.Extent.Y; ++y) {

                  for(int32 x =0; x < Desc.Extent.X; ++x) {

                        uint16* Dest = (uint16*)(DestBuffer + x *2*sizeof(uint16) + y *
DestStride);

                        for(intk =0; k <2; k++)
                              Dest[k] = FFloat16(LTC_Amp[4* (x + y * LTC_Size) + k]).Encoded;
                  }
            }
            RHICmdList.UnlockTexture2D((FTexture2DRHIRef&)LTCAmp-
> GetRenderTargetItem().ShaderResourceTexture,0,false); }


      (.....)
}
```

5.5.4 LightingPass Summary

The following summarizes the main processes and steps of the lighting calculation pixel shader:


DeferredLightPixelMain
SetupLightDataForStandardDeferred
GetDynamicLighting
GetDynamicLightingSplit
GetLocalLightAttenuation
GetShadowTerms
IntegrateBxDF_CapsuleLight
LineIrradiance
SphereHorizonCosWrap
ClosestPointLineToRay
IntegrateBxDF_AreaLight
ShadingModelID
SUBSURFACE
CLEAR_COAT
HAIR
SINGLELAYERWATER DEFAULT_LIT THIN_TRANSLUCENT
CLOTH
EYE
...
SubsurfaceBxDF
ClearCoatBxDF
HairBxDF
DefaultLitBxDF
ClothBxDF
EyeBxDF
...
SphereMaxNoH
Diffuse_Lambert
RectGGXApproxLTC
SpecularGGX
LightAccumulator_AddSplit

LightAccumulator_GetResultSplit

The specific logic of the above main steps will not be repeated here, you can refer to the previous code analysis.

5.6 UE Shadows

UE's shadows are of various types, complex to implement, and involve many optimizations techniques. This chapter will spend a lot of space to introduce and explain UE's shadow rendering mechanism.

5.6.1 Shadow Overview

There are many types of UE light sources, and UE shadows have richer types, as described below.

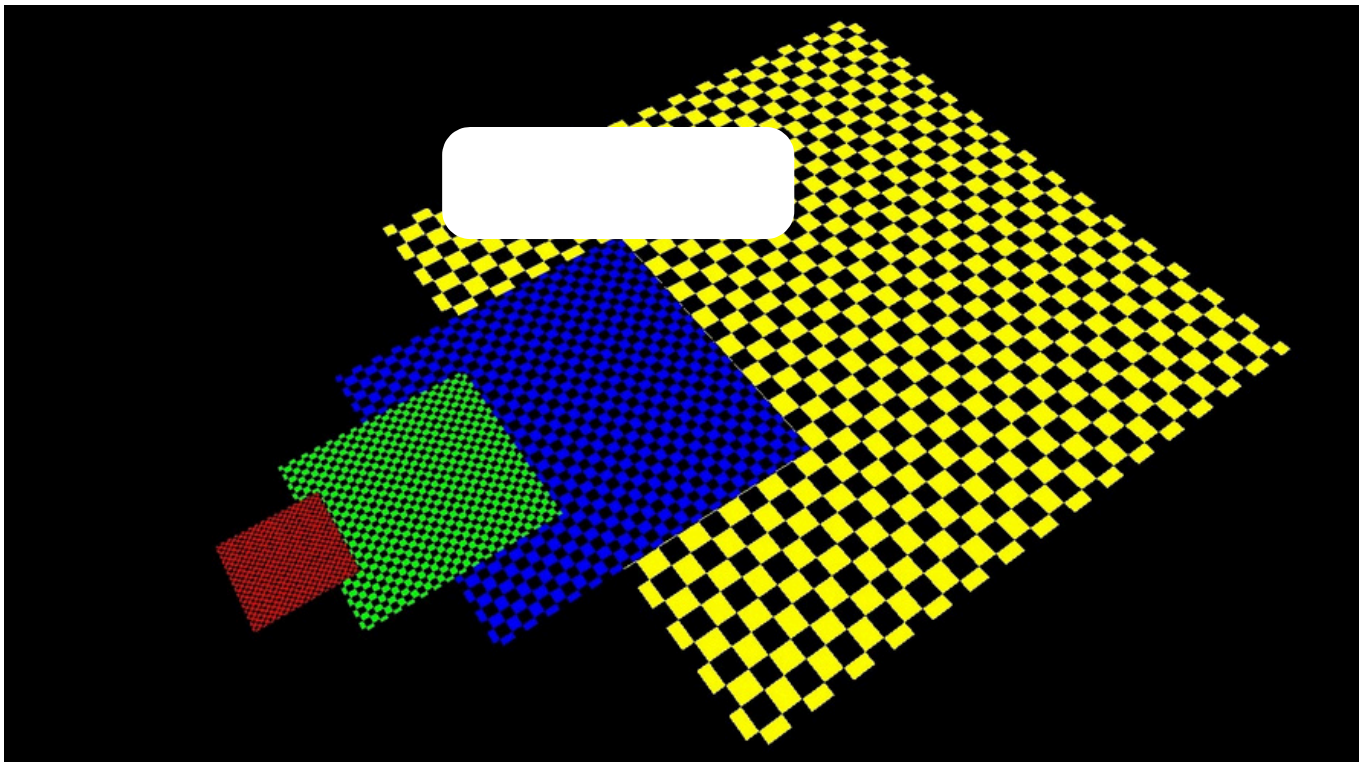
- **Static Shadow**

Static shadows are projections from static light sources onto static objects in static receivers, so they only occur in static objects. Dynamic objects usually cannot produce static shadows.

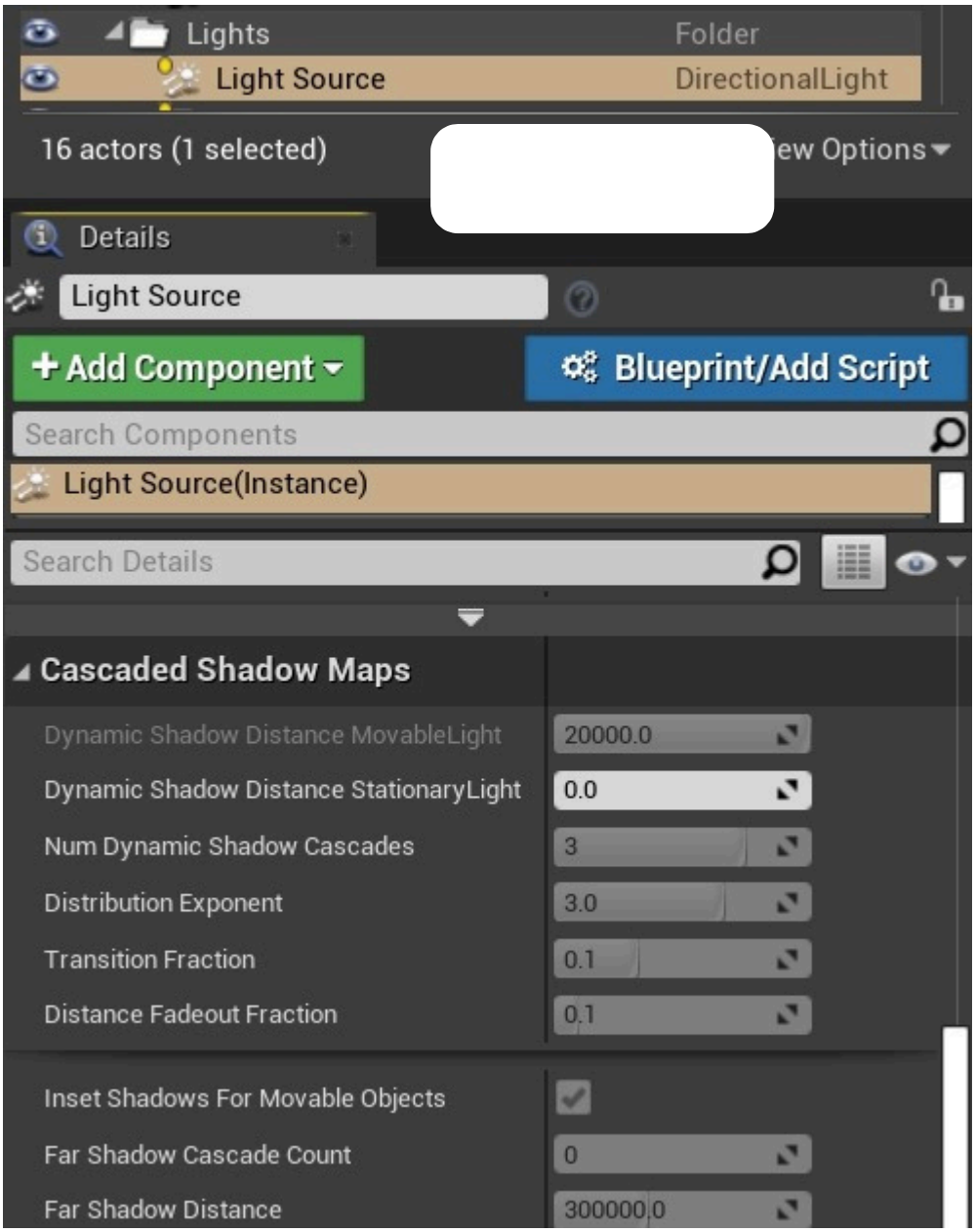
The dynamic character on the left will not be illuminated or cast a shadow under a static light source, while the static character on the right will.

- **Cascading Shadow Map**

Cascading shadows, also known as Directional Light Cascading Shadow Maps or Whole Scene Shadows, are multi-level shadow maps for the entire scene affected by parallel light, in order to solve the problem of various visual artifacts caused by insufficient accuracy of depth maps in the distant light source view space.



Schematic diagram of cascaded shadows. In the light source view space, red is closest and has the smallest depth map resolution, while yellow is farthest and has the largest depth map resolution.



The Cascaded Shadow Maps property group of the parallel light can set the detailed parameters of the cascade shadows.

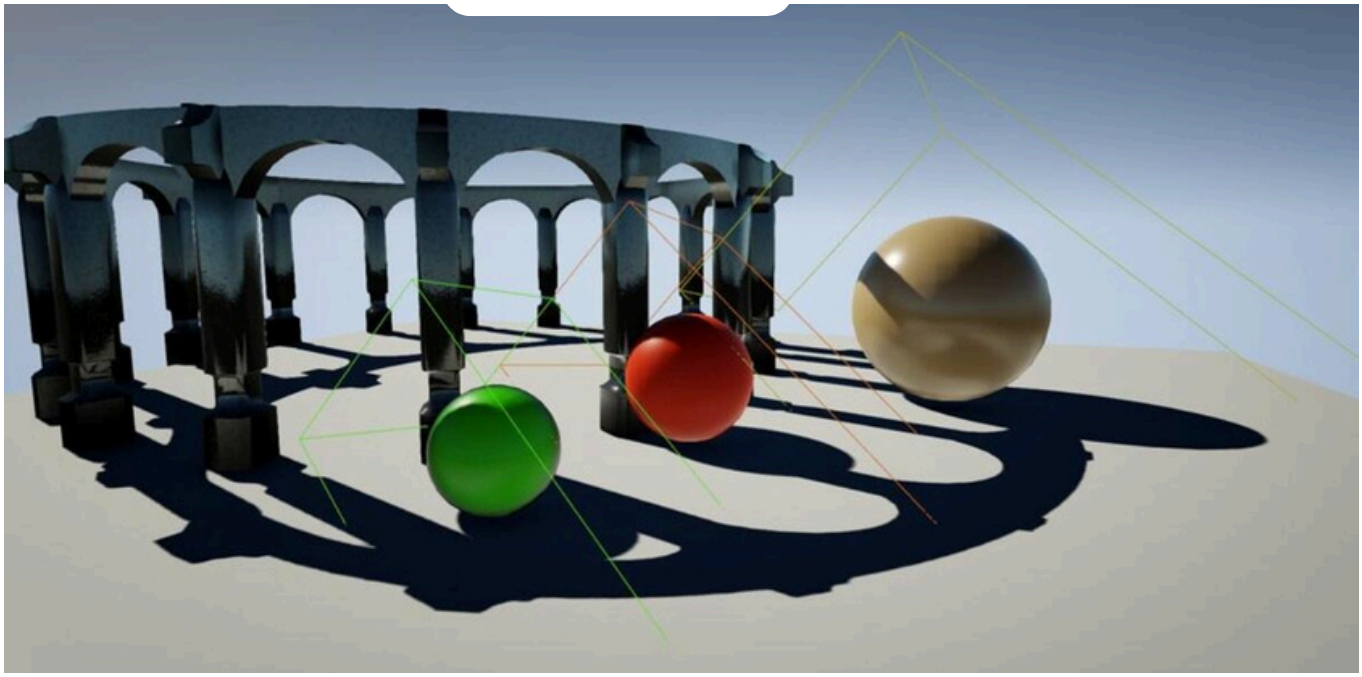
In addition, Directional Stationary Lights are special because they support static shadows while supporting shadows for the entire scene through cascaded shadow maps. This is very useful for levels with a large amount of animated vegetation, because they can support shadows for dynamic objects, and gradually transition to static shadows at a certain distance in the distance.

> The transition area can be adjusted by adjusting the value of Dynamic Shadow Distance StationaryLight in the parallel light cascade shadow property group.

- **Stationary Light Shadows**

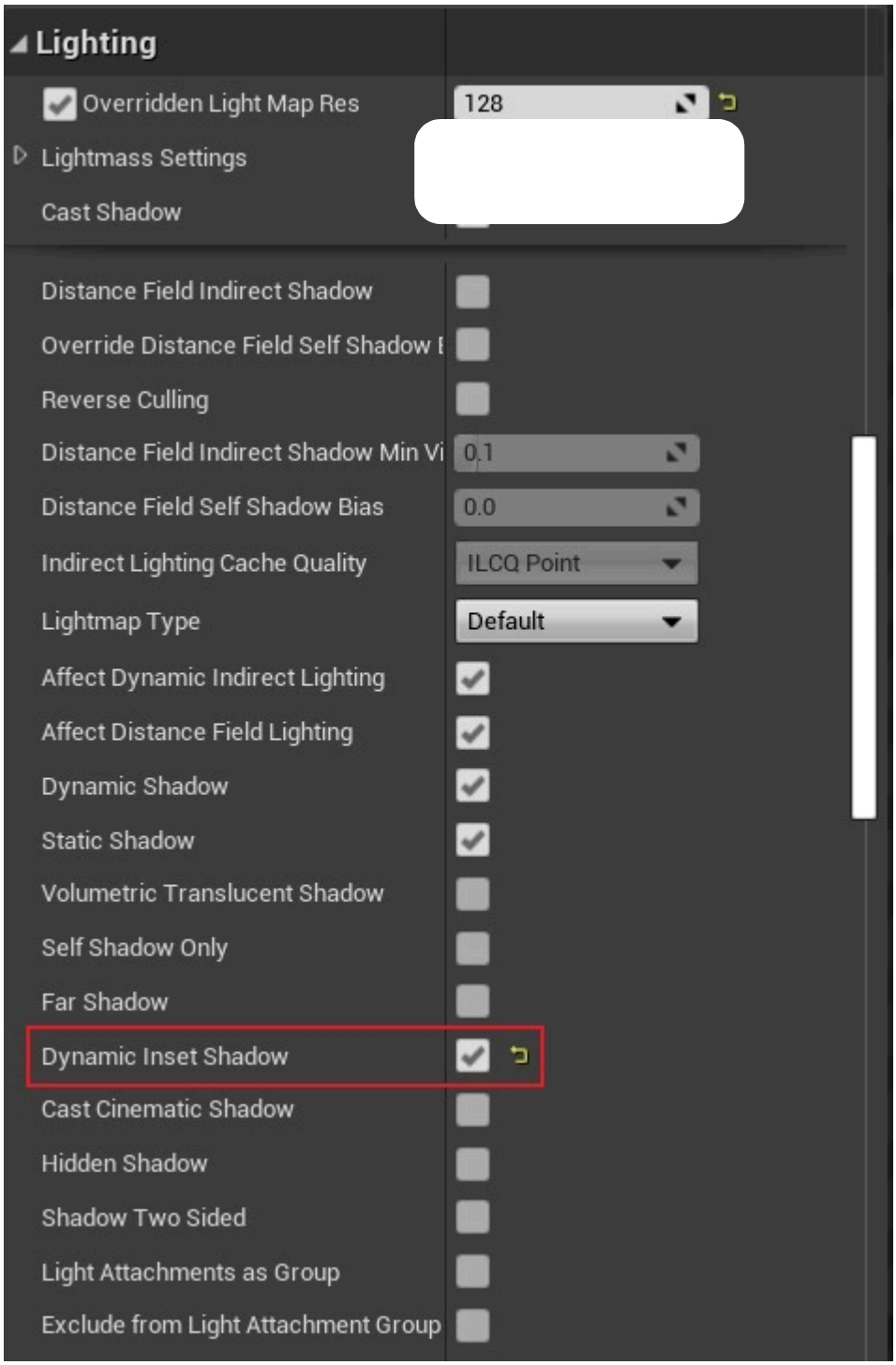
Dynamic objects must be integrated into the static shadows of the world from the distance field shadow map. This is done with the help of per-object shadows. Each movable object generates two dynamic shadows from a stationary light source: one shadow for the static world casting onto the dynamic object, and one shadow for the dynamic object casting onto the world.

With this setup, the only shadowing cost for a stationary light comes from the dynamic objects it affects. This means the cost can be either large or small, depending on how many dynamic objects there are. If there are enough dynamic objects in the scene, it can be more efficient to use movable lights.



- **Per Object Shadow**

Per-object shadows used by movable components apply the shadow map to the bounding box of the object, so the bounding box must be accurate. For skeletal meshes, this means they should have a physics asset. For particle systems, any fixed bounding box must be large enough to contain all the particles.



In the Lighting property group of the mesh, Dynamic Inset Shadow can turn on per-object shadows, which is very useful for objects that require high quality and high precision.

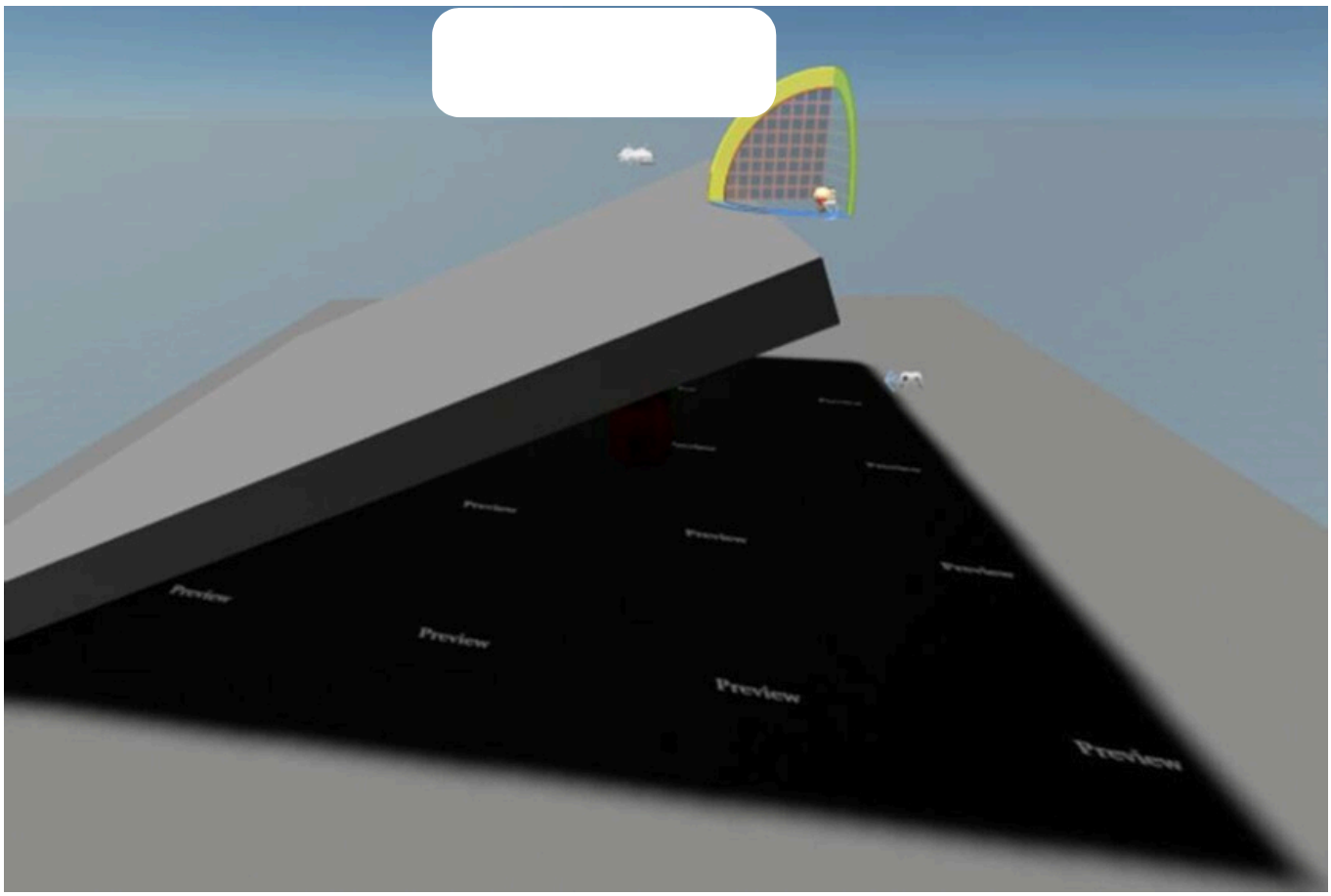
- **Dynamic Shadow**

Movable Lights cast fully dynamic shadows (and light) on all objects. None of this light's data will be baked into the lightmap, and it is free to cast dynamic shadows on everything. Static Meshes, Skeletal Meshes, Particle Effects, etc. will fully cast and receive dynamic shadows from Movable Lights.

Generally speaking, movable dynamic shadow casting lights are the most expensive.

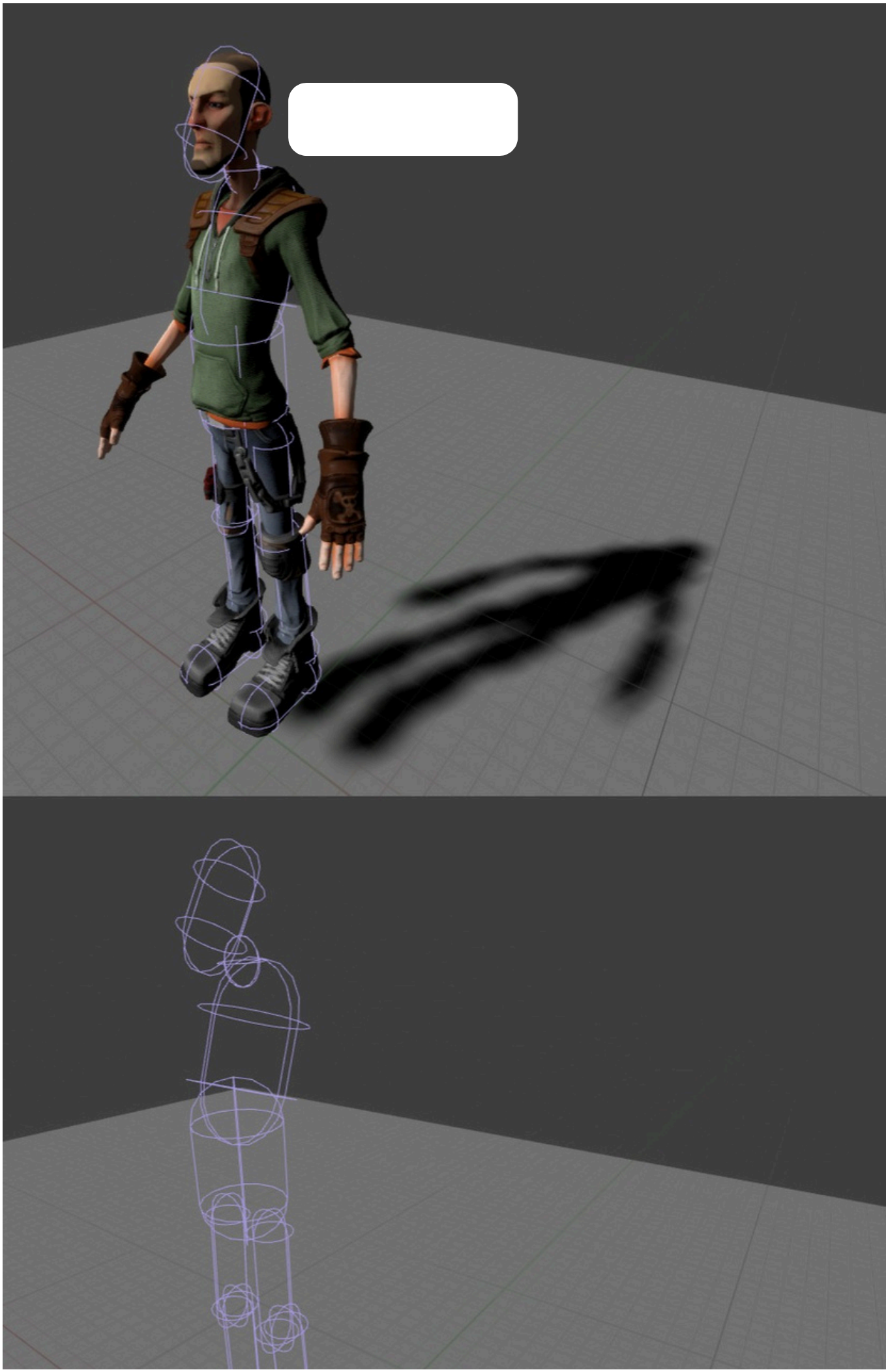
- **Preview Shadow**

When editing stationary or static lights, and they have not been built in time, UE will use preview shadows instead of unbuilt shadows. The preview shadow will have the word Preview:



- **Capsule Shadow**

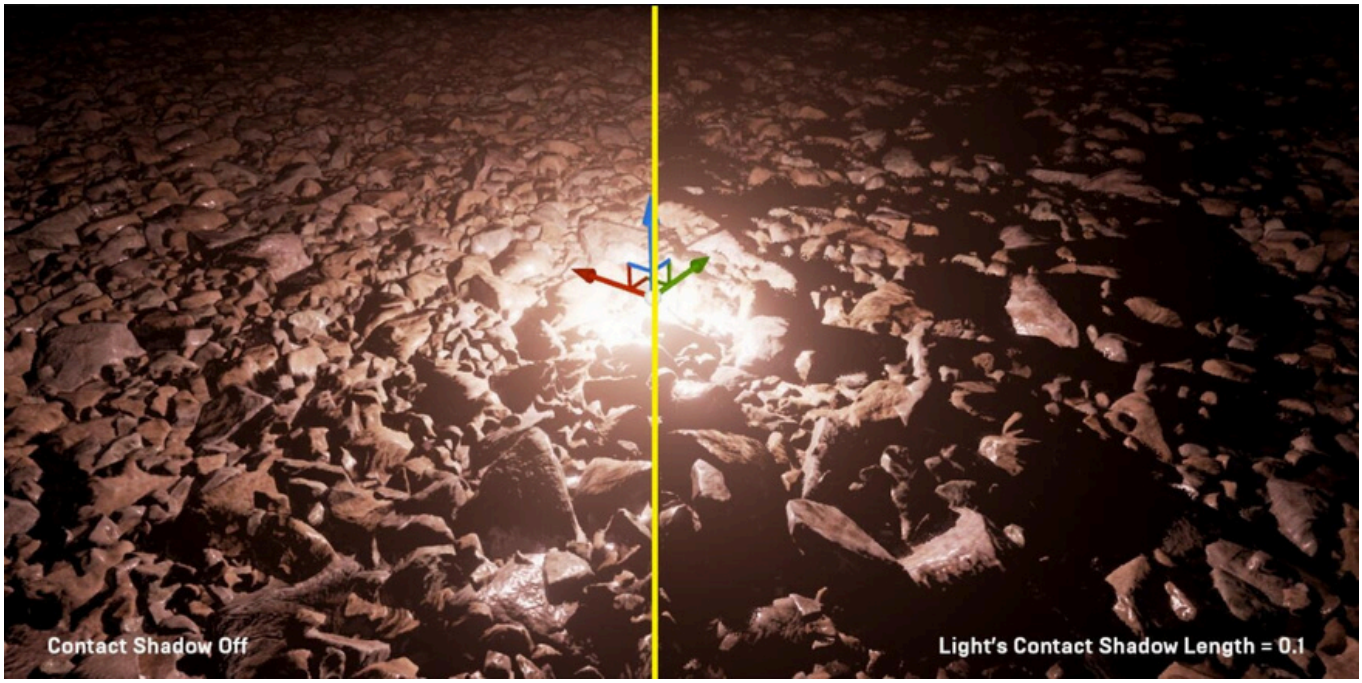
UE supports using capsules generated by the object's physics asset to cast shadows instead of the object itself, which can produce soft shadows and is mainly used for skinned meshes with physics assets.



Top: Capsule shadow. Bottom: corresponding physics asset representation.

- **Contact Shadow**

Contact shadows are screen-space compensated shadows for each light source. The principle is that during the shadow calculation stage, if the contact shadow of the light source is turned on, an additional visibility determination ray will be projected, and Ray Marching of the scene depth buffer will be performed to determine whether the pixel is occluded (in the shadow) , thereby obtaining more accurate shadow information.



Left: Contact shadows off. Right: Contact shadows on.

- **Distance Field Shadow**

Distance field shadows, as the name implies, are shadows cast using pre-generated distance field information.

> Before using features that require scene distance fields, you need to turn on**Generate Mesh Distance Fields**in Project Settings, including distance field shadows, distance field collision, distance field AO, etc.

Distance field shadows support soft shadows, indirect shadows, long-distance shadows, etc., and can also be combined with cascaded shadows to achieve more delicate and realistic shadow effects (see

the figure below).

Top: Cascade Shadows effect; Bottom: Cascade Shadows + Distance Field Shadows.

The following table compares the performance of traditional shadows and distance field shadows (GPU is Radeon 7870, resolution is 1080p, unit is ms):

Scenario	Cascaded, cube shadow map consumption	Distance Field Shadow Cost	Speed up
Parallel light, 10k distance units, 3 CSM levels	3.1	2.3	25%

Scenario	Cascaded, cube shadow map consumption	Distance Field Shadow Cost	Speed up
Parallel light, 30k distance units, 6-level CSM	4.9	2.8	43%
Point light source with a very large radius	1.8	1.3	30%
5 point lights with small radius	3.2	1.8	45%

It can be seen that the rendering performance of distance field shadows will be higher, but it will increase the scene construction time and increase the consumption of disk, memory, and video memory.

5.6.2 Shadow Basic Types

Before entering the analysis of shadow rendering, let's first understand some key concepts related to shadows in detail:

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

//Cast shadow information.
classFProjectedShadowInfo: public FRefCountedObject {

public:
    typedefTArray<constFPrimitiveSceneInfo*,SceneRenderingAllocator> PrimitiveArrayType;

    //Used when rendering shadowsview.
    FViewInfo* ShadowDepthView;
    //Shadows must be rendered on the mainview, NullIndicates independence fromview
    Shadow. FViewInfo* DependentView;

    //shadowUniform Buffer.
    TUniformBufferRef<FShadowDepthPassUniformParameters> ShadowDepthPassUniformBuffer;
    TUniformBufferRef<FMobileShadowDepthPassUniformParameters>
MobileShadowDepthPassUniformBuffer;

    //Shadow map render texture (depth or color).
    FShadowMapRenderTargets RenderTargets;
    // FVisibleLightInfo::AllProjectedShadowsThe index of . int32
    ShadowId;

    //Cache mode.
    EShadowDepthCacheMode CacheMode; //Position to
    offset before transforming the shadow matrix.
    FVector PreShadowTranslation;
    //The shadow's view matrix, used insteadDependentViewThe view
    matrix of . FMatrix ShadowViewMatrix;
```

```
//Subject and Receiver Matrix,  The matrix used to render the shadow depth buffer.
FMatrix     SubjectAndReceiverMatrix;
FMatrix     ReceiverMatrix;
FMatrix     InvReceiverMatrix;

floatInvMaxSubjectDepth;

//The body depth scale, in world space units. Can be used to convert shadow depth values to world
space. floatMaxSubjectZ;
float     MinSubjectZ;
float     MinPreSubjectZ;

//The cone that contains all potential shadow casters.
FConvexVolume     CasterFrustum;
FConvexVolume     ReceiverFrustum;
//The spherical bounding box of the shadow.
FSphere ShadowBounds;
//Cascade shadow settings.
FShadowCascadeSettings
                                    CascadeSettings;

//Boundary size, to preventatlasIntroduces incorrect effects when filtering
shadows. uint32 BorderSize;
//Shadows in the depth buffer (oratlasThe actual shadow map content is in:X + BorderSize, Y + BorderSize. uint32
          X;
uint32     Y;
//The shadow resolution, including the border. The actual shadow resolution allocated is:ResolutionX + 2 *
BorderSize, ResolutionY + 2 * BorderSize.
uint32     ResolutionX;
uint32     ResolutionY;

//Maximum screen percentage, take any oneviewThe maximum width or
height of the . floatMaxScreenPercent; //EachviewThe transition value of .

TArray<float, TInlineAllocator<2> > FadeAlphas;

//Whether the shadow is allocated in the depth buffer, if so, thenXandYThe properties will be
initialized. uint32 bAllocated :1; //Whether shadow casters have been rendered.

uint32 bRendered :1;

//Is the shadow already inpreshadowAllocated in cache, if yes, thenXandYthat ispreshadowCache DepthbufferThe offset of.
uint32 bAllocatedInPreshadowCache:1; //Is the shadow inpreshadowThe cache is in place and its depth has been updated.
uint32 bDepthsCached :1;


uint32 bDirectionalLight :1;

//Whether to render in the same passcubemapPoint light shadows on all faces
of . uint32 bOnePassPointLightShadow:1; //Whether shadows affect the entire
scene or a group of objects.
uint32 bWholeSceneShadow :1; //whetherRSM
(Reflective Shadowmap). uint32
bReflectiveShadowmap :1; //Whether
transparent objects have shadows.
uint32 bTranslucentShadow :1; //Whether to
use capsule shadows.
uint32 bCapsuleShadow :1;
//Whether to pre-shadow, which is the process of processing per-object shadows cast by static environments onto dynamic receivers.
```

```
    uint32 bPreShadow :1;
    //Is it only self-shadowing? If so, it will not be projected onto objects other than itself, and has high-quality shadows (suitable for first-person
    games). uint32 bSelfShadowOnly :1; //Whether to use per-object opaque shadows.

    uint32 bPerObjectOpaqueShadow:1; //Whether to
    enable backlight transmission.
    uint32 bTransmission :1;

    //For point light renderingcubemap6The view projection matrix used by the shadow
    map for each face. TArray<FMatrix> OnePassShadowViewProjectionMatrices; //For
    point light renderingcubemap6The view matrix used by the shadow map for each
    face. TArray<FMatrix> OnePassShadowViewMatrices;
    /** Frustums for each cubemap face, used for object culling one pass point light shadows. */

    TArray<FConvexVolume> OnePassShadowFrustums;

    (.....)

    //Control transition parameters beyond per-object shadows,    Prevents super sharp shadows in the distance.
    float    PerObjectShadowFadeStart;
    float    InvPerObjectShadowFadeLength;

public:
    //Set up per-object shadows.
    bool SetupPerObjectProjection(FLightSceneInfo* InLightSceneInfo, ...); //Sets the whole scene
    (panoramic) shadow.
    void SetupWholeSceneProjection(FLightSceneInfo* InLightSceneInfo, ...);

    //Renders shadow depth for opaque objects.
    void RenderDepth(FRHICommandListImmediate& RHICmdList, ...); //Renders shadow
    depth for transparent objects.
    void RenderTranslucencyDepths(FRHICommandList& RHICmdList, ...); //For specialview
    Renders shadows cast onto the scene.
    void RenderProjection(FRHICommandListImmediate& RHICmdList, ...)const; //Renders single-
    pass point light shadows.
    void RenderOnePassPointLightProjection(FRHICommandListImmediate& RHICmdList, ...) const;

    void RenderFrustumWireframe(FPrimitiveDrawInterface*PDI)const;

    //Rendering state interface.
    void SetStateForView(FRHICommandList& RHICmdList)const;
    void SetStateForDepth(FMeshPassProcessorRenderState& DrawRenderState)const; void ClearDepth
    (FRHICommandList& RHICmdList, class FSceneRenderer* SceneRenderer,
. . . ) ;
    staticFRHIBlendState*GetBlendStateForProjection(int32 ShadowMapChannel, ...); FRHIBlendState*
    GetBlendStateForProjection(boolbProjectingForForwardShading, ...) const;


    //Add the main graphics element that needs to cast shadows.
    void AddSubjectPrimitive(FPrimitiveSceneInfo* PrimitiveSceneInfo, TArray<FViewInfo>* ViewArray, ...);

    //Add shadow receiver primitives.
    void AddReceiverPrimitive(FPrimitiveSceneInfo* PrimitiveSceneInfo);

    //Gather dynamic mesh elements for all primitives that should cast shadows.
    void GatherDynamicMeshElements(FSceneRenderer& Renderer, ...); //Will
    DynamicMeshElementConvert toFMeshDrawCommand.
    void SetupMeshDrawCommandsForShadowDepth(FSceneRenderer& Renderer, ...);
```

```
    void SetupMeshDrawCommandsForProjectionStenciling(FSceneRenderer& Renderer);

    //Create a new one from the memory poolviewAnd inShadowDepthViewCache it in the image to render shadow depth.
    void SetupShadowDepthView(FRHICommandListImmediate& RHICmdList, FSceneRenderer*
    SceneRenderer);
    //Setup and UpdateUniform Buffer.
    void SetupShadowUniformBuffers(FRHICommandListImmediate& RHICmdList, FScene* Scene,
...);
    //Ensure that the cached shadow map is inEReadablestate.
    void TransitionCachedShadowmap(FRHICommandListImmediate& RHICmdList, FScene* Scene);

     //   Calculation and UpdateShaderDepthBiasandShaderSlopeDepthBias.
    void   UpdateShaderDepthBias();
    //calculatePCFCompare parameters.
    float ComputeTransitionSize()              const;

     //Data acquisition and operation interface.
    float     GetShaderDepthBias()const;
    float     GetShaderSlopeDepthBias()const;
    float     GetShaderMaxSlopeDepthBias()       const;
    float     GetShaderReceiverDepthBias()       const;

    bool HasSubjectPrims()const;
    bool SubjectsVisible(constFViewInfo& View)const; void
    ClearTransientArrays();
    friend uint32GetTypeHash(constFProjectedShadowInfo* ProjectedShadowInfo);

    FMatrixGetScreenToShadowMatrix(constFSceneView& View)const; FMatrix
    GetScreenToShadowMatrix(constFSceneView& View, ...)const; FMatrix
    GetWorldToShadowMatrix(FVector4& ShadowmapMinMax, ...)const; FIntPoint
    GetShadowBufferResolution()const

    bool    IsWholeSceneDirectionalShadow()          const;
     bool IsWholeScenePointLightShadow()        const;
    constFLightSceneInfo&GetLightSceneInfo()const;
    constFLightSceneInfoCompact&GetLightSceneInfoCompact()const; const
    FPrimitiveSceneInfo*GetParentSceneInfo()const; FShadowDepthType
    GetShadowDepthType()const;

    (.....)

private:
    constFLightSceneInfo*          LightSceneInfo;
    FLightSceneInfoCompact         LightSceneInfoCompact;
    constFPrimitiveSceneInfo*            ParentSceneInfo;

    //List of shadow casting primitives.
    PrimitiveArrayType DynamicSubjectPrimitives; //Receiver
    primitive, onlypreshadowefficient.
    PrimitiveArrayTypeReceiverPrimitives; //List of transparent
    shadow casting primitives.
    PrimitiveArrayType SubjectTranslucentPrimitives;

    //The mesh element corresponding to the primitive that casts the shadow.
    TArray<FMeshBatchAndRelevance,SceneRenderingAllocator>                DynamicSubjectMeshElements;
    TArray<FMeshBatchAndRelevance,SceneRenderingAllocator>
    DynamicSubjectTranslucentMeshElements;
```

```
    TArray<constFStaticMeshBatch*, SceneRenderingAllocator>
    SubjectMeshCommandBuildRequests;

    // DynamicSubjectMeshElementsquantity. int32
    NumDynamicSubjectMeshElements; //
    SubjectMeshCommandBuildRequestsquantity.
    int32 NumSubjectMeshCommandBuildRequestElements;


    //The drawing commands/rendering state etc. needed to draw the shadow.
    FMeshCommandOneFrameArray ShadowDepthPassVisibleCommands;
    FParallelMeshDrawCommandPass ShadowDepthPass;
    TArray<FShadowMeshDrawCommandPass, TInlineAllocator<2>> ProjectionStencilingPasses;
    FDynamicMeshDrawCommandStorage DynamicMeshDrawCommandStorage;
    FGraphicsMinimalPipelineStateSet GraphicsMinimalPipelineStateSet; bool
    NeedsShaderInitialisation;


    //The offset value when rendering shadows. Will beUpdateShaderDepthBias()Set, beGetShaderDepthBias()Get, -1Indicates
    uninitialized.
    float     ShaderDepthBias;
    float     ShaderSlopeDepthBias;
    float     ShaderMaxSlopeDepthBias;


    //Internal Interface
    void CopyCachedShadowMap(FRHICommandList& RHICmdList, ...); void
    RenderDepthInner(FRHICommandListImmediate& RHICmdList, ...);
    void ModifyViewForShadow(FRHICommandList& RHICmdList, FViewInfo* FoundView)const; FViewInfo*
    FindViewForShadow(FSceneRenderer* SceneRenderer)const; void AddCachedMeshDrawCommandsForPass
    (int32 PrimitiveIndex, ...); bool ShouldDrawStaticMeshes(FViewInfo& InCurrentView, ...); void
    GetShadowTypeNameForDrawEvent(FString& TypeName)const; int32UpdateShadowCastingObjectBuffers()
    const;


    void GatherDynamicMeshElementsArray(FViewInfo* FoundView, ...); void
    SetupFrustumForProjection(constFViewInfo* View, ...)const;
    void SetupProjectionStencilMask(FRHICommandListImmediate& RHICmdList, ...)const;
};
```

As can be seen from the above code,FProjectedShadowInfoit almost includes the important data and operation interfaces required for shadow processing and rendering. Of course, UE's shadow system is too complex, and it alone is not enough to solve all the rendering functions of shadows.
Let's continue to analyze other basic or key types:

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

//View-independent visible light source information, mainly shadow-related
information. classFVisibleLightInfo {

public:
    //In the scene memory stack (mem stack)Allocates and manages cast shadow information.
    TArray<FProjectedShadowInfo*,SceneRenderingAllocator> //All visible cast shadow information, output by the
    shadow setup phase, not all of which will be rendered. TArray<FProjectedShadowInfo*,SceneRenderingAllocator>
    AllProjectedShadows;

    //Special shadow casting information, used for specialized features such as projectile/
    capsule shadows/RSMwait. TArray<FProjectedShadowInfo*,SceneRenderingAllocator>
    TArray<FProjectedShadowInfo*,SceneRenderingAllocator>
    TArray<FProjectedShadowInfo*,SceneRenderingAllocator>
```

```
MemStackProjectedShadows;
ShadowsToProject;
CapsuleShadowsToProject;
RSMsToProject;
```

```
    //All visible cast pre-shadows. These are not allocated and managed in the scene's memory stack, so need to be used
    TRefCountPtrReference counting. TArray<TRefCountPtr<FProjectedShadowInfo>,SceneRenderingAllocator>
    ProjectedPreShadows;
    //Occluded per-object shadows, which need to be tracked in order to submit occlusion culling requests.
    TArray<FProjectedShadowInfo*,SceneRenderingAllocator> OccludedPerObjectShadows;
};

//View-related visible light source information.
classFVisibleLightViewInfo {

public:
    //The visible primitives that the light can affect.
    TArray<FPrimitiveSceneInfo*,SceneRenderingAllocator>                VisibleDynamicLitPrimitives;
    //correspondFVisibleLightInfo::AllProjectedShadowsThe shadow visibility map for .
    FSceneBitArray ProjectedShadowVisibilityMap;
    //correspondFVisibleLightInfo::AllProjectedShadowsShadowViewRelevance.
    TArray<FPrimitiveViewRelevance,SceneRenderingAllocator>
    ProjectedShadowViewRelevanceMap;
    //Is it inside the viewing frustum? (Directional lights/Skylights are
    alwaystrue) uint32 bInViewFrustum :1;


    (......)
};

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

classFSceneViewState
{
public:
    //The key value of the cast shadow. Mainly used to compare whether two cast shadow
    instances are the same. classFProjectedShadowKey {

    public:
        //Key-value comparison interface.
        inlinebooloperator == (constFProjectedShadowKey &Other)const {

            return(PrimitiveId == Other.PrimitiveId && Light == Other.Light &&
ShadowSplitIndex == Other.ShadowSplitIndex && bTranslucentShadow ==
Other.bTranslucentShadow);
        }
        //Key-value hash interface.
        friendinlineuint32GetTypeHash(constFSceneViewState::FProjectedShadowKey& Key) {


            returnPointerHash(Key.Light,GetTypeHash(Key.PrimitiveId));
        }

    private:
        //Shadow Primitivesid.
        FPrimitiveComponentId          PrimitiveId;
        //The light source of the shadow.
        constULightComponent*
        //The index of the shadow in the shadow atlas.      Light;
        int32 ShadowSplitIndex; //Whether to
        transparently shadow.
        boolbTranslucentShadow;
```

```
    };
};
```

5.6.3 Shadow Initialization

This section will spend a lot of time analyzing the initialization of shadows. If you don't want to read redundant code analysis, you can jump directly to**5.6.3.12 Summary of Shadow Initialization**.

The initialization of the shadow is in the InitViews stage, and the call stack diagram is as follows:

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

    bool FDeferredShadingSceneRenderer::InitViews(RHICmdList, ...) {

        void
FDeferredShadingSceneRenderer::InitViewsPossiblyAfterPrepass(FRHICommandListImmediate& RHICmdList, ...)

        {
            //Initialize dynamic shadows.
            void FSceneRenderer::InitDynamicShadows(FRHICommandListImmediate& RHICmdList,
. . . )
            {
                (......)
            }
        }
    }
}
```

5.6.3.1 InitDynamicShadows

The following isFSceneRenderer::InitDynamicShadowsa code analysis (excerpt):

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

void FSceneRenderer::InitDynamicShadows(FRHICommandListImmediate&          RHICmdList,
FGlobalDynamicIndexBuffer& DynamicIndexBuffer,          FGlobalDynamicVertexBuffer&
DynamicVertexBuffer, FGlobalDynamicReadBuffer& {          DynamicReadBuffer)

    //Initialize various flags and quantities.
    const boolbMobile = FeatureLevel < ERHIFeatureLevel::SM5; bool
    bStaticSceneOnly =false;
    for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        FViewInfo& View = Views[ViewIndex];
        bStaticSceneOnly = bStaticSceneOnly || View.bStaticSceneOnly;
    }

    const boolbProjectEnablePointLightShadows = Scene-
> ReadOnlyCVARCache.bEnablePointLightShadows; uint32
    NumPointShadowCachesUpdatedThisFrame =0; uint32
    NumSpotShadowCachesUpdatedThisFrame =0;

    //Precomputed shadows.
    TArray<FProjectedShadowInfo*,SceneRenderingAllocator> PreShadows;
```

```
//The full-view shadow associated with the view.
TArray<FProjectedShadowInfo*,SceneRenderingAllocator> //The view's associated full-view shadow that needs to be clipped.
ViewDependentWholeSceneShadows;
TArray<FProjectedShadowInfo*,SceneRenderingAllocator>
ViewDependentWholeSceneShadowsThatNeedCulling;
{
    // Traverse all light sources and add different types of light sources to the list of different types of shadows to be rendered.
    for (TSparseArray<FLightSceneInfoCompact>::TConstIterator LightIt(Scene->Lights);
LightIt; ++LightIt)
    {
        constFLightSceneInfoCompact& LightSceneInfoCompact = *LightIt; FLightSceneInfo*
        LightSceneInfo = LightSceneInfoCompact.LightSceneInfo;

        FScopeCycleCounterContext(LightSceneInfo->Proxy->GetStatId());

        FVisibleLightInfo& VisibleLightInfo = VisibleLightInfos[LightSceneInfo->Id];

        //LightOcclusionTypeThere are two types: shadow map and ray tracing. If it is not a shadow map type, it
        will be ignored. constFLightOcclusionType OcclusionType =
GetLightOcclusionType(LightSceneInfoCompact);
        if(OcclusionType != FLightOcclusionType::Shadowmap)
            continue;

        // If the light does not have shadows enabled or the shadow quality is too low, the
        if shadow map is ignored. ((LightSceneInfoCompact.bCastStaticShadow ||
LightSceneInfoCompact.bCastDynamicShadow) && GetShadowQuality() >0)
        {
            //Detect whether the light source is in a certainviewIf not visible, it
            is ignored. boolbIsVisibleInAnyView =false;
            for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                bIsVisibleInAnyView = LightSceneInfo-
> ShouldRenderLight(Views[ViewIndex]);

                if(bIsVisibleInAnyView) {

                    break;
                }
            }

            // All clipping conditions are passed, processing the shadow of the light source.
            if (bIsVisibleInAnyView)
            {
                //Initialize various flags and variables of the shadow.
                static constautoAllowStaticLightingVar =
IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.AllowStaticLighting"));
                const boolbAllowStaticLighting = (!AllowStaticLightingVar ||
AllowStaticLightingVar->GetValueOnRenderThread() !=0);
                //Whether to use point light shadows. Note that rectangular lights are also treated as point lights.
                const boolbPointLightShadow = LightSceneInfoCompact.LightType ==
LightType_Point || LightSceneInfoCompact.LightType == LightType_Rect;

                //For moving lights that do not precompute shadows, only full-view shadows are created (
                whole scene shadow). const boolbShouldCreateShadowForMovableLight =
                    LightSceneInfoCompact.bCastDynamicShadow
                    && (!LightSceneInfo->Proxy->HasStaticShadowing() ||
!bAllowStaticLighting);
```

```
const boolbCreateShadowForMovableLight = bShouldCreateShadowForMovableLight && (!bPointLightShadow || bProjectEnablePointLightShadows);

//Create full-view shadows for lights with precomputed shadows that have not yet been built.
const boolbShouldCreateShadowToPreviewStaticLight =
    LightSceneInfo->Proxy->HasStaticShadowing() &&
        LightSceneInfoCompact.bCastStaticShadow !LightSceneInfo-
    && >IsPrecomputedLightingValid();

const boolbCreateShadowToPreviewStaticLight =
    bShouldCreateShadowToPreviewStaticLight
    && (!bPointLightShadow || bProjectEnablePointLightShadows);

//Create full-view shadows for lights that need static shadows but do not have a valid shadow map due to overlap. const boolbShouldCreateShadowForOverflowStaticShadowing =
    LightSceneInfo->Proxy->HasStaticShadowing() &&
        !LightSceneInfo->Proxy->HasStaticLighting()
    &&LightSceneInfoCompact.bCastStaticShadow LightSceneInfo-
    && >IsPrecomputedLightingValid()
    && LightSceneInfo->Proxy->GetShadowMapChannel() == INDEX_NONE;

const boolbCreateShadowForOverflowStaticShadowing =
    bShouldCreateShadowForOverflowStaticShadowing
    && (!bPointLightShadow || bProjectEnablePointLightShadows);

//Added full scene shadows for point lights.
const boolbPointLightWholeSceneShadow =
(bShouldCreateShadowForMovableLight || bShouldCreateShadowForOverflowStaticShadowing ||
bShouldCreateShadowToPreviewStaticLight) && bPointLightShadow;
if(bPointLightWholeSceneShadow) {


UsedWholeScenePointLightNames.Add(LightSceneInfoCompact.LightSceneInfo->Proxy-
> GetComponentName());
}

//Creates full-view shadows for light sources.
if(bCreateShadowForMovableLight || bCreateShadowToPreviewStaticLight
||  bCreateShadowForOverflowStaticShadowing)
{
    CreateWholeSceneProjectedShadow(LightSceneInfo,
NumPointShadowCachesUpdatedThisFrame, NumSpotShadowCachesUpdatedThisFrame);
}

//  Allows creation of both mobile and fixed light sourcesCSM(Cascaded Shadows), or static lights
if  that have not yet been built. ((!LightSceneInfo->Proxy->HasStaticLighting() &&
LightSceneInfoCompact.bCastDynamicShadow)        || bCreateShadowToPreviewStaticLight)
{
    //Added view-related full-view shadows.
    if(   !bMobile ||
        ((LightSceneInfo->Proxy->UseCSMForDynamicObjects()        ||
LightSceneInfo->Proxy->IsMovable())
            && (LightSceneInfo == Scene->MobileDirectionalLights[0] ||
LightSceneInfo == Scene->MobileDirectionalLights[1] || LightSceneInfo == Scene-
> MobileDirectionalLights[2])))
    {
```

```
AddViewDependentWholeSceneShadowsForView(ViewDependentWholeSceneShadows,
ViewDependentWholeSceneShadowsThatNeedCulling, VisibleLightInfo, *LightSceneInfo);
                                }

                                //Handles interactive shadows, where interaction refers to the influence between light sources and primitives. ContainsPerObjectShadow, Transparency
Shadows, self-shadows, etc.
                                if( !bMobile || (LightSceneInfo->Proxy->CastsModulatedShadows() &&
!LightSceneInfo->Proxy->UseCSMForDynamicObjects()))
                                {
                                    Scene->FlushAsyncLightPrimitiveInteractionCreation();

                                    //Handles interactive shadows for dynamic primitives.
                                    for(FLightPrimitiveInteraction* Interaction = LightSceneInfo-
> GetDynamicInteractionOftenMovingPrimitiveList(false); Interaction; Interaction = Interaction-
>GetNextPrimitive())
                                    {
                                        SetupInteractionShadows(RHICmdList, Interaction,
VisibleLightInfo, bStaticSceneOnly, ViewDependentWholeSceneShadows, PreShadows);
                                    }

                                    //Handles interactive shadows for static primitives.
                                    for(FLightPrimitiveInteraction* Interaction = LightSceneInfo-
> GetDynamicInteractionStaticPrimitiveList(false); Interaction; Interaction = Interaction-
> GetNextPrimitive())
                                    {
                                        SetupInteractionShadows(RHICmdList, Interaction,
VisibleLightInfo, bStaticSceneOnly, ViewDependentWholeSceneShadows, PreShadows);
                                    }
                                }
                            }
                        }
                    }
                }

                //Calculates visibility of cast shadows.
                InitProjectedShadowVisibility(RHICmdList);
            }

            //Clears old precomputed shadows and tries to add new ones to the cache.
            UpdatePreshadowCache(FSceneRenderTargets::Get(RHICmdList));

            //Collects a list of primitives for drawing different types of shadows.
            GatherShadowPrimitives(PreShadows, ViewDependentWholeSceneShadowsThatNeedCulling,
            bStaticSceneOnly);

            //Assigns the shadow depth render texture.
            AllocateShadowDepthTargets(RHICmdList);

            //Collect the dynamic mesh elements of the shadow, as previously analyzedGatherDynamicMeshElements
            similar. GatherShadowDynamicMeshElements(DynamicIndexBuffer, DynamicVertexBuffer,
            DynamicReadBuffer);
}
```

5.6.3.2 CreateWholeSceneProjectedShadow

Continue to analyze several important interfaces involved in the above code, first of all

CreateWholeSceneProjectedShadow:

```
void FSceneRenderer::CreateWholeSceneProjectedShadow(FLightSceneInfo* LightSceneInfo, uint32&
InOutNumPointShadowCachesUpdatedThisFrame, uint32&
InOutNumSpotShadowCachesUpdatedThisFrame)
{
	SCOPE_CYCLE_COUNTER(STAT_CreateWholeSceneProjectedShadow); FVisibleLightInfo&
	VisibleLightInfo = VisibleLightInfos[LightSceneInfo->Id];

	//Attempts to create a full-view shadow-casting initializer for a light.
	TArray<FWholeSceneProjectedShadowInitializer, TInlineAllocator<6> >
	ProjectedShadowInitializers;
	if(LightSceneInfo->Proxy->GetWholeSceneProjectedShadowInitializer(ViewFamily,
	ProjectedShadowInitializers))
	{
		FSceneRenderTargets& SceneContext_ConstantsOnly =
		FSceneRenderTargets::Get_FrameConstantsOnly();

		//Shadow resolution constant.
		constuint32 ShadowBorder =
ProjectedShadowInitializers[0].bOnePassPointLightShadow ?0: SHADOW_BORDER;
		constuint32 EffectiveDoubleShadowBorder = ShadowBorder *2; constuint32
		MinShadowResolution = FMath::Max<int32>(0,
		CVarMinShadowResolution.GetValueOnRenderThread());
		constint32 MaxShadowResolutionSetting =
		GetCachedScalabilityCVars().MaxShadowResolution;
		constFIntPoint ShadowBufferResolution =
		SceneContext_ConstantsOnly.GetShadowDepthTextureResolution();
		constuint32 MaxShadowResolution = FMath::Min(MaxShadowResolutionSetting,
ShadowBufferResolution.X) - EffectiveDoubleShadowBorder;
		constuint32 MaxShadowResolutionY = FMath::Min(MaxShadowResolutionSetting,
ShadowBufferResolution.Y) - EffectiveDoubleShadowBorder;
		constuint32 ShadowFadeResolution = FMath::Max<int32>(0,
		CVarShadowFadeResolution.GetValueOnRenderThread());

		//The maximum resolution required to calculate shadows within the view, including unrestricted
		resolution for transitions. floatMaxDesiredResolution =0;
		TArray<float, TInlineAllocator<2> > FadeAlphas; float
		MaxFadeAlpha =0; boolbStaticSceneOnly =false; bool
		bAnyViewIsSceneCapture =false;

		for(int32 ViewIndex =0, ViewCount = Views.Num(); ViewIndex < ViewCount;
+ + ViewIndex)
		{
			constFViewInfo& View = Views[ViewIndex];

			const floatScreenRadius = LightSceneInfo->Proxy-
> GetEffectiveScreenRadius(View.ShadowViewMatrices);

			//Calculate the resolution scaling factor
			UnclampedResolution. floatUnclampedResolution =1.0f;

			switch(LightSceneInfo->Proxy->GetLightType()) {

			caseLightType_Point:
				UnclampedResolution = ScreenRadius *
CVarShadowTexelsPerPixelPointlight.GetValueOnRenderThread();
				break;
			caseLightType_Spot:
```

```
                    UnclampedResolution = ScreenRadius *
CVarShadowTexelsPerPixelSpotlight.GetValueOnRenderThread();
                    break;
                caseLightType_Rect:
                    UnclampedResolution = ScreenRadius *
CVarShadowTexelsPerPixelRectlight.GetValueOnRenderThread();
                    break;
                default:
                    //Directional lights are not handled here.
                    checkf(false, TEXT("Unexpected LightType %d appears in
CreateWholeSceneProjectedShadow %s"),
                        (int32)LightSceneInfo->Proxy->GetLightType(),
                        * LightSceneInfo->Proxy->GetComponentName().ToString());
                }

                //In ApplicationShadowResolutionScaleCalculate the transition factor before contributingFadeAlpha.
                const floatFadeAlpha = CalculateShadowFadeAlpha( UnclampedResolution,
ShadowFadeResolution, MinShadowResolution ) * LightSceneInfo->Proxy->GetShadowAmount();
                MaxFadeAlpha = FMath::Max(MaxFadeAlpha, FadeAlpha);
                FadeAlphas.Add(FadeAlpha);

                const floatShadowResolutionScale = LightSceneInfo->Proxy-
> GetShadowResolutionScale();

                floatClampedResolution = UnclampedResolution;

                if(ShadowResolutionScale >1.0f) {

                    ClampedResolution *= ShadowResolutionScale;
                }

                ClampedResolution = FMath::Min<float>(ClampedResolution, MaxShadowResolution);

                if(ShadowResolutionScale <=1.0f) {

                    ClampedResolution *= ShadowResolutionScale;
                }

                MaxDesiredResolution = FMath::Max(
                    MaxDesiredResolution,
                    FMath::Max<float>(
                        ClampedResolution,
                        FMath::Min<float>(MinShadowResolution,          ShadowBufferResolution.X          -
EffectiveDoubleShadowBorder)
                        )
                    ) ;

                bStaticSceneOnly = bStaticSceneOnly || View.bStaticSceneOnly; bAnyViewIsSceneCapture =
                bAnyViewIsSceneCapture || View.bIsSceneCapture;
            }

            //A shadow is created only when the transition factor is greater than the threshold.
            if(MaxFadeAlpha >1.0f/256.0f) {

                Scene->FlushAsyncLightPrimitiveInteractionCreation();

                //Traverse all shadow initializers of the light source, create and set according to the number of cascadesProjectedShadowInfo.
                for(int32 ShadowIndex =0, ShadowCount = ProjectedShadowInitializers.Num();
```

```
ShadowIndex < ShadowCount; ShadowIndex++)
		{
			FWholeSceneProjectedShadowInitializer&		ProjectedShadowInitializer		=
ProjectedShadowInitializers[ShadowIndex];

			int32 RoundedDesiredResolution = FMath::Max<int32>((1<<
(FMath::CeilLogTwo(MaxDesiredResolution +1.0f) -1)) - ShadowBorder *2,1);
			int32 SizeX = MaxDesiredResolution >= MaxShadowResolution ?
MaxShadowResolution : RoundedDesiredResolution;
			int32 SizeY = MaxDesiredResolution >= MaxShadowResolutionY ?
MaxShadowResolutionY : RoundedDesiredResolution;

			if(ProjectedShadowInitializer.bOnePassPointLightShadow) {

				// Round to a resolution that is supported for one pass point light
shadows
				SizeX = SizeY =
SceneContext_ConstantsOnly.GetCubeShadowDepthZResolution(SceneContext_ConstantsOnly.GetCub
eShadowDepthZIndex(MaxDesiredResolution));
			}

			int32 NumShadowMaps =1;
			EShadowDepthCacheMode CacheMode[2] = { SDCM_Uncached, SDCM_Uncached };

			if(!bAnyViewIsSceneCapture &&
!ProjectedShadowInitializer.bRayTracedDistanceField)
			{
				FIntPointShadowMapSize(SizeX + ShadowBorder *2, SizeY + ShadowBorder
*  2);

				//Calculates the cache mode for full-view shadows, including data such as the size and number of
				shadow maps. ComputeWholeSceneShadowCacheModes(
					LightSceneInfo,
					ProjectedShadowInitializer.bOnePassPointLightShadow,
					ViewFamily.CurrentRealTime,
					MaxDesiredResolution,
					FIntPoint(MaxShadowResolution, MaxShadowResolutionY), Scene,

					//The following are the incoming or outgoing parameters, which can be
					changed by the interface internally. ProjectedShadowInitializer,
					ShadowMapSize,
					InOutNumPointShadowCachesUpdatedThisFrame,
					InOutNumSpotShadowCachesUpdatedThisFrame,
					NumShadowMaps,
					CacheMode);

				SizeX = ShadowMapSize.X - ShadowBorder *2; SizeY =
				ShadowMapSize.Y - ShadowBorder *2;
			}

			//createNumShadowMapsshadow maps, and the data of each shadow map is stored in
			FProjectedShadowInfoIn the example. for(int32 CacheModeIndex =0; CacheModeIndex < NumShadowMaps;
CacheModeIndex++)
			{
				//createFProjectedShadowInfoExamples.
				FProjectedShadowInfo* ProjectedShadowInfo = new(FMemStack::Get(),1,
16)FProjectedShadowInfo;
```

```
//set upProjectedShadowInfoPanoramic projection parameters (view frustum bounds, transformation matrix, depth, depth offset wait).
ProjectedShadowInfo->SetupWholeSceneProjection(
	LightSceneInfo,
	NULL,
	ProjectedShadowInitializer, SizeX,

	Size Y,
	ShadowBorder,
	false		// no RSM
	) ;

ProjectedShadowInfo->CacheMode		= CacheMode[CacheModeIndex];
ProjectedShadowInfo->FadeAlphas		=FadeAlphas;

//Adds visible light sources to the list of cast shadows.
VisibleLightInfo.MemStackProjectedShadows.Add(ProjectedShadowInfo);

// Single-pass point light shadows.
if (ProjectedShadowInitializer.bOnePassPointLightShadow)
{
	const staticFVector CubeDirections[6] = {

		FVector(-1,	0, 0),
		FVector(1,	0, 0),
		FVector(0,	- 1,	0),
		FVector(0,	1, 0),
		FVector(0,	0,	- 1),
		FVector(0,	0,	1)
	};

	const staticFVector UpVectors[6] = {

		FVector(0,	1,	0),
		FVector(0,	1,	0),
		FVector(0,	0,	- 1),
		FVector(0,	0,	1),
		FVector(0,	1,	0),
		FVector(0,	1,	0)
	};

	constFLightSceneProxy& LightProxy = *(ProjectedShadowInfo-> GetLightSceneInfo().Proxy);

	constFMatrix FaceProjection = FPerspectiveMatrix(PI /4.0f,1,1, 1, LightProxy.GetRadius());
	constFVector LightPosition = LightProxy.GetPosition();

	ProjectedShadowInfo->OnePassShadowViewMatrices.Empty(6); ProjectedShadowInfo->OnePassShadowViewProjectionMatrices.Empty(6); ProjectedShadowInfo->OnePassShadowFrustums.Empty(6); ProjectedShadowInfo->OnePassShadowFrustums.AddZeroed(6); constFMatrix ScaleMatrix = FScaleMatrix(FVector(1,-1,1));

	//Fill all (6) surface data. ProjectedShadowInfo->CasterFrustum.Planes.Empty(); for(int32 FaceIndex =0; FaceIndex <6; FaceIndex++) {
```

```
                                        //Create a view-projection matrix for each face.
                                        constFMatrix WorldToLightMatrix =
FLookAtMatrix(LightPosition, LightPosition + CubeDirections[FaceIndex], UpVectors[FaceIndex])
* ScaleMatrix;
                                        ProjectedShadowInfo -
> OnePassShadowViewMatrices.Add(WorldToLightMatrix);
                                        constFMatrix ShadowViewProjectionMatrix = WorldToLightMatrix
* FaceProjection;
                                        ProjectedShadowInfo -
> OnePassShadowViewProjectionMatrices.Add(ShadowViewProjectionMatrix);
                                        //Create a convex volume bounding cone for fast clipping of
                                        objects. GetViewFrustumBounds(ProjectedShadowInfo-
> OnePassShadowFrustums[FaceIndex], ShadowViewProjectionMatrix,false);

                                        //  Make sure you have valid cones.
                                        if  (ProjectedShadowInfo-
> OnePassShadowFrustums[FaceIndex].Planes.Num() >0)
                                        {
                                            //Assuming the last face is the far plane,     Must includePreShadowTranslation
                                            FPlane Src = ProjectedShadowInfo-
> OnePassShadowFrustums[FaceIndex].Planes.Last();
                                            // add world space preview translation Src.W += (FVector(Src)
                                            | ProjectedShadowInfo-
> PreShadowTranslation);
                                            ProjectedShadowInfo->CasterFrustum.Planes.Add(Src);
                                        }
                                    }
                                    //Initializes the projection cone.
                                    ProjectedShadowInfo->CasterFrustum.Init();
                                }

                                //  For non-ray-traced distance field shadows, performCPUSide cropping. (!
                                if  ProjectedShadowInfo->bRayTracedDistanceField)
                                {
                                    //Constructs a convex volume of the light source's view to clip shadow casters.
                                    FLightViewFrustumConvexHulls LightViewFrustumConvexHulls; if
                                    (CacheMode[CacheModeIndex] != SDCM_StaticPrimitivesOnly) {

                                        FVectorconst& LightOrigin = LightSceneInfo->Proxy-
> GetOrigin();
                                        BuildLightViewFrustumConvexHulls(LightOrigin, Views,
LightViewFrustumConvexHulls);
                                    }

                                    boolbCastCachedShadowFromMovablePrimitives =
GCachedShadowsCastFromMovablePrimitives || LightSceneInfo->Proxy-
> GetForceCachedShadowsForMovablePrimitives();
                                    if(CacheMode[CacheModeIndex] != SDCM_StaticPrimitivesOnly
                                        && (CacheMode[CacheModeIndex] != SDCM_MovablePrimitivesOnly ||
bCastCachedShadowFromMovablePrimitives))
                                    {
                                        //Add all primitives affected by the light source to the affected primitive list (subject primitive
list).
                                        for(FLightPrimitiveInteraction* Interaction = LightSceneInfo-
> GetDynamicInteractionOftenMovingPrimitiveList(false);
                                            Interaction;
                                            Interaction = Interaction->GetNextPrimitive())
                                        {
```

```
                                        //  Casts shadows and is not self-shadowing (InsetShader)Only the primitives that intersect with the light source are addedsubject
primitive       list.
                                        if    (Interaction->HasShadow()
                                              && !Interaction->CastsSelfShadowOnly() && (!
                                              bStaticSceneOnly || Interaction-
> GetPrimitiveSceneInfo()->Proxy->HasStaticLighting()))
                                        {
                                              FBoxSphereBoundsconst& Bounds = Interaction-
> GetPrimitiveSceneInfo()->Proxy->GetBounds();
                                              if(IntersectsConvexHulls(LightViewFrustumConvexHulls,
Bounds)
                                              {
                                                    ProjectedShadowInfo -
> AddSubjectPrimitive(Interaction->GetPrimitiveSceneInfo(), &Views, FeatureLevel,false);
                                              }
                                        }
                                  }
                            }

                            if(CacheMode[CacheModeIndex] != SDCM_MovablePrimitivesOnly) {

                                  //Add all primitives affected by the shadow casting cone toSubjectPrimitiveList. for
                                  (FLightPrimitiveInteraction* Interaction = LightSceneInfo-
> GetDynamicInteractionStaticPrimitiveList(false);
                                        Interaction;
                                        Interaction = Interaction->GetNextPrimitive())
                                  {
                                        if(Interaction->HasShadow()
                                              && !Interaction->CastsSelfShadowOnly() && (!
                                              bStaticSceneOnly || Interaction-
> GetPrimitiveSceneInfo()->Proxy->HasStaticLighting()))
                                        {
                                              FBoxSphereBoundsconst& Bounds = Interaction-
> GetPrimitiveSceneInfo()->Proxy->GetBounds();
                                              if(IntersectsConvexHulls(LightViewFrustumConvexHulls,
Bounds)
                                              {
                                                    ProjectedShadowInfo -
> AddSubjectPrimitive(Interaction->GetPrimitiveSceneInfo(), &Views, FeatureLevel,false);
                                              }
                                        }
                                  }
                            }
                      }

                      boolbRenderShadow =true;

                      //Whether the cache mode has only static primitives.
                      if(CacheMode[CacheModeIndex] == SDCM_StaticPrimitivesOnly) {

                            const boolbHasStaticPrimitives = ProjectedShadowInfo-
> HasSubjectPrims();
                            //Static shadows do not need to be rendered.
                            bRenderShadow = bHasStaticPrimitives; FCachedShadowMapData&
                            CachedShadowMapData = Scene-
> CachedShadowMaps.FindChecked(ProjectedShadowInfo->GetLightSceneInfo().Id);
                            CachedShadowMapData.bCachedShadowMapHasPrimitives =
bHasStaticPrimitives;
```

```
                        }

                        //  If you need to render shadows, add them toVisibleLightInfo.AllProjectedShadows
                        if  List. (bRenderShadow)
                        {
                            VisibleLightInfo.AllProjectedShadows.Add(ProjectedShadowInfo);
                        }
                    }
                }
            }
        }
}
```

5.6.3.3 AddViewDependentWholeSceneShadowsForView

The following will analyzeAddViewDependentWholeSceneShadowsForView:

```
void FSceneRenderer::AddViewDependentWholeSceneShadowsForView(
    TArray<FProjectedShadowInfo*,    SceneRenderingAllocator>&    ShadowInfos,
    TArray<FProjectedShadowInfo*,    SceneRenderingAllocator>&    ShadowInfosThatNeedCulling,
    FVisibleLightInfo&  VisibleLightInfo,
    FLightSceneInfo& LightSceneInfo)
{
    //Traverse allview,For eachviewcreateviewThe associated full-view shadow.
    for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        FViewInfo& View = Views[ViewIndex];

        const floatLightShadowAmount = LightSceneInfo.Proxy->GetShadowAmount(); TArray<float,
        TInlineAllocator<2> > FadeAlphas; FadeAlphas.Init(0.0f, FadeAlphas[ViewIndex]
                                    Views.Num());
                                    =LightShadowAmount;

        (......)

        //  If it is the main view, handle the cast shadow.
        if  (IStereoRendering::IsAPrimaryView(View))
        {
            const boolbExtraDistanceFieldCascade = LightSceneInfo.Proxy-
> ShouldCreateRayTracedCascade(View.GetFeatureLevel(),
LightSceneInfo.IsPrecomputedLightingValid(), View.MaxShadowCascades);

            //Get the number of projections associated with a view.
            constint32 ProjectionCount = LightSceneInfo.Proxy-
> GetNumViewDependentWholeSceneShadows(View, LightSceneInfo.IsPrecomputedLightingValid())                    +
(bExtraDistanceFieldCascade?1:0);

            FSceneRenderTargets& SceneContext_ConstantsOnly =
FSceneRenderTargets::Get_FrameConstantsOnly();

            //According to the number of projections, create a projection shadow initializer and the
            correspondingFProjectedShadowInfo. for(int32 Index =0; Index < ProjectionCount; Index++) {

                FWholeSceneProjectedShadowInitializer ProjectedShadowInitializer;

                int32 LocalIndex = Index;
```

```
                    // Indexing like this puts the ray traced shadow cascade last (might not
be  needed
                    if(bExtraDistanceFieldCascade && LocalIndex +1== ProjectionCount) {

                        LocalIndex = INDEX_NONE;
                    }

                    if(LightSceneInfo.Proxy-
> GetViewDependentWholeSceneProjectedShadowInitializer(View, LocalIndex,
LightSceneInfo.IsPrecomputedLightingValid(), ProjectedShadowInitializer))
                    {
                        constFIntPointShadowBufferResolution
                        ( FMath::Clamp(GetCachedScalabilityCVars().MaxCSMShadowResolution,                1,
(int32)GMaxShadowDepthBufferSizeX),
                        FMath::Clamp(GetCachedScalabilityCVars().MaxCSMShadowResolution,                1,
(int32)GMaxShadowDepthBufferSizeY));

                        //createFProjectedShadowInfoExamples.
                        FProjectedShadowInfo* ProjectedShadowInfo = new(FMemStack::Get(),1,
16)FProjectedShadowInfo;
                        //Shadow border.
                        uint32 ShadowBorder = NeedsUnatlasedCSMDepthsWorkaround(FeatureLevel)
?0: SHADOW_BORDER;
                        //Set up panoramic projection.
                        ProjectedShadowInfo->SetupWholeSceneProjection(
                            &LightSceneInfo,
                            &View,
                            ProjectedShadowInitializer, ShadowBufferResolution.X -
                            ShadowBorder *2, ShadowBufferResolution.Y -
                            ShadowBorder *2, ShadowBorder,

                            false           // no RSM
                            ) ;

                        ProjectedShadowInfo->FadeAlphas = FadeAlphas;

                        //WillProjectedShadowInfoAdded to the relevant list of visible light source
                        information. FVisibleLightInfo& LightViewInfo =
VisibleLightInfos[LightSceneInfo.Id];
                        VisibleLightInfo.MemStackProjectedShadows.Add(ProjectedShadowInfo);
                        VisibleLightInfo.AllProjectedShadows.Add(ProjectedShadowInfo);
                        ShadowInfos.Add(ProjectedShadowInfo);

                        //  Add to the list of shadows to be clipped.
                        if  (!ProjectedShadowInfo->bRayTracedDistanceField)
                        {
                            ShadowInfosThatNeedCulling.Add(ProjectedShadowInfo);
                        }
                    }
                }

                //deal withRSM (Refletive Shadow Map),Used forLPVillumination.
                FSceneViewState* ViewState = (FSceneViewState*)View.State; if(ViewState)

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

```
    }
}
```

5.6.3.4 SetupInteractionShadows

Then continue the analysisSetupInteractionShadows:

```
void FSceneRenderer::SetupInteractionShadows(
    FRHICommandListImmediate& RHICmdList,
    FLightPrimitiveInteraction*        Interaction,
    FVisibleLightInfo&        VisibleLightInfo,
    bool bStaticSceneOnly,
    const TArray<FProjectedShadowInfo*,SceneRenderingAllocator>&
ViewDependentWholeSceneShadows,
    TArray<FProjectedShadowInfo*,SceneRenderingAllocator>&        PreShadows)
{
    FPrimitiveSceneInfo* PrimitiveSceneInfo = Interaction->GetPrimitiveSceneInfo(); FLightSceneProxy*
    LightProxy = Interaction->GetLight()->Proxy;
    externboolGUseTranslucencyShadowDepths;

    boolbShadowHandledByParent =false;

    //Process the additional root component of the light source. If there is an additional root node, set
    bShadowHandledByParentfortrue. if(PrimitiveSceneInfo->LightingAttachmentRoot.IsValid()) {

        FAttachmentGroupSceneInfo& AttachmentGroup = Scene-
> AttachmentGroups.FindChecked(PrimitiveSceneInfo->LightingAttachmentRoot);
        bShadowHandledByParent = AttachmentGroup.ParentSceneInfo &&
AttachmentGroup.ParentSceneInfo->Proxy->LightAttachmentsAsGroup();
    }

    //  bShadowHandledByParentThe shadow will be processed by the parent node
    if  component. (!bShadowHandledByParent)
    {
        const boolbCreateTranslucentObjectShadow = GUseTranslucencyShadowDepths && Interaction-
        >HasTranslucentObjectShadow();
        const boolbCreateInsetObjectShadow = Interaction->HasInsetObjectShadow(); const bool
        bCreateObjectShadowForStationaryLight = ShouldCreateObjectShadowForStationaryLight(Interaction-
>GetLight(), PrimitiveSceneInfo-
> Proxy, Interaction->IsShadowMapped());

        if(Interaction->HasShadow()
            && (!bStaticSceneOnly || PrimitiveSceneInfo->Proxy->HasStaticLighting()) &&
            (bCreateTranslucentObjectShadow || bCreateInsetObjectShadow ||
bCreateObjectShadowForStationaryLight))
        {
            //Create per-object shadows.
            CreatePerObjectProjectedShadow(RHICmdList, Interaction,
bCreateTranslucentObjectShadow, bCreateInsetObjectShadow || bCreateObjectShadowForStationaryLight,
ViewDependentWholeSceneShadows, PreShadows);
        }
    }
}
```

5.6.3.5 CreatePerObjectProjectedShadow

The above code involvesCreatePerObjectProjectedShadowentering its implementation code analysis:

```
void FSceneRenderer::CreatePerObjectProjectedShadow(
        FRHICommandListImmediate& RHICmdList,
        FLightPrimitiveInteraction* bool        Interaction,
                bCreateTranslucentObjectShadow,
        bool bCreateOpaqueObjectShadow,
        const TArray<FProjectedShadowInfo*,SceneRenderingAllocator>&
 ViewDependentWholeSceneShadows,
         TArray<FProjectedShadowInfo*,          SceneRenderingAllocator>& OutPreShadows)
{
        FPrimitiveSceneInfo* PrimitiveSceneInfo = Interaction->GetPrimitiveSceneInfo(); constint32 PrimitiveId =
        PrimitiveSceneInfo->GetIndex();

        FLightSceneInfo* LightSceneInfo = Interaction->GetLight(); FVisibleLightInfo& VisibleLightInfo =
        VisibleLightInfos[LightSceneInfo->Id];

        //Check if the shadow is in anyviewVisible inside.
        boolbShadowIsPotentiallyVisibleNextFrame =false; bool
        bOpaqueShadowIsVisibleThisFrame =false; boolbSubjectIsVisible =
        false; boolbOpaque =false;

        boolbTranslucentRelevance =false;
        boolbTranslucentShadowIsVisibleThisFrame =false;
        int32 NumBufferedFrames = FOcclusionQueryHelpers::GetNumBufferedFrames(FeatureLevel);

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

                constFViewInfo& View = Views[ViewIndex];

                //Get    the    primitive    cacheViewRelevance.
                FPrimitiveViewRelevance          ViewRelevance      =
                View.PrimitiveViewRelevanceMap[PrimitiveId];

                if(!ViewRelevance.bInitializedThisFrame) {

                        // Compute the subject primitive's view relevance since it wasn't cached ViewRelevance =
                        PrimitiveSceneInfo->Proxy->GetViewRelevance(&View);
                }

                // Check if the subject primitive is shadow relevant.
                const boolbPrimitiveIsShadowRelevant = ViewRelevance.bShadowRelevance;

                //The opaque object shadow key values are:id,The address of the light source component, the index of the shadow after
                splitting, and whether the shadow is transparent. constFSceneViewState::FProjectedShadowKeyOpaqueKey(PrimitiveSceneInfo-
> PrimitiveComponentId, LightSceneInfo->Proxy->GetLightComponent(), INDEX_NONE,false);

                //Detects whether opaque objects' shadows or pre-shadows are blocked.
                 const boolbOpaqueShadowIsOccluded =
                        !bCreateOpaqueObjectShadow || (

                                !View.bIgnoreExistingQueries ((FSceneV&ie&wStateV*ie)Vwe&Swta.Stteate)-
                                >IsShadowOccluded(RHICmdList,                                    OpaqueKey,
NumBufferedFrames)
                        ) ;

                //Transparent object shadow sorting key value.
```

```
            constFSceneViewState::FProjectedShadowKeyTranslucentKey(PrimitiveSceneInfo-
> PrimitiveComponentId, LightSceneInfo->Proxy->GetLightComponent(), INDEX_NONE,true);

                //Detects whether the transparent object shadow or pre-shadow is blocked.
                 const boolbTranslucentShadowIsOccluded =
                        !bCreateTranslucentObjectShadow || (

                            !View.bIgnoreExistingQueries && View.State && ((FSceneViewState*)View.State)-
                            >IsShadowOccluded(RHICmdList,
TranslucentKey, NumBufferedFrames)
                        ) ;

                //    Ignore the absence of the LordPassThe primitive to render.
                if    (PrimitiveSceneInfo->Proxy->ShouldRenderInMainPass())
                {
                     const boolbSubjectIsVisibleInThisView =
View.PrimitiveVisibilityMap[PrimitiveSceneInfo->GetIndex()];
                     bSubjectIsVisible |= bSubjectIsVisibleInThisView;
                }

                //A shadow is considered visible if it is associated with a view and is not obscured.
                bOpaqueShadowIsVisibleThisFrame |= (bPrimitiveIsShadowRelevant && !
                bOpaqueShadowIsOccluded);
                bTranslucentShadowIsVisibleThisFrame |= (bPrimitiveIsShadowRelevant && !
                bTranslucentShadowIsOccluded);
                bShadowIsPotentiallyVisibleNextFrame |= bPrimitiveIsShadowRelevant; bOpaque |=
                ViewRelevance.bOpaque;
                bTranslucentRelevance |= ViewRelevance.HasTranslucency(); }// for


        //If it is not visible in this frame and not potentially visible in the next frame, return directly to skip subsequent shadow creation and setting.
        if(!bOpaqueShadowIsVisibleThisFrame && !bTranslucentShadowIsVisibleThisFrame && !
        bShadowIsPotentiallyVisibleNextFrame)
        {
                return;
        }

        //Collect shadow group primitives.
        TArray<FPrimitiveSceneInfo*, SceneRenderingAllocator> ShadowGroupPrimitives; PrimitiveSceneInfo-
        >GatherLightingAttachmentGroupPrimitives(ShadowGroupPrimitives);

#if    ENABLE_NAN_DIAGNOSTIC
        //There is no valid shadow group primitive, so just return directly.
        if(ShadowGroupPrimitives.Num() ==0) {


                return;
        }
#endif

        //Compute the combined bounding box of the group of shadow primitives.
        FBoxSphereBounds OriginalBounds = ShadowGroupPrimitives[0]->Proxy->GetBounds(); //Fix illegal
        bounding boxes.
        if(!ensureMsgf(OriginalBounds.ContainsNaN() ==false, TEXT("OriginalBound contains NaN : %s"),
        *OriginalBounds.ToString()))
        {
                OriginalBounds = FBoxSphereBounds(FVector::ZeroVector, FVector(1.f),1.f);
        }
        for(int32 ChildIndex =1; ChildIndex < ShadowGroupPrimitives.Num(); ChildIndex++)
```

```
{
    constFPrimitiveSceneInfo* ShadowChild = ShadowGroupPrimitives[ChildIndex]; if(ShadowChild-
    >Proxy->CastsDynamicShadow()) {

        FBoxSphereBounds ChildBound = ShadowChild->Proxy->GetBounds();
        OriginalBounds = OriginalBounds + ChildBound;

        if(!ensureMsgf(OriginalBounds.ContainsNaN() ==false, TEXT("Child %s contains
NaN: %s"), *ShadowChild->Proxy->GetOwnerName().ToString(), *ChildBound.ToString()))
        {
            // fix up OriginalBounds. This is going to cause flickers OriginalBounds =
            FBoxSphereBounds(FVector::ZeroVector, FVector(1.f),1.f);
        }
    }
}

//The following code andCreateWholeSceneProjectedShadowThey are quite similar, so the analysis will be
omitted... FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

//Shadow constant.
constuint32 MaxShadowResolutionSetting =
GetCachedScalabilityCVars().MaxShadowResolution;
constFIntPoint ShadowBufferResolution =
SceneContext.GetShadowDepthTextureResolution();
constuint32 MaxShadowResolution = FMath::Min<int32>(MaxShadowResolutionSetting,
ShadowBufferResolution.X) - SHADOW_BORDER *2;
constuint32 MaxShadowResolutionY = FMath::Min<int32>(MaxShadowResolutionSetting,
ShadowBufferResolution.Y) - SHADOW_BORDER *2;
constuint32 MinShadowResolution                    = FMath::Max<int32>(0,
CVarMinShadowResolution.GetValueOnRenderThread());
constuint32 ShadowFadeResolution = FMath::Max<int32>(0,
CVarShadowFadeResolution.GetValueOnRenderThread());
constuint32 MinPreShadowResolution = FMath::Max<int32>(0,
CVarMinPreShadowResolution.GetValueOnRenderThread());
constuint32 PreShadowFadeResolution = FMath::Max<int32>(0,
CVarPreShadowFadeResolution.GetValueOnRenderThread());

//Maximum shadow resolution.
uint32 MaxDesiredResolution =0; float
MaxScreenPercent =0;
TArray<float, TInlineAllocator<2> > ResolutionFadeAlphas; TArray<float, TInlineAllocator<
2> > ResolutionPreShadowFadeAlphas; floatMaxResolutionFadeAlpha =0; float
MaxResolutionPreShadowFadeAlpha =0;



(.....)

FBoxSphereBounds Bounds = OriginalBounds;

//Whether to render pre-shadows (shadow cache).
const boolbRenderPreShadow =
    CVarAllowPreshadows.GetValueOnRenderThread() &&
        LightSceneInfo->Proxy->HasStaticShadowing()
    && bSubjectIsVisible
    && (!PrimitiveSceneInfo->Proxy->HasStaticLighting() || !Interaction-
> IsShadowMapped()
    && !(PrimitiveSceneInfo->Proxy->UseSingleSampleShadowFromStationaryLights() LightSceneInfo-          &&
>Proxy->GetLightType() == LightType_Directional);
```

```
//If pre-shadows need to be rendered, enlarge the bounding box to improve cache
utilization. if(bRenderPreShadow && ShouldUseCachePreshadows()) {

    floatPreshadowExpandFraction =
    FMath::Max(CVarPreshadowExpandFraction.GetValueOnRenderThread(),      0.0f);
    Bounds.SphereRadius += (Bounds.BoxExtent * PreshadowExpandFraction).Size(); Bounds.BoxExtent
    *= PreshadowExpandFraction +1.0f;
}

//Shadow initializer.
FPerObjectProjectedShadowInitializer ShadowInitializer;

if((MaxResolutionFadeAlpha >1.0f/256.0f|| (bRenderPreShadow &&
MaxResolutionPreShadowFadeAlpha >1.0f/256.0f))
    && LightSceneInfo->Proxy->GetPerObjectProjectedShadowInitializer(Bounds, ShadowInitializer))

{
    const floatMaxFadeAlpha = MaxResolutionFadeAlpha;

    //Shadows that have not been completely transitioned need to create shadow casting instances.
    if(CVarAllowPerObjectShadows.GetValueOnRenderThread() && MaxFadeAlpha >1.0f/
256.0f)
    {
        constint32 SizeX = MaxDesiredResolution >= MaxShadowResolution ?
MaxShadowResolution : (1<< (FMath::CeilLogTwo(MaxDesiredResolution) -1));

        if(bOpaque && bCreateOpaqueObjectShadow && (bOpaqueShadowIsVisibleThisFrame
||      bShadowIsPotentiallyVisibleNextFrame))
        {
            //createFProjectedShadowInfoExamples.
            FProjectedShadowInfo* ProjectedShadowInfo = new(FMemStack::Get(),1,16)
FProjectedShadowInfo;

            if(ProjectedShadowInfo->SetupPerObjectProjection(
                LightSceneInfo,
                PrimitiveSceneInfo,
                ShadowInitializer,
                false,                          // no preshadow
                SizeX,
                MaxShadowResolutionY,
                SHADOW_BORDER,

                MaxScreenPercent,
                false))                          // no translucent shadow
            {
                ProjectedShadowInfo->bPerObjectOpaqueShadow Projected=Strhuaed;owInfo->FadeAlphas =
                ResolutionFadeAlphas;
                VisibleLightInfo.MemStackProjectedShadows.Add(ProjectedShadowInfo);

                //   WillProjectedShadowInfoAdd to the corresponding
                if   list. (bOpaqueShadowIsVisibleThisFrame)
                {
                    VisibleLightInfo.AllProjectedShadows.Add(ProjectedShadowInfo);

                    for(int32 ChildIndex =0, ChildCount =
ShadowGroupPrimitives.Num(); ChildIndex < ChildCount; ChildIndex++)
                    {
                        FPrimitiveSceneInfo* ShadowChild =
```

```
ShadowGroupPrimitives[ChildIndex];
                                    ProjectedShadowInfo->AddSubjectPrimitive(ShadowChild,          &Views,
FeatureLevel,false);
                                }
                            }
                            else if(bShadowIsPotentiallyVisibleNextFrame) {


VisibleLightInfo.OccludedPerObjectShadows.Add(ProjectedShadowInfo);
                            }
                        }
                    }

                    //  Shadows from semi-transparent objects.
                    if  (bTranslucentRelevance
                        && Scene->GetFeatureLevel() >= ERHIFeatureLevel::SM5 &&
                            bCreateTranslucentObjectShadow
                        &&(bTranslucentShadowIsVisibleThisFrame ||
bShadowIsPotentiallyVisibleNextFrame))
                    {
                        (......)
                    }
                }

                const floatMaxPreFadeAlpha = MaxResolutionPreShadowFadeAlpha;

                //Handles effective pre-shadowing.
                if(MaxPreFadeAlpha >1.0f/256.0f
                    && bRenderPreShadow
                    && bOpaque
                    && Scene->GetFeatureLevel() >= ERHIFeatureLevel::SM5)
                {
                    int32 PreshadowSizeX =1<<
(FMath::CeilLogTwo(FMath::TruncToInt(MaxDesiredResolution                *
CVarPreShadowResolutionFactor.GetValueOnRenderThread()))                 -  1);

                    constFIntPoint PreshadowCacheResolution =
SceneContext.GetPreShadowCacheTextureResolution();

                    //Check if it is within the full shadow.
                    boolbIsOutsideWholeSceneShadow =true;
                    for(int32 i =0; i < ViewDependentWholeSceneShadows.Num(); i++) {

                        constFProjectedShadowInfo* WholeSceneShadow =
ViewDependentWholeSceneShadows[i];
                        constFVector2D DistanceFadeValues = WholeSceneShadow-
> GetLightSceneInfo().Proxy->GetDirectionalLightDistanceFadeParameters(Scene-
> GetFeatureLevel(), WholeSceneShadow->GetLightSceneInfo().IsPrecomputedLightingValid(), WholeSceneShadow-
>DependentView->MaxShadowCascades);
                        const floatDistanceFromShadowCenterSquared = (WholeSceneShadow-
> ShadowBounds.Center - Bounds.Origin).SizeSquared();
                        const floatDistanceFromViewSquared = ((FVector)WholeSceneShadow-
> DependentView->ShadowViewMatrices.GetViewOrigin()          - Bounds.Origin).SizeSquared();
                        //  ifpreshadowIf the spherical bounding box of is within the near transition distance, it means it is within the
                        if  full-view shadow. (DistanceFromShadowCenterSquared <
FMath::Square(FMath::Max(WholeSceneShadow->ShadowBounds.W - Bounds.SphereRadius,0.0f))
                            && DistanceFromViewSquared <
 FMath::Square(FMath::Max(DistanceFadeValues.X -200.0f- Bounds.SphereRadius,0.0f)))
```

```
                {
                    bIsOutsideWholeSceneShadow = false;
                    break;
                }
            }

            // Only partial shadow casters create opacity outside of full-view shadows
            if preshadow. (bIsOutsideWholeSceneShadow)
            {
                //Try to reuse from cachepreshadow.
                TRefCountPtr<FProjectedShadowInfo> ProjectedPreShadowInfo =
GetCachedPreshadow(Interaction, ShadowInitializer, OriginalBounds, PreshadowSizeX);

                boolbOk =true;

                //Creation and SetupProjectedPreShadowInfo.
                if(!ProjectedPreShadowInfo) {

                    ProjectedPreShadowInfo = new FProjectedShadowInfo;

                    b = ProjectedPreShadowInfo-
                    >SetupPerObjectProjection( LightSceneInfo,
                    PrimitiveSceneInfo,
                    ShadowInitializer,
                    true,                       // preshadow
                    PreshadowSizeX,
                    FMath::TruncToInt(MaxShadowResolutionY *
CVarPreShadowResolutionFactor.GetValueOnRenderThread()),
                    SHADOW_BORDER,
                    MaxScreenPercent,
                    false                       // not translucent shadow
                    ) ;
                }

                // Continue to set the validProjectedPreShadowInfoOther data ofVisibleLightInfoRelated
List.
                if (bOk)
                {
                    ProjectedPreShadowInfo->FadeAlphas = ResolutionPreShadowFadeAlphas;

                    VisibleLightInfo.AllProjectedShadows.Add(ProjectedPreShadowInfo);
                    VisibleLightInfo.ProjectedPreShadows.Add(ProjectedPreShadowInfo);

                    //ifpreshadowIf there is no depth buffer, add itOutPreShadowsList.OutPreShadowsuse
to generate only the information needed to render the shadow map.
                    if(!ProjectedPreShadowInfo->bDepthsCached && ProjectedPreShadowInfo-
> CasterFrustum.PermutedPlanes.Num())
                    {
                        OutPreShadows.Add(ProjectedPreShadowInfo);
                    }

                    //Add all visible primitives to the list of receiving primitives for the shadow.
                    for(int32 ChildIndex =0; ChildIndex < ShadowGroupPrimitives.Num();
ChildIndex++)
                    {
                        FPrimitiveSceneInfo* ShadowChild =
ShadowGroupPrimitives[ChildIndex];
                        boolbChildIsVisibleInAnyView =false;
```

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

                            constFViewInfo& View = Views[ViewIndex];
                            if(View.PrimitiveVisibilityMap[ShadowChild->GetIndex()]) {

                                bChildIsVisibleInAnyView =true;
                                break;
                            }
                        }
                        if(bChildIsVisibleInAnyView) {

                            ProjectedPreShadowInfo->AddReceiverPrimitive(ShadowChild);
                        }
                    }
                }
            }
        }
    }
}
```

5.6.3.6 InitProjectedShadowVisibility

The visibility of the shadow is initialized by InitProjectedShadowVisibility , and its partial code and analysis are as follows:

```
void FSceneRenderer::InitProjectedShadowVisibility(FRHICommandListImmediate& RHICmdList) {

    int32 NumBufferedFrames = FOcclusionQueryHelpers::GetNumBufferedFrames(FeatureLevel);

    //Traverse all light sources in the scene and initialize the viewProjectedShadowVisibilityMaps,Remove shadows that have no
    host primitives. for(TSparseArray<FLightSceneInfoCompact>::TConstIterator LightIt(Scene-
>Lights);LightIt;++LightIt) {

        FVisibleLightInfo& VisibleLightInfo = VisibleLightInfos[LightIt.GetIndex()];

        //Assigns cast shadow visibility within a view and associated container.
        for(int32 ViewIndex =0;ViewIndex < Views.Num();ViewIndex++) {

            FViewInfo& View = Views[ViewIndex]; FVisibleLightViewInfo&
            VisibleLightViewInfo =
View.VisibleLightInfos[LightIt.GetIndex()];

VisibleLightViewInfo.ProjectedShadowVisibilityMap.Init(false,VisibleLightInfo.AllProjected Shadows.Num());


VisibleLightViewInfo.ProjectedShadowViewRelevanceMap.Empty(VisibleLightInfo.AllProjectedShadows.Num());


VisibleLightViewInfo.ProjectedShadowViewRelevanceMap.AddZeroed(VisibleLightInfo.AllProject edShadows.Num());

        }

        //Iterates over all cast shadow instances of visible light sources.
        for(int32 ShadowIndex=0; ShadowIndex<VisibleLightInfo.AllProjectedShadows.Num(); ShadowIndex++ )

        {
```

```
                FProjectedShadowInfo& ProjectedShadowInfo =
* VisibleLightInfo.AllProjectedShadows[ShadowIndex];

                //Save the shadow index.
                ProjectedShadowInfo.ShadowId = ShadowIndex;

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

                    FViewInfo& View = Views[ViewIndex];

                    //Handles shadows associated with a view.
                    if(ProjectedShadowInfo.DependentView && ProjectedShadowInfo.DependentView
!=   &View)
                    {
                        (.....)
                    }

                    FVisibleLightViewInfo& VisibleLightViewInfo =
View.VisibleLightInfos[LightIt.GetIndex()];

                    //Make sure the light source is within the viewing frustum.
                    if(VisibleLightViewInfo.bInViewFrustum) {

                        //Computes view-related data for a host entity.
                        FPrimitiveViewRelevance ViewRelevance; if
                        (ProjectedShadowInfo.GetParentSceneInfo()) {

                            ViewRelevance = ProjectedShadowInfo.GetParentSceneInfo()->Proxy-
> GetViewRelevance(&View);
                        }
                        else
                        {
                            ViewRelevance.bDrawRelevance = ViewRelevance.bStaticRelevance =
ViewRelevance.bDynamicRelevance = ViewRelevance.bShadowRelevance =true;
                        }
                        VisibleLightViewInfo.ProjectedShadowViewRelevanceMap[ShadowIndex] =
ViewRelevance;

                        // Check if the subject primitive's shadow is view relevant. const bool
                        bPrimitiveIsShadowRelevant =
ViewRelevance.bShadowRelevance;

                        //Determines whether the shadow is blocked.
                        boolbShadowIsOccluded =false;
                        if(!View.bIgnoreExistingQueries && View.State) {

                            // Check if the shadow is occluded.
                            bShadowIsOccluded =
                                ((FSceneViewState*)View.State)->IsShadowOccluded( RHICmdList,

                                FSceneViewState::FProjectedShadowKey(ProjectedShadowInfo),
                                NumBufferedFrames
                                ) ;
                        }

                        //If the visibility condition is met, then the flag is set
                        if(bPrimitiveIsShadowRelevant && !bShadowIsOccluded) {
```

```
                                VisibleLightViewInfo.ProjectedShadowVisibilityMap[ShadowIndex] =
true;
                        }

                        //If the shadow is visible and notRSMShadow, then draw the shadow cone.
                        if(bPrimitiveIsShadowRelevant && !bShadowIsOccluded &&
!ProjectedShadowInfo.bReflectiveShadowmap)
                        {
                            boolbDrawPreshadowFrustum =
CVarDrawPreshadowFrustum.GetValueOnRenderThread() !=0;

                            // drawpreshadowcone.
                            if ((ViewFamily.EngineShowFlags.ShadowFrustums)
                                && ((bDrawPreshadowFrustum && ProjectedShadowInfo.bPreShadow)
|| (!bDrawPreshadowFrustum && !ProjectedShadowInfo.bPreShadow)))
                            {
                                //Draw the shadow conePDI.
                                FViewElementPDIShadowFrustumPDI(&Views[ViewIndex], nullptr,
&Views[ViewIndex].DynamicPrimitiveShaderData);

                                //Panoramic parallel shadows require drawing cones.
                                if(ProjectedShadowInfo.IsWholeSceneDirectionalShadow()) {

                                    // Get split color FColor Color =
                                    FColor::White;

switch(ProjectedShadowInfo.CascadeSettings.ShadowSplitIndex)
                                    {
                                        case0: Color = FColor::Red;break; case1: Color =
                                        FColor::Yellow;break; case2: Color = FColor::Green;
                                        break; case3: Color = FColor::Blue;break;


                                    }

                                    constFMatrix ViewMatrix =
View.ViewMatrices.GetViewMatrix();
                                    constFMatrix ProjectionMatrix =
View.ViewMatrices.GetProjectionMatrix();
                                    constFVector4 ViewOrigin =
View.ViewMatrices.GetViewOrigin();

                                    floatAspectRatio = ProjectionMatrix.M[1][1] /
ProjectionMatrix.M[0][0];
                                    floatActualFOV = (ViewOrigin.W >0.0f) ? FMath::Atan(1.0f
/ProjectionMatrix.M[0][0]) : PI/4.0f;

                                    floatNear =
ProjectedShadowInfo.CascadeSettings.SplitNear;
                                    floatMid =
ProjectedShadowInfo.CascadeSettings.FadePlaneOffset;
                                    floatFar = ProjectedShadowInfo.CascadeSettings.SplitFar;

                                    //Camera sub-cone.
                                    DrawFrustumWireframe(&ShadowFrustumPDI, (ViewMatrix *
FPerspectiveMatrix(ActualFOV, AspectRatio,1.0f, Near, Mid)).Inverse(), Color,0);
                                    DrawFrustumWireframe(&ShadowFrustumPDI, (ViewMatrix *
FPerspectiveMatrix(ActualFOV, AspectRatio,1.0f, Mid, Far)).Inverse(), FColor::White,0);
```

```
                                        //Shadow map projection bounding box.
                                        DrawFrustumWireframe(&ShadowFrustumPDI,
ProjectedShadowInfo.SubjectAndReceiverMatrix.Inverse() * FTranslationMatrix(-
ProjectedShadowInfo.PreShadowTranslation), Color,0);
                                    }
                                    else//Non-full-view shadow, direct callProjectedShadowInfoThe drawing interface. {


ProjectedShadowInfo.RenderFrustumWireframe(&ShadowFrustumPDI);
                                    }
                                }
                            }
                        }
                    }
                }

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

The above calls are not drawn immediately, butDrawFrustumWireframegenerated and then saved in the view so that they can be actually drawn in the shadow rendering stage later.

RenderFrustumWireframe FViewElementPDI FBatchedElements

5.6.3.7 UpdatePreshadowCache

Let's continue to analyze the initialization phaseUpdatePreshadowCache:

```
void FSceneRenderer::UpdatePreshadowCache(FSceneRenderTargets& SceneContext) {

    if(ShouldUseCachePreshadows() && !Views[0].bIsSceneCapture) {

        //Initialize texture layout.
        if(Scene->PreshadowCacheLayout.GetSizeX() ==0) {

            constFIntPoint PreshadowCacheBufferSize =
SceneContext.GetPreShadowCacheTextureResolution();
            Scene->PreshadowCacheLayout = FTextureLayout(1,1, PreshadowCacheBufferSize.X,
PreshadowCacheBufferSize.Y,false, ETextureLayoutAspectRatio::None,false);
        }


        //Iterate over all cached pre-shadows, deleting instances that are not rendered in this frame.
        for(int32 CachedShadowIndex = Scene->CachedPreshadows.Num() -1;
CachedShadowIndex >=0; CachedShadowIndex--)
        {
            TRefCountPtr<FProjectedShadowInfo> CachedShadow = Scene-
> CachedPreshadows[CachedShadowIndex];
            boolbShadowBeingRenderedThisFrame =false;

            for(int32 LightIndex =0; LightIndex < VisibleLightInfos.Num() &&
!bShadowBeingRenderedThisFrame; LightIndex++)
            {
                bShadowBeingRenderedThisFrame =
VisibleLightInfos[LightIndex].ProjectedPreShadows.Find(CachedShadow) != INDEX_NONE;
            }
```

```
                if(!bShadowBeingRenderedThisFrame) {

                    Scene->CachedPreshadows.RemoveAt(CachedShadowIndex);
                }
            }

            TArray<TRefCountPtr<FProjectedShadowInfo>, SceneRenderingAllocator>
            UncachedPreShadows;

            //Collection can be cachedpreshadowList.
            for(int32 LightIndex =0; LightIndex < VisibleLightInfos.Num(); LightIndex++) {

                for(int32 ShadowIndex =0; ShadowIndex <
VisibleLightInfos[LightIndex].ProjectedPreShadows.Num(); ShadowIndex++)
                {
                    TRefCountPtr<FProjectedShadowInfo> CurrentShadow =
VisibleLightInfos[LightIndex].ProjectedPreShadows[ShadowIndex];
                    checkSlow(CurrentShadow->bPreShadow);

                    if(!CurrentShadow->bAllocatedInPreshadowCache) {

                        UncachedPreShadows.Add(CurrentShadow);
                    }
                }
            }

            //rightpreshadowSort from largest to smallest, assuming the largerpreshadowThere are more objects when
            rendering in depth. UncachedPreShadows.Sort(FComparePreshadows());

            for(int32 ShadowIndex =0; ShadowIndex < UncachedPreShadows.Num(); ShadowIndex++) {

                TRefCountPtr<FProjectedShadowInfo> CurrentShadow =
UncachedPreShadows[ShadowIndex];
                //Try givingpreshadowFind the space, and if found, set the relevant data.
                if(Scene->PreshadowCacheLayout.AddElement(CurrentShadow->X, CurrentShadow->Y,
CurrentShadow->ResolutionX + CurrentShadow->BorderSize *2, CurrentShadow->ResolutionY + CurrentShadow-
>BorderSize *2))
                {
                    CurrentShadow->bAllocatedInPreshadowCache =true;
                    CurrentShadow->bAllocated =true; Scene-
                    >CachedPreshadows.Add(CurrentShadow);
                }
            }
        }
}
```

5.6.3.8 GatherShadowPrimitives

Next, analyze the initialization phaseGatherShadowPrimitives:

```
void FSceneRenderer::GatherShadowPrimitives(constTArray<FProjectedShadowInfo*,
SceneRenderingAllocator>& PreShadows,constTArray<FProjectedShadowInfo*, SceneRenderingAllocator>&
ViewDependentWholeSceneShadows,boolbStaticSceneOnly) {


    //Pre-shadowing or view-dependent full-view shadowing is present.
    if(PreShadows.Num() || ViewDependentWholeSceneShadows.Num()) {
```

```
TArray<FGatherShadowPrimitivesPacket*,SceneRenderingAllocator> Packets;

// Octree traversal clipping shadows.
if (GUseOctreeForShadowCulling)
{
    Packets.Reserve(100);

    //Find primitives that intersect the shadow cone located in the octree.
    for(FScenePrimitiveOctree::TConstIterator<SceneRenderingAllocator>
PrimitiveOctreeIt(Scene->PrimitiveOctree); PrimitiveOctreeIt.HasPendingNodes();
PrimitiveOctreeIt.Advance())
    {
        constFScenePrimitiveOctree::FNode& PrimitiveOctreeNode =
PrimitiveOctreeIt.GetCurrentNode();
        constFOctreeNodeContext& PrimitiveOctreeNodeContext =
PrimitiveOctreeIt.GetCurrentContext();

        {
            //Find the child nodes of an octree node that may contain the
            associated primitive. FOREACH_OCTREE_CHILD_NODE(ChildRef) {

                if(PrimitiveOctreeNode.HasChild(ChildRef)) {

                    //Checks if a child node is inside at least one shadow.
                    constFOctreeNodeContext ChildContext =
PrimitiveOctreeNodeContext.GetChildContext(ChildRef);
                    boolbIsInFrustum =false;

                    if(!bIsInFrustum)
                    {
                        //Traverse allpreshadow,      Determine whether the child node ispreshadowintersect.
                        for(int32 ShadowIndex =0, Num = PreShadows.Num();
ShadowIndex < Num; ShadowIndex++)
                        {
                            FProjectedShadowInfo* ProjectedShadowInfo          =
PreShadows[ShadowIndex];
                            //Checks whether the primitive is inside the shadow cone.
                            if(ProjectedShadowInfo-
> CasterFrustum.IntersectBox(ChildContext.Bounds.Center          +ProjectedShadowInfo-
> PreShadowTranslation, ChildContext.Bounds.Extent))
                            {
                                bIsInFrustum =      true;
                                break;
                            }
                        }
                    }

                    // If it is not inside the cone, perform an intersection check between the bounding box of the child node and the panoramic shadow associated with the view.
Test.
                    if (!bIsInFrustum)
                    {
                        for(int32 ShadowIndex =0, Num =
ViewDependentWholeSceneShadows.Num(); ShadowIndex < Num; ShadowIndex++)
                        {
                            FProjectedShadowInfo* ProjectedShadowInfo =
ViewDependentWholeSceneShadows[ShadowIndex];

                            //check(ProjectedShadowInfo-
```

```
> CasterFrustum.PermutedPlanes.Num());
                                        // Check if this primitive is in the shadow's frustum. if
                                        (ProjectedShadowInfo->CasterFrustum.IntersectBox(
                                            ChildContext.Bounds.Center + ProjectedShadowInfo-
> PreShadowTranslation,
                                            ChildContext.Bounds.Extent ) )

                                        {
                                            bIsInFrustum =true; break;

                                        }
                                    }
                                }

                                    //If a primitive intersects at least one shadow, it is added to the pending node stack of the primitive octree iterator.
(pending    node  stack)middle.
                                    if(bIsInFrustum)
                                    {
                                        PrimitiveOctreeIt.PushChild(ChildRef);
                                    }
                                }
                            }  //    FOREACH_OCTREE_CHILD_NODE
                        }

                        if(PrimitiveOctreeNode.GetElementCount() >0) {

                            FGatherShadowPrimitivesPacket* Packet = new(FMemStack::Get())
FGatherShadowPrimitivesPacket(Scene, Views, &PrimitiveOctreeNode,0,0, PreShadows,
ViewDependentWholeSceneShadows, FeatureLevel, bStaticSceneOnly);
                            Packets.Add(Packet);
                        }
                    }// for
                }
                else//Non-octree traversal.
                {
                    constint32 PacketSize =
CVarParallelGatherNumPrimitivesPerPacket.GetValueOnRenderThread();
                    constint32 NumPackets = FMath::DivideAndRoundUp(Scene->Primitives.Num(),
PacketSize);

                    Packets.Reserve(NumPackets);

                    //In non-octree mode, directly traverse all primitives linearly and add the same number ofFGatherShadowPrimitivesPacketReality
example.
                    for(int32 PacketIndex =0; PacketIndex < NumPackets; PacketIndex++) {

                        constint32 StartPrimitiveIndex = PacketIndex * PacketSize;
                        constint32 NumPrimitives = FMath::Min(PacketSize, Scene->Primitives.Num()
- StartPrimitiveIndex);
                        FGatherShadowPrimitivesPacket* Packet = new(FMemStack::Get())
FGatherShadowPrimitivesPacket(Scene, Views,NULL, StartPrimitiveIndex, NumPrimitives, PreShadows,
ViewDependentWholeSceneShadows, FeatureLevel, bStaticSceneOnly);
                        Packets.Add(Packet);
                    }
                }

                //Calling ParallelForFilter out primitives that do not intersect
                with the shadow ParallelFor(Packets.Num(),
```

```
            [&Packets](int32 Index) {

                Packets[Index]->AnyThreadTask();
            },
            !(FApp::ShouldUseThreadingForPerformance()                &&
CVarParallelGatherShadowPrimitives.GetValueOnRenderThread() >0) );

        //Release resources.
        for(int32 PacketIndex =0; PacketIndex < Packets.Num(); PacketIndex++) {

            FGatherShadowPrimitivesPacket* Packet = Packets[PacketIndex]; Packet-
            >RenderThreadFinalize();
            Packet->~FGatherShadowPrimitivesPacket();
        }
    }
}
```

5.6.3.9 FGatherShadowPrimitivesPacket

In the above code, calls are madePackets[Index]->AnyThreadTask()to collect shadow primitive data packets, calls are made toFilterPrimitiveForShadowsfilter primitives that are not affected by shadows, and callsPacket->RenderThreadFinalize()are made to add the collected primitives to the corresponding shadow list. The following is an analysis of their specific execution logic:

```
void FGatherShadowPrimitivesPacket::AnyThreadTask() {

    if(Node)//If there is a node {

        //  Use an octree to filter and gather primitives affected by shadows.
        for   (FScenePrimitiveOctree::ElementConstIt NodePrimitiveIt(Node->GetElementIt());
NodePrimitiveIt;          + +NodePrimitiveIt)
        {
            if(NodePrimitiveIt->PrimitiveFlagsCompact.bCastDynamicShadow) {

                FilterPrimitiveForShadows(NodePrimitiveIt->Bounds, NodePrimitiveIt-
> PrimitiveFlagsCompact, NodePrimitiveIt->PrimitiveSceneInfo, NodePrimitiveIt->Proxy);
            }
        }
    }
    else
    {
        //Use the index range traversal of the information package to filter and collect the primitives affected by the shadow one by one.
        for(int32 PrimitiveIndex = StartPrimitiveIndex; PrimitiveIndex < StartPrimitiveIndex +
NumPrimitives; PrimitiveIndex++)
        {
            FPrimitiveFlagsCompact PrimitiveFlagsCompact = Scene-
> PrimitiveFlagsCompact[PrimitiveIndex];

            if(PrimitiveFlagsCompact.bCastDynamicShadow) {

                FilterPrimitiveForShadows(Scene-
> PrimitiveBounds[PrimitiveIndex].BoxSphereBounds, PrimitiveFlagsCompact,                Scene-
> Primitives[PrimitiveIndex], Scene->PrimitiveSceneProxies[PrimitiveIndex]);
            }
        }
    }
```

```
}

void FGatherShadowPrimitivesPacket::FilterPrimitiveForShadows(constFBoxSphereBounds& PrimitiveBounds,
FPrimitiveFlagsCompact PrimitiveFlagsCompact, FPrimitiveSceneInfo* PrimitiveSceneInfo, FPrimitiveSceneProxy*
PrimitiveProxy)
{
	//Checks whether the primitive is affected by anypreshadowInfluence.
	if(PreShadows.Num() && PrimitiveFlagsCompact.bCastStaticShadow &&
	PrimitiveFlagsCompact.bStaticLighting)
	{
		for(int32 ShadowIndex =0, Num = PreShadows.Num(); ShadowIndex < Num; ShadowIndex++)

		{
			FProjectedShadowInfo* RESTRICT ProjectedShadowInfo = PreShadows[ShadowIndex]; //Intersection test between primitive
			bounding box and shadow casting cone.
			boolbInFrustum = ProjectedShadowInfo-
> CasterFrustum.IntersectBox(PrimitiveBounds.Origin, ProjectedShadowInfo-
> PreShadowTranslation, PrimitiveBounds.BoxExtent);
			//Additions within the projection cone that influence each other
			PreShadowSubjectPrimitivesmiddle. if(bInFrustum && ProjectedShadowInfo-
> GetLightSceneInfoCompact().AffectsPrimitive(PrimitiveBounds, PrimitiveProxy))
			{
				PreShadowSubjectPrimitives[ShadowIndex].Add(PrimitiveSceneInfo);
			}
		}
	}

	//Primitive and full-view shadow intersection test.
	for(int32 ShadowIndex =0, Num = ViewDependentWholeSceneShadows.Num();ShadowIndex <
Num;ShadowIndex++)
	{
		constFProjectedShadowInfo* RESTRICT ProjectedShadowInfo =
		ViewDependentWholeSceneShadows[ShadowIndex];
		constFLightSceneInfo& RESTRICT LightSceneInfo = ProjectedShadowInfo-
> GetLightSceneInfo();
		constFLightSceneProxy& RESTRICT LightProxy = *LightSceneInfo.Proxy;

		constFVector LightDirection = LightProxy.GetDirection();
		constFVector PrimitiveToShadowCenter = ProjectedShadowInfo->ShadowBounds.Center -
		PrimitiveBounds.Origin;
		//Project the primitive's bounding box onto the light vector.
		const floatProjectedDistanceFromShadowOriginAlongLightDir =

PrimitiveToShadowCenter | LightDirection;
		const floatPrimitiveDistanceFromCylinderAxisSq = (-LightDirection *
ProjectedDistanceFromShadowOriginAlongLightDir + PrimitiveToShadowCenter).SizeSquared();
		const floatCombinedRadiusSq = FMath::Square(ProjectedShadowInfo->ShadowBounds.W +
		PrimitiveBounds.SphereRadius);

		//Checks whether the primitive is inside the shadow's [cylinder].
		if(PrimitiveDistanceFromCylinderAxisSq < CombinedRadiusSq
			&& !(ProjectedDistanceFromShadowOriginAlongLightDir <0&&
PrimitiveToShadowCenter.SizeSquared() > CombinedRadiusSq)
			&& ProjectedShadowInfo-
> CascadeSettings.ShadowBoundsAccurate.IntersectBox(PrimitiveBounds.Origin,
PrimitiveBounds.BoxExtent))
		{
			//forRSMPerform distance clipping.
			const floatMinScreenRadiusForShadowCaster = ProjectedShadowInfo-
```

```
> bReflectiveShadowmap ? GMinScreenRadiusForShadowCasterRSM :
GMinScreenRadiusForShadowCaster;

                    //Screen space size cropping
                    boolbScreenSpaceSizeCulled =false; {

                        const floatDistanceSquared = (PrimitiveBounds.Origin -
ProjectedShadowInfo->DependentView->ShadowViewMatrices.GetViewOrigin()).SizeSquared();
                        bScreenSpaceSizeCulled = FMath::Square(PrimitiveBounds.SphereRadius) <
FMath::Square(MinScreenRadiusForShadowCaster) * DistanceSquared * ProjectedShadowInfo-
> DependentView->LODDistanceFactorSquared;
                    }

                    //Whether to calculate inset shadows (InsetShadow).
                    boolbCastsInsetShadows = PrimitiveProxy->CastsInsetShadow(); //
                        Additional root component that handles light sources for primitives.
                    if    (PrimitiveSceneInfo->LightingAttachmentRoot.IsValid())
                    {
                        FAttachmentGroupSceneInfo& AttachmentGroup = PrimitiveSceneInfo->Scene-
> AttachmentGroups.FindChecked(PrimitiveSceneInfo->LightingAttachmentRoot);
                        bCastsInsetShadows = AttachmentGroup.ParentSceneInfo &&
AttachmentGroup.ParentSceneInfo->Proxy->CastsInsetShadow();
                    }

                    //Test various conditions and join only after all conditions are passed
ViewDependentWholeSceneShadowSubjectPrimitivesList.
                    if(!bScreenSpaceSizeCulled
                        && ProjectedShadowInfo-
> GetLightSceneInfoCompact().AffectsPrimitive(PrimitiveBounds,                PrimitiveProxy
                        && (!LightProxy.HasStaticLighting() ||
(!LightSceneInfo.IsPrecomputedLightingValid() || LightProxy.UseCSMForDynamicObjects()))
                        && !(ProjectedShadowInfo->bReflectiveShadowmap && !PrimitiveProxy-
> AffectsDynamicIndirectLighting())
                        && (!bCastsInsetShadows || ProjectedShadowInfo->bReflectiveShadowmap) && !
                        ShouldCreateObjectShadowForStationaryLight(&LightSceneInfo, true)
PrimitiveProxy,
                        && (!bStaticSceneOnly || PrimitiveProxy->HasStaticLighting()) && (!
                        LightProxy.UseCSMForDynamicObjects() || !PrimitiveProxy-
> HasStaticLighting())
                    {

ViewDependentWholeSceneShadowSubjectPrimitives[ShadowIndex].Add(PrimitiveSceneInfo);
                    }
                }
            }
        }
}

void FGatherShadowPrimitivesPacket::RenderThreadFinalize() {

        //EachPreShadowAll primitives within the instance are added to the shadow instance.
        for(int32 ShadowIndex =0; ShadowIndex < PreShadowSubjectPrimitives.Num(); ShadowIndex++)

        {
            FProjectedShadowInfo* ProjectedShadowInfo = PreShadows[ShadowIndex];

            for(int32 PrimitiveIndex =0; PrimitiveIndex <
PreShadowSubjectPrimitives[ShadowIndex].Num(); PrimitiveIndex++)
            {
```

```
            ProjectedShadowInfo -
> AddSubjectPrimitive(PreShadowSubjectPrimitives[ShadowIndex][PrimitiveIndex], FeatureLevel,false);     &Views,

        }
    }

    //Add all primitives in each full-view shadow instance to the
    shadow instance. for(int32 ShadowIndex =0; ShadowIndex <
    ViewDependentWholeSceneShadowSubjectPrimitives.Num();         ShadowIndex++)
    {
        FProjectedShadowInfo* ProjectedShadowInfo =
        ViewDependentWholeSceneShadows[ShadowIndex];

        (......)

        for(int32 PrimitiveIndex =0; PrimitiveIndex <
        ViewDependentWholeSceneShadowSubjectPrimitives[ShadowIndex].Num();      PrimitiveIndex++)
        {
            ProjectedShadowInfo -
> AddSubjectPrimitive(ViewDependentWholeSceneShadowSubjectPrimitives[ShadowIndex] [PrimitiveIndex],
NULL, FeatureLevel, bRecordShadowSubjectsForMobile);
        }
    }
}
```

5.6.3.10 AllocateShadowDepthTargets

Continue to analyze the initialization phaseAllocateShadowDepthTargets:

```
void FSceneRenderer::AllocateShadowDepthTargets(FRHICommandListImmediate& RHICmdList) {

    FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

    //    Sort visible shadows based on allocation need.
    //    This frame2dShadow maps can be combined into atlases across lights.
    TArray<FProjectedShadowInfo*,            SceneRenderingAllocator>            Shadows;
    //Across multiple frames2dShadow maps cannot be merged into atlases.
    TArray<FProjectedShadowInfo*, TArray<FPrSocjencteeRdeSnhdadeoriwngIAnlfloc*a,toSrc>endaeCtaecdAhteldSpotlightShadows;
    multiple frames2dShadow map TArray<FProSjceecntedRSehnaddeorwinIngfAol*l,ocatSor>   TranslucentShadows;
    TArray<FProjectedShadowInfo*, //Point light sourcecubemap,Cannot be
    merged into an atlas. TArray<FProjectedShaSdcoewneInRfeon*,d eringAllocator>   CachedPreShadows;
    SceneRenderingAllocator>                SceneRenderingAllocator>           RSM Shadows;

                                                                              WholeScenePointShadows;

    //    Iterate over all light sources
    for    (TSparseArray<FLightSceneInfoCompact>::TConstIterator                LightIt(Scene->Lights);
LightIt;      + + LightIt)
    {
        constFLightSceneInfoCompact& LightSceneInfoCompact = *LightIt; FLightSceneInfo* LightSceneInfo =
        LightSceneInfoCompact.LightSceneInfo; FVisibleLightInfo& VisibleLightInfo =
        VisibleLightInfos[LightSceneInfo->Id];

        //All cascaded shadows from the same light source must be in the same texture.
        TArray<FProjectedShadowInfo*, SceneRenderingAllocator>
        WholeSceneDirectionalShadows;
```

```
//Iterate over all instances of a light source that casts shadows.
for(int32 ShadowIndex =0; ShadowIndex <
VisibleLightInfo.AllProjectedShadows.Num(); ShadowIndex++)
{
    FProjectedShadowInfo* ProjectedShadowInfo =
VisibleLightInfo.AllProjectedShadows[ShadowIndex];

    //Checks whether the shadow is in at least one
    viewVisible in. boolbShadowIsVisible =false;
    for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        FViewInfo& View = Views[ViewIndex];

        if(ProjectedShadowInfo->DependentView && ProjectedShadowInfo-!=
> DependentView      &View)
        {
            continue;
        }

        constFVisibleLightViewInfo& VisibleLightViewInfo =
View.VisibleLightInfos[LightSceneInfo->Id];
        constFPrimitiveViewRelevance ViewRelevance =
VisibleLightViewInfo.ProjectedShadowViewRelevanceMap[ShadowIndex];
        const boolbHasViewRelevance = (ProjectedShadowInfo->bTranslucentShadow &&
ViewRelevance.HasTranslucency())
            || (!ProjectedShadowInfo->bTranslucentShadow &&
ViewRelevance.bOpaque);

        bShadowIsVisible |= bHasViewRelevance &&
VisibleLightViewInfo.ProjectedShadowVisibilityMap[ShadowIndex];
    }

    //Checks conditional visibility when shadow cache mode is movable primitives.
    if(ProjectedShadowInfo->CacheMode == SDCM_MovablePrimitivesOnly &&
!ProjectedShadowInfo->HasSubjectPrims())
    {
        FCachedShadowMapData& CachedShadowMapData = Scene-
> CachedShadowMaps.FindChecked(ProjectedShadowInfo->GetLightSceneInfo().Id);
        if(!CachedShadowMapData.bCachedShadowMapHasPrimitives) {

            bShadowIsVisible =false;
        }
    }

    //The mobile renderer only supports per-shadow or
    CSM. if(FeatureLevel < ERHIFeatureLevel::SM5
        && (!ProjectedShadowInfo->bPerObjectOpaqueShadow && !(ProjectedShadowInfo-
> bDirectionalLight && ProjectedShadowInfo->bWholeSceneShadow)))
    {
        bShadowIsVisible =false;
    }

    //In the forward rendering path, dynamic shadows are cast onto the light attenuation texture (light attenuation texture)in the
    channel. if(IsForwardShadingEnabled(ShaderPlatform)
        && ProjectedShadowInfo->GetLightSceneInfo().GetDynamicShadowMapChannel()
==  - 1)
    {
        bShadowIsVisible =false;
```

```
                }

                //  If the shadow is visible, it will be added to different shadow instance lists according to
                if  different types. (bShadowIsVisible)
                {
                        (.....)

                        boolbNeedsProjection = ProjectedShadowInfo->CacheMode !=
SDCM_StaticPrimitivesOnly
                                // Mobile rendering only projects opaque per object shadows. && (FeatureLevel >=
                                ERHIFeatureLevel::SM5 || ProjectedShadowInfo-
> bPerObjectOpaqueShadow);

                        if(bNeedsProjection)
                        {
                                if(ProjectedShadowInfo->bReflectiveShadowmap) {

                                        VisibleLightInfo.RSMsToProject.Add(ProjectedShadowInfo);
                                }
                                else if(ProjectedShadowInfo->bCapsuleShadow) {

                                        VisibleLightInfo.CapsuleShadowsToProject.Add(ProjectedShadowInfo);
                                }
                                else
                                {
                                        VisibleLightInfo.ShadowsToProject.Add(ProjectedShadowInfo);
                                }
                        }

                        const boolbNeedsShadowmapSetup = !ProjectedShadowInfo->bCapsuleShadow &&
!ProjectedShadowInfo->bRayTracedDistanceField;

                        if(bNeedsShadowmapSetup) {

                                if(ProjectedShadowInfo->bReflectiveShadowmap) {

                                        check(ProjectedShadowInfo->bWholeSceneShadow);
                                        RSMShadows.Add(ProjectedShadowInfo);
                                }
                                else if(ProjectedShadowInfo->bPreShadow && ProjectedShadowInfo-
> bAllocatedInPreshadowCache)
                                {
                                        CachedPreShadows.Add(ProjectedShadowInfo);
                                }
                                else if(ProjectedShadowInfo->bDirectionalLight &&
ProjectedShadowInfo->bWholeSceneShadow)
                                {
                                        WholeSceneDirectionalShadows.Add(ProjectedShadowInfo);
                                }
                                else if(ProjectedShadowInfo->bOnePassPointLightShadow) {

                                        WholeScenePointShadows.Add(ProjectedShadowInfo);
                                }
                                else if(ProjectedShadowInfo->bTranslucentShadow) {

                                        TranslucentShadows.Add(ProjectedShadowInfo);
                                }
                                else if(ProjectedShadowInfo->CacheMode == SDCM_StaticPrimitivesOnly)
```

```
                        {
                            check(ProjectedShadowInfo->bWholeSceneShadow);
                            CachedSpotlightShadows.Add(ProjectedShadowInfo);
                        }
                        else
                        {
                            Shadows.Add(ProjectedShadowInfo);
                        }
                    }
                }
            }

            //To sort cascaded shadows, blending between cascades is required.

VisibleLightInfo.ShadowsToProject.Sort(FCompareFProjectedShadowInfoBySplitIndex());
            VisibleLightInfo.RSMsToProject.Sort(FCompareFProjectedShadowInfoBySplitIndex());

            //distributeCSMDepth render texture.
            AllocateCSMDepthTargets(RHICmdList, WholeSceneDirectionalShadows);
        }

        //Handling cachePreShadow.
        if(CachedPreShadows.Num() >0) {

            //  Creating a scenePreShadowCache depth
            if  texture. (!Scene->PreShadowCacheDepthZ)
            {
                FPooledRenderTargetDesc
Desc(FPooledRenderTargetDesc::Create2DDesc(SceneContext.GetPreShadowCacheTextureResolution (),
PF_ShadowDepth, FClearValueBinding::None, TexCreate_None,
TexCreate_DepthStencilTargetable | TexCreate_ShaderResource,false));
                Desc.AutoWritable =false; GRenderTargetPool.FindFreeElement(RHICmdList,
                Desc, Scene-
> PreShadowCacheDepthZ, TEXT("PreShadowCacheDepthZ"),true,
ERenderTargetTransience::NonTransient);
            }

            SortedShadowsForShadowDepthPass.PreshadowCache.RenderTargets.DepthTarget = Scene-
>PreShadowCacheDepthZ;

            for(int32 ShadowIndex =0; ShadowIndex < CachedPreShadows.Num(); ShadowIndex++) {

                FProjectedShadowInfo* ProjectedShadowInfo = CachedPreShadows[ShadowIndex];
                ProjectedShadowInfo->RenderTargets.DepthTarget = Scene-
> PreShadowCacheDepthZ.GetReference();
                //Setting the shadow depth view will find a dedicated view for shadows in the scene
                renderer. ProjectedShadowInfo->SetupShadowDepthView(RHICmdList, this);

SortedShadowsForShadowDepthPass.PreshadowCache.Shadows.Add(ProjectedShadowInfo);
            }
        }

        //Assign render textures for various types of shadows, including point lightscubemap,RSM,Cached spotlights, per-object, transparent shadows, etc.CSMAlready allocated previously
Passed)
        AllocateOnePassPointLightDepthTargets(RHICmdList, WholeScenePointShadows);
        AllocateRSMDepthTargets(RHICmdList, RSMShadows);
        AllocateCachedSpotlightShadowDepthTargets(RHICmdList, CachedSpotlightShadows);
        AllocatePerObjectShadowDepthTargets(RHICmdList, Shadows);
```

```
    AllocateTranslucentShadowDepthTargets(RHICmdList, TranslucentShadows);

    //Update transparent shadow mapuniform buffer.
    for(int32 TranslucentShadowIndex =0; TranslucentShadowIndex <
TranslucentShadows.Num(); ++TranslucentShadowIndex)
    {
        FProjectedShadowInfo* ShadowInfo = TranslucentShadows[TranslucentShadowIndex]; constint32
        PrimitiveIndex = ShadowInfo->GetParentSceneInfo()->GetIndex();

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

            FViewInfo& View = Views[ViewIndex];
            FUniformBufferRHIRef* UniformBufferPtr =
View.TranslucentSelfShadowUniformBufferMap.Find(PrimitiveIndex);
            if(UniformBufferPtr)
            {
                FTranslucentSelfShadowUniformParameters Parameters;
                SetupTranslucentSelfShadowUniformParameters(ShadowInfo,          Parameters);
                RHIUpdateUniformBuffer(*UniformBufferPtr, &Parameters);
            }
        }
    }

    //Delete shadow caches that are not used at all.
    for(TMap<int32, FCachedShadowMapData>::TIterator CachedShadowMapIt(Scene-
> CachedShadowMaps); CachedShadowMapIt; ++CachedShadowMapIt) {

        FCachedShadowMapData& ShadowMapData = CachedShadowMapIt.Value(); if
        (ShadowMapData.ShadowMap.IsValid() && ViewFamily.CurrentRealTime -
ShadowMapData.LastUsedTime >2.0f)
        {
            ShadowMapData.ShadowMap.Release();
        }
    }
}
```

5.6.3.11 GatherShadowDynamicMeshElements

The last step of the shadow initialization phase isGatherShadowDynamicMeshElements:

```
void FSceneRenderer::GatherShadowDynamicMeshElements(FGlobalDynamicIndexBuffer&
DynamicIndexBuffer, FGlobalDynamicVertexBuffer& DynamicVertexBuffer, FGlobalDynamicReadBuffer&
DynamicReadBuffer)
{
    TArray<constFSceneView*> ReusedViewsArray;
    ReusedViewsArray.AddZeroed(1);

    //Iterate over all shadow map atlases.
    for(int32 AtlasIndex =0; AtlasIndex <
    SortedShadowsForShadowDepthPass.ShadowMapAtlases.Num();      AtlasIndex++)
    {
        FSortedShadowMapAtlas& Atlas =
        SortedShadowsForShadowDepthPass.ShadowMapAtlases[AtlasIndex];

        //Iterate over all shadow instances in the shadow map atlas and collect the mesh elements they need to cast shadows on.
        for(int32 ShadowIndex =0; ShadowIndex < Atlas.Shadows.Num(); ShadowIndex++) {
```

```
                FProjectedShadowInfo* ProjectedShadowInfo = Atlas.Shadows[ShadowIndex]; FVisibleLightInfo&
                VisibleLightInfo = VisibleLightInfos[ProjectedShadowInfo-
> GetLightSceneInfo().Id];
                ProjectedShadowInfo->GatherDynamicMeshElements(*this, VisibleLightInfo,
ReusedViewsArray, DynamicIndexBuffer, DynamicVertexBuffer, DynamicReadBuffer);
            }
        }

        //Traverse allRSMShadow map atlases, which collect mesh elements of all shadow
        instances within each atlas. for(int32 AtlasIndex =0; AtlasIndex <
SortedShadowsForShadowDepthPass.RSMAtlases.Num(); AtlasIndex++)
        {
            FSortedShadowMapAtlas& Atlas =
            SortedShadowsForShadowDepthPass.RSMAtlases[AtlasIndex];

            for(int32 ShadowIndex =0; ShadowIndex < Atlas.Shadows.Num(); ShadowIndex++) {

                FProjectedShadowInfo* ProjectedShadowInfo = Atlas.Shadows[ShadowIndex]; FVisibleLightInfo&
                VisibleLightInfo = VisibleLightInfos[ProjectedShadowInfo-
> GetLightSceneInfo().Id];
                ProjectedShadowInfo->GatherDynamicMeshElements(*this, VisibleLightInfo,
ReusedViewsArray, DynamicIndexBuffer, DynamicVertexBuffer, DynamicReadBuffer);
            }
        }

        //Iterate over all point light cube shadow maps, collecting the mesh elements of all shadow instances
        within each cube shadow map. for(int32 AtlasIndex =0; AtlasIndex <
SortedShadowsForShadowDepthPass.ShadowMapCubemaps.Num(); AtlasIndex++)
        {
            FSortedShadowMapAtlas& Atlas =
            SortedShadowsForShadowDepthPass.ShadowMapCubemaps[AtlasIndex];

            for(int32 ShadowIndex =0; ShadowIndex < Atlas.Shadows.Num(); ShadowIndex++) {

                FProjectedShadowInfo* ProjectedShadowInfo = Atlas.Shadows[ShadowIndex]; FVisibleLightInfo&
                VisibleLightInfo = VisibleLightInfos[ProjectedShadowInfo-
> GetLightSceneInfo().Id];
                ProjectedShadowInfo->GatherDynamicMeshElements(*this, VisibleLightInfo,
ReusedViewsArray, DynamicIndexBuffer, DynamicVertexBuffer, DynamicReadBuffer);
            }
        }

        //Traverse allPreShadowCached shadow map, collecting mesh elements of
        shadow instances. for(int32 ShadowIndex =0; ShadowIndex <
        SortedShadowsForShadowDepthPass.PreshadowCache.Shadows.Num();          ShadowIndex++)
        {
            FProjectedShadowInfo* ProjectedShadowInfo =
            SortedShadowsForShadowDepthPass.PreshadowCache.Shadows[ShadowIndex];
            FVisibleLightInfo& VisibleLightInfo = VisibleLightInfos[ProjectedShadowInfo-
> GetLightSceneInfo().Id]; ProjectedShadowInfo->GatherDynamicMeshElements(*this, VisibleLightInfo,
            ReusedViewsArray, DynamicIndexBuffer, DynamicVertexBuffer, DynamicReadBuffer);

        }

        //Traverse all transparent object shadow map atlases and collect the mesh elements of all
        shadow instances in each atlas. for(int32 AtlasIndex =0; AtlasIndex <
        SortedShadowsForShadowDepthPass.TranslucencyShadowMapAtlases.Num();                AtlasIndex++)
        {
```

```
            FSortedShadowMapAtlas& Atlas =
            SortedShadowsForShadowDepthPass.TranslucencyShadowMapAtlases[AtlasIndex];

            for(int32 ShadowIndex =0; ShadowIndex < Atlas.Shadows.Num(); ShadowIndex++) {

                FProjectedShadowInfo* ProjectedShadowInfo = Atlas.Shadows[ShadowIndex]; FVisibleLightInfo&
                VisibleLightInfo = VisibleLightInfos[ProjectedShadowInfo-
> GetLightSceneInfo().Id];
                ProjectedShadowInfo->GatherDynamicMeshElements(*this, VisibleLightInfo,
ReusedViewsArray, DynamicIndexBuffer, DynamicVertexBuffer, DynamicReadBuffer);
            }
        }
}
```

5.6.3.12 Summary of Shadow Initialization

It takes a lot of words and sections to analyze the specific steps and details of shadow initialization in detail, which will inevitably make many children's shoes daunting. So this section briefly summarizes

InitDynamicShadowsthe main process of shadow initialization as follows:

- Initialize shadow related tags according to view, scene light source, and console variables.
- Loop through all light sources in the scene (Scene->Lights) and do the following:
    - If the light source does not have shadows enabled or the shadow quality is too low, or the light source is not visible in any view, it is ignored and no shadow casting is performed.
    - If it is a point light panoramic shadow, add the light source component name to the Scene's UsedWholeScenePointLightNames list.
    - If the conditions for creating a panoramic shadow are met, call CreateWholeSceneProjectedShadow:
        - Initialize shadow data and calculate the resolution and transition factor (FadeAlpha) required for the shadow.
        - If the transition factor is too small (less than 1/256), it will be returned directly. Iterate
        - over the number of cast shadows of the light source, performing resolution calculations, position (SizeX, SizeY) calculations, number of shadow maps to be created, etc. each time.
            - Create the same number of FProjectedShadowInfo shadow instances as the number of shadow maps. For each shadow instance:
                - Set panorama projection parameters (frustum bounds, transformation matrix, depth, depth offset, etc.), transition factor, cache mode, etc. Add to
                - the VisibleLightInfo.MemStackProjectedShadows list; if it is a single-channel point light shadow, fill in the data of the 6 cubemap surfaces of the shadow instance (view projection matrix, bounding box, near and far clipping planes, etc.), and initialize the cone.

            - For non-ray-traced distance field shadows, perform CPU-side clipping. Construct a convex volume of the light source view, then traverse the light source's moving primitive list, call IntersectsConvexHulls to intersect the light source bounding box with the primitive bounding box, and only those primitives that intersect will be added to the main primitive list of the shadow instance. Perform similar operations for the light source's static primitives.
            - Finally, add the shadow instance that needs to be rendered to the VisibleLightInfo.AllProjectedShadows list.

    - Create CSM (cascade shadows) for two types of light sources (mobile and fixed light sources, static light sources that have not yet been built). Call AddViewDependentWholeSceneShadowsForView to create a view-associated CSM:

        - Traverse all views, for each view:
            - If it is not the main view, skip the subsequent steps.
            - Get the number of panoramic projections related to the view, create an equal number of FProjectedShadowInfo shadow instances, set the panoramic projection parameters for each shadow instance, and finally add them to the shadow instance list and the list to be clipped.

    - Process interactive shadows (referring to the influence between light sources and primitives, including PerObject shadows, transparent shadows, self-shadows, etc.), traverse the dynamic and static primitives of the light source, and call SetupInteractionShadows to set interactive shadows for each primitive:

        - Process the light source's additional root component and set related flags. If there is an additional root component, skip the subsequent steps.
        - If you need to create a shadow instance, call CreatePerObjectProjectedShadow to create a per-object shadow:
            - Traverse all views and collect shadow related tags.
            - If it is not visible in this frame and is not potentially visible in the next frame, return directly and skip the subsequent shadow creation and setting.
            - If there is no valid shadow group primitive, return directly.
            - Calculates various shadow data (shadow frustum, resolution, visibility flags, atlas position, etc.).
            - If the transition factor (FadeAlpha) is less than a certain threshold (1.0/256.0), return directly.
            - If the shadow creation conditions are met and it is an opaque shadow, a shadow instance is created and set, and added to the VisibleLightInfo.MemStackProjectedShadows list. If it is visible in this frame, it is added to the main primitive list of the shadow instance. If it is potentially

visible in the next frame, it is added to the instance of VisibleLightInfo.OccludedPerObjectShadows.
        - If the shadow creation conditions are met and it is a semi-transparent shadow, perform similar operations as above.

- Call InitProjectedShadowVisibility to perform shadow visibility determination:
    - Iterate over all light sources in the scene, and for each light source:
        - Assigns cast shadow visibility within a view and the associated container.
        - Traverse all shadow instances of visible light sources, and for each shadow instance:
            - Save the shadow index. Traverse
            - all views, for each view:
                - Process view-dependent shadows, skipping lights that are not within the view frustum.
                - Calculate the view-related data of the main element, check whether the shadow is blocked, and set the shadow visibility mark.
                - If a shadow is visible and is not an RSM, draw the shadow cone using FViewElementPDI.

- Call UpdatePreshadowCache to clean up old precomputed shadows and try to add new ones to the cache:
    - Initialize texture layout.
    - Iterate over all cached pre-shadows, removing instances that are not rendered this frame.
    - Collects a list of PreShadows that can be cached.
    - Sort PreShadow from largest to smallest (larger PreShadows will have more objects when rendering depth).
    - Iterate over all uncached PreShadows and try to find space for the PreShadow from the texture layout. If found, set the relevant data and add it to the Scene-> CachedPreshadows list.

- Call GatherShadowPrimitives to collect a list of primitives to handle different types of shadows:
    - If there is no PreShadow and no view-associated full-scene shadows (ViewDependentWholeSceneShadows), return directly.
    - If octree traversal is allowed (determined by GUseOctreeForShadowCulling), use the octree traversal Scene->PrimitiveOctree for each child node:
        - Checks if the child node is inside at least one shadow (including PreShadow and view-dependent full-view shadows), and if so, pushes it into the node container.
        - If the element of the primitive node is greater than 0, create an FGatherShadowPrimitivesPacket instance from FMemStack, store the relevant data of the node in it, and add it to the FGatherShadowPrimitivesPacket instance list.

    - If it is non-octree traversal mode, traverse the primitives linearly, create FGatherShadowPrimitivesPacket and add it to the list.
    - Use ParallelFor to filter out the primitives that do not intersect with the shadow in parallel and collect the primitives that intersect with the shadow.
    - In the final stage of collection, the primitives affected by the shadow are added to the SubjectPrimitive list of the shadow instance, and the previously applied resources are cleaned up.

- Call AllocateShadowDepthTargets to allocate the render texture required for the shadow map:
    - Initializes a shadow list of a specified allocator of a different type.
    - Traverse all light sources, for each light source:
        - Iterate over all shadow instances of the light source, and for each shadow instance:
            - Checks whether the shadow is visible in at least one view.
            - Checks conditional visibility when shadow cache mode is movable primitives.
            - Other special visibility judgments.
            - If the shadow is visible, it is added to different shadow instance lists according to the different types.
        - Sort cascaded shadows, since blending between cascades is required to be ordered.
        - Call AllocateCSMDepthTargets to allocate the CSM depth render texture.
        - Process PreShadow.
        - Assign point light cubemap, RSM, cached spotlight, per-object, transparent shadow render textures in turn.
        - Updates the uniform buffer for the transparent shadow map.
        - Delete shadow caches that are not being used at all.

- Call GatherShadowDynamicMeshElements to collect the dynamic mesh elements of the shadow:
    - Iterate over all ShadowMapAtlases and collect the mesh elements of all shadow instances in each atlas.
    - Iterate over all RSM shadow map atlases (RSMAtlases) and collect the mesh elements of all shadow instances within each atlas.
    - Iterate over all point light cube shadow maps (ShadowMapCubemaps) and collect the mesh elements of all shadow instances within each cube shadow map.
    - Traverse all PreShadow cached shadow maps (PreshadowCache) and collect mesh elements of shadow instances.
    - Iterate over all TranslucencyShadowMapAtlases and collect the mesh elements of all shadow instances in each atlas.

The shadow initialization is summarized. It can be seen that the shadow processing is very complicated, involving a lot of logic and optimization techniques. Here is a rough summary of**the**

optimization techniquesinvolved in the shadow initialization stage:

- Use the simple shapes of objects (views, light sources, shadows, primitives) to perform intersection tests and remove non-intersecting shadow elements.
- Use various tags (objects, views, consoles, global variables, etc.) and derived tags to remove shadow elements that do not conform.
- Use intermediate data (transition factor, screen size, depth value, etc.) to remove shadow elements that do not conform.
- Special data structures (texture layout, shadow atlas, octree, contiguous linear array) and traversal methods (parallel, octree, linear) improve performance.
- Make full use of caches (PreShadowCache, Latent Visibility, etc.) to reduce rendering effects. Sort
- shadow types to reduce rendering state switching, reduce CPU and GPU interaction data, and improve cache hit rate.
- Occlusion culling at different granularities.

5.6.4 Shadow Rendering

After analyzing the initialization phase of the shadow, let's analyze the rendering phase of the shadow. The shadow rendering Pass is after PrePass and before BasePass:

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

    (......)

    RenderPrePass(RHICmdList, ...);

    (......)

    //Shadow RenderingPass.
    RenderShadowDepthMaps(RHICmdList);

    (......)

    RenderBasePass(RHICmdList, ...);

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

5.6.4.1 RenderShadowDepthMaps

Let's go directly to`RenderShadowDepthMaps` the analysis source code:

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

void FSceneRenderer::RenderShadowDepthMaps(FRHICommandListImmediate& RHICmdList) {

    (......)

    //Renders a shadow atlas.
    FSceneRenderer::RenderShadowDepthMapAtlases(RHICmdList);
```

```
//Renders a point light shadow cubemap.
for(int32 CubemapIndex =0; CubemapIndex <
SortedShadowsForShadowDepthPass.ShadowMapCubemaps.Num();    CubemapIndex++)
{
    constFSortedShadowMapAtlas& ShadowMap =
    SortedShadowsForShadowDepthPass.ShadowMapCubemaps[CubemapIndex];
    FSceneRenderTargetItem& RenderTarget = ShadowMap.RenderTargets.DepthTarget-
> GetRenderTargetItem();
    FIntPoint TargetSize = ShadowMap.RenderTargets.DepthTarget->GetDesc().Extent;

    FProjectedShadowInfo* ProjectedShadowInfo = ShadowMap.Shadows[0];

    //Is it possible to draw in parallel?
    const boolbDoParallelDispatch = RHICmdList.IsImmediate() &&// translucent shadows are draw on the render
thread, using a recursive cmdlist (which is not immediate)
        GRHICommandList.UseParallelAlgorithms() &&
CVarParallelShadows.GetValueOnRenderThread() &&
        (ProjectedShadowInfo->IsWholeSceneDirectionalShadow() ||
CVarParallelShadowsNonWholeScene.GetValueOnRenderThread());

    FString LightNameWithLevel; GetLightNameForDrawEvent(ProjectedShadowInfo-
    >GetLightSceneInfo().Proxy, LightNameWithLevel);


    //set upUniform Buffer. ProjectedShadowInfo->SetupShadowUniformBuffers(RHICmdList,
    Scene);

    //Shadow RenderingPassstart.
    autoBeginShadowRenderPass = [this, &RenderTarget, &SceneContext](FRHICommandList& InRHICmdList,bool
    bPerformClear)
    {
        FRHITexture* DepthTarget = RenderTarget.TargetableTexture;
        ERenderTargetLoadAction DepthLoadAction = bPerformClear?
ERenderTargetLoadAction::EClear : ERenderTargetLoadAction::ELoad;

        //RenderingPassinformation.
        FRHIRenderPassInfoRPInfo(DepthTarget,
MakeDepthStencilTargetActions(MakeRenderTargetActions(DepthLoadAction,
ERenderTargetStoreAction::EStore), ERenderTargetActions::Load_Store),                nullptr,
FExclusiveDepthStencil::DepthWrite_StencilWrite);

        if(!GSupportsDepthRenderTargetWithoutColorRenderTarget) {

            RPInfo.ColorRenderTargets[0].Action =
ERenderTargetActions::DontLoad_DontStore;
            RPInfo.ColorRenderTargets[0].ArraySlice        =-1;
            RPInfo.ColorRenderTargets[0].MipIndex =        0;
            RPInfo.ColorRenderTargets[0].RenderTarget        =
SceneContext.GetOptionalShadowDepthColorSurface(InRHICmdList,        DepthTarget->GetTexture2D() -
> GetSizeX(), DepthTarget->GetTexture2D()->GetSizeY());

            InRHICmdList.TransitionResource(EResourceTransitionAccess::EWritable,
RPInfo.ColorRenderTargets[0].RenderTarget);
        }
        //Converts the render texture state to writable.

        InRHICmdList.TransitionResource(EResourceTransitionAccess::EWritable,
DepthTarget);
```

```
                InRHICmdList.BeginRenderPass(RPInfo, TEXT("ShadowDepthCubeMaps"));
            };

            //Whether the shadow map needs to be cleaned up.
            {
                boolbDoClear =true;

                if(ProjectedShadowInfo->CacheMode == SDCM_MovablePrimitivesOnly
                    && Scene->CachedShadowMaps.FindChecked(ProjectedShadowInfo-
> GetLightSceneInfo().Id).bCachedShadowMapHasPrimitives)
                {
                    // Skip the clear when we'll copy from a cached shadowmap bDoClear =false;

                }

                BeginShadowRenderPass(RHICmdList, bDoClear);
            }

            if(bDoParallelDispatch) {

                // In parallel mode this first pass will just be the clear.
                RHICmdList.EndRenderPass();
            }

            //Now it's time to actually render the shadow map.
            ProjectedShadowInfo->RenderDepth(RHICmdList,          this,     BeginShadowRenderPass,
            bDoParallelDispatch);

            if(!bDoParallelDispatch) {

                RHICmdList.EndRenderPass();
            }

            //Converts render texture state to readable.
            RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
            RenderTarget.TargetableTexture);
        }

        // Preshadowcache.
        if(SortedShadowsForShadowDepthPass.PreshadowCache.Shadows.Num() >0) {

            FSceneRenderTargetItem& RenderTarget =
            SortedShadowsForShadowDepthPass.PreshadowCache.RenderTargets.DepthTarget-
> GetRenderTargetItem();

            //Traverse allPreshadowCacheAll shadow instances of . for(int32
            ShadowIndex =0; ShadowIndex <
            SortedShadowsForShadowDepthPass.PreshadowCache.Shadows.Num(); ShadowIndex++)
            {
                FProjectedShadowInfo* ProjectedShadowInfo =
 SortedShadowsForShadowDepthPass.PreshadowCache.Shadows[ShadowIndex];

                //  Only those that are not cached need to be drawn.
                if  (!ProjectedShadowInfo->bDepthsCached)
                {
                    const boolbDoParallelDispatch = RHICmdList.IsImmediate() &&
GRHICommandList.UseParallelAlgorithms() && CVarParallelShadows.GetValueOnRenderThread() &&
(ProjectedShadowInfo->IsWholeSceneDirectionalShadow() ||
```

```
CVarParallelShadowsNonWholeScene.GetValueOnRenderThread());

                        ProjectedShadowInfo->SetupShadowUniformBuffers(RHICmdList, Scene);

                        autoBeginShadowRenderPass = [this, ProjectedShadowInfo](FRHICommandList& bool
InRHICmdList,           bPerformClear)
                        {
                            FRHITexture* PreShadowCacheDepthZ = Scene->PreShadowCacheDepthZ-
> GetRenderTargetItem().TargetableTexture.GetReference();
                            InRHICmdList.TransitionResources(EResourceTransitionAccess::EWritable,
&PreShadowCacheDepthZ,1);

                            FRHIRenderPassInfoRPInfo(PreShadowCacheDepthZ,
EDepthStencilTargetActions::LoadDepthStencil_StoreDepthStencil, nullptr,
FExclusiveDepthStencil::DepthWrite_StencilWrite);

                            // Must preserve existing contents as the clear will be scissored
                            InRHICmdList.BeginRenderPass(RPInfo, TEXT("ShadowDepthMaps"));
                            ProjectedShadowInfo->ClearDepth(InRHICmdList, this,0, nullptr,
PreShadowCacheDepthZ, bPerformClear);
                        };

                        BeginShadowRenderPass(RHICmdList,true);

                        if(bDoParallelDispatch) {

                            // In parallel mode the first pass is just the clear.
                            RHICmdList.EndRenderPass();
                        }

                        //Start drawing.
                        ProjectedShadowInfo->RenderDepth(RHICmdList,        this,    BeginShadowRenderPass,
bDoParallelDispatch);

                        if(!bDoParallelDispatch) {

                            RHICmdList.EndRenderPass();
                        }

                        //Already drawn, marker cached.
                        ProjectedShadowInfo->bDepthsCached =true;
                    }
                }

                RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
                RenderTarget.TargetableTexture);
            }

            //Atlas of shadows for semi-transparent objects.
            for(int32 AtlasIndex =0; AtlasIndex <
            SortedShadowsForShadowDepthPass.TranslucencyShadowMapAtlases.Num();        AtlasIndex++)
            {
                constFSortedShadowMapAtlas& ShadowMapAtlas =
                SortedShadowsForShadowDepthPass.TranslucencyShadowMapAtlases[AtlasIndex];
                FIntPoint TargetSize = ShadowMapAtlas.RenderTargets.ColorTargets[0]-
> GetDesc().Extent;

                //Semi-transparent shadows require two render textures.
```

```
        FSceneRenderTargetItem ColorTarget0 =
ShadowMapAtlas.RenderTargets.ColorTargets[0]->GetRenderTargetItem();
        FSceneRenderTargetItem ColorTarget1 =
ShadowMapAtlas.RenderTargets.ColorTargets[1]->GetRenderTargetItem();

        FRHITexture* RenderTargetArray[2] = {

            ColorTarget0.TargetableTexture,
            ColorTarget1.TargetableTexture
        };

        FRHIRenderPassInfoRPInfo(UE_ARRAY_COUNT(RenderTargetArray),          RenderTargetArray,
        ERenderTargetActions::Load_Store);
        TransitionRenderPassTargets(RHICmdList,          RPInfo);
        RHICmdList.BeginRenderPass(RPInfo, {          TEXT("RenderTranslucencyDepths"));

            //Iterate over all shadow instances in the semi-transparent shadow atlas,     Performs semi-transparent shadow depth drawing.
            for(int32 ShadowIndex =0; ShadowIndex < ShadowMapAtlas.Shadows.Num();
ShadowIndex++)
            {
                FProjectedShadowInfo* ProjectedShadowInfo =
ShadowMapAtlas.Shadows[ShadowIndex];
                    //Renders semi-transparent shadow depth.
                    ProjectedShadowInfo->RenderTranslucencyDepths(RHICmdList, this);
            }
        }
        RHICmdList.EndRenderPass();

        RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
        ColorTarget0.TargetableTexture);
        RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
        ColorTarget1.TargetableTexture);
    }

    //set upLPVofRSM uniform Buffer,So that drawing can be submitted in parallel later. {

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

            FViewInfo& View = Views[ViewIdx]; FSceneViewState*
            ViewState = View.ViewState;

            if(ViewState)
            {
                FLightPropagationVolume* Lpv = ViewState-
> GetLightPropagationVolume(FeatureLevel);

                    if(Lpv)
                    {
                        Lpv->SetRsmUniformBuffer();
                    }
            }
        }
    }

    //RenderingRSM (Reflective Shadow Map,Reflection Shadow Maps) atlas. for
    (int32 AtlasIndex =0; AtlasIndex <
SortedShadowsForShadowDepthPass.RSMAtlases.Num(); AtlasIndex++)
    {
```

```
checkSlow(RHICmdList.IsOutsideRenderPass());

constFSortedShadowMapAtlas& ShadowMapAtlas =
SortedShadowsForShadowDepthPass.RSMAtlases[AtlasIndex];
FSceneRenderTargetItem ColorTarget0 =
ShadowMapAtlas.RenderTargets.ColorTargets[0]->GetRenderTargetItem();
FSceneRenderTargetItem ColorTarget1 =
ShadowMapAtlas.RenderTargets.ColorTargets[1]->GetRenderTargetItem();
FSceneRenderTargetItem DepthTarget = ShadowMapAtlas.RenderTargets.DepthTarget-
> GetRenderTargetItem();
FIntPoint TargetSize = ShadowMapAtlas.RenderTargets.DepthTarget->GetDesc().Extent;

SCOPED_DRAW_EVENTF(RHICmdList, EventShadowDepths, TEXT("RSM%u %ux%u"), AtlasIndex, TargetSize.X,
TargetSize.Y);

for(int32 ShadowIndex =0; ShadowIndex < ShadowMapAtlas.Shadows.Num(); ShadowIndex++)

{
    FProjectedShadowInfo* ProjectedShadowInfo =
ShadowMapAtlas.Shadows[ShadowIndex];
    SCOPED_GPU_MASK(RHICmdList, GetGPUMaskForShadow(ProjectedShadowInfo));

    const boolbDoParallelDispatch = RHICmdList.IsImmediate() &&// translucent
shadows are draw on the render thread, using a recursive cmdlist (which is not immediate)
        GRHICommandList.UseParallelAlgorithms() &&
CVarParallelShadows.GetValueOnRenderThread() &&
        (ProjectedShadowInfo->IsWholeSceneDirectionalShadow() ||
CVarParallelShadowsNonWholeScene.GetValueOnRenderThread());

    FSceneViewState* ViewState = (FSceneViewState*)ProjectedShadowInfo-
> DependentView->State;
    FLightPropagationVolume* LightPropagationVolume = ViewState-
> GetLightPropagationVolume(FeatureLevel);

    ProjectedShadowInfo->SetupShadowUniformBuffers(RHICmdList, Scene,
LightPropagationVolume);

    autoBeginShadowRenderPass = [this, LightPropagationVolume,
ProjectedShadowInfo, &ColorTarget0, &ColorTarget1, &DepthTarget](FRHICommandList& InRHICmdList,
       boolbPerformClear)
    {
        FRHITexture* RenderTargets[2];
        RenderTargets[0]        =  ColorTarget0.TargetableTexture;
        RenderTargets1]         = ColorTarget1.TargetableTexture;

        // Hook up the geometry volume UAVs
        FRHIUnorderedAccessView* Uavs[U0a]vs4];
                =LightPropagationVolume->GetGvListBufferUav();
        Uavs1]     =LightPropagationVolume->GetGvListHeadBufferUav();
        Uavs2]     =LightPropagationVolume->GetVplListBufferUav();
        Uavs3]     = LightPropagationVolume->GetVplListHeadBufferUav();

        FRHIRenderPassInfo        RPInfo(UE_ARRAY_COUNT(RenderTargets), RenderTargets,
ERenderTargetActions::Load_Store);
        RPInfo.DepthStencilRenderTarget.Action                =
EDepthStencilTargetActions::LoadDepthStencil_StoreDepthStencil;
        RPInfo.DepthStencilRenderTarget.DepthStencilTarget                    =
DepthTarget.TargetableTexture;
```

```
                RPInfo.DepthStencilRenderTarget.ExclusiveDepthStencil =
FExclusiveDepthStencil::DepthWrite_StencilWrite;


                InRHICmdList.TransitionResources(EResourceTransitionAccess::ERWBarrier,
EResourceTransitionPipeline::EGfxToGfx, Uavs, UE_ARRAY_COUNT(Uavs));
                InRHICmdList.BeginRenderPass(RPInfo, TEXT("ShadowAtlas"));

                ProjectedShadowInfo->ClearDepth(InRHICmdList, this,
UE_ARRAY_COUNT(RenderTargets), RenderTargets, DepthTarget.TargetableTexture, bPerformClear);

            };

            {
                SCOPED_DRAW_EVENT(RHICmdList, Clear);
                BeginShadowRenderPass(RHICmdList,true);
            }

            // In parallel mode the first renderpass is just the clear. if(bDoParallelDispatch) {


                RHICmdList.EndRenderPass();
            }

            ProjectedShadowInfo->RenderDepth(RHICmdList, this, BeginShadowRenderPass,
bDoParallelDispatch);

            if(!bDoParallelDispatch) {

                RHICmdList.EndRenderPass();
            }
            {
                // Resolve the shadow depth z surface.
                RHICmdList.CopyToResolveTarget(DepthTarget.TargetableTexture,
DepthTarget.ShaderResourceTexture, FResolveParams());
                RHICmdList.CopyToResolveTarget(ColorTarget0.TargetableTexture,
ColorTarget0.ShaderResourceTexture, FResolveParams());
                RHICmdList.CopyToResolveTarget(ColorTarget1.TargetableTexture,
ColorTarget1.ShaderResourceTexture, FResolveParams());

                FRHIUnorderedAccessView*    UavsToReadable[2];
                UavsToReadable[0]    =LightPropagationVolume->GetGvListBufferUav();
                UavsToReadable[1]    = LightPropagationVolume->GetGvListHeadBufferUav();
                RHICmdList.TransitionResources(EResourceTransitionAccess::EReadable,
EResourceTransitionPipeline::EGfxToGfx, UavsToReadable, UE_ARRAY_COUNT(UavsToReadable));
            }
        }
    }
}
```

To summarize the shadow rendering process, the panoramic shadow atlas, point light source shadow cube map, PreShadow, translucent object shadow atlas, and RSM are drawn in sequence. During this period, RenderDepth and RenderTranslucencyDepths of FProjectedShadowInfo are called to render opaque shadows and translucent shadows.

5.6.4.2 FProjectedShadowInfo::RenderDepth

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

void FProjectedShadowInfo::RenderDepth(FRHICommandListImmediate& RHICmdList, FSceneRenderer* SceneRenderer,
FBeginShadowRenderPassFunction BeginShadowRenderPass,bool bDoParallelDispatch)

{
	RenderDepthInner(RHICmdList, SceneRenderer, BeginShadowRenderPass,
	bDoParallelDispatch);
}

void FProjectedShadowInfo::RenderDepthInner(FRHICommandListImmediate& RHICmdList, FSceneRenderer*
SceneRenderer, FBeginShadowRenderPassFunction BeginShadowRenderPass,bool bDoParallelDispatch)

{
	constERHIFeatureLevel::Type FeatureLevel = ShadowDepthView->FeatureLevel; FRHIUniformBuffer*
	PassUniformBuffer = ShadowDepthPassUniformBuffer;

	const boolbIsWholeSceneDirectionalShadow = IsWholeSceneDirectionalShadow();

	// Parallel light panoramic shadowUniform Buffer
	if renew. (bIsWholeSceneDirectionalShadow)
	{
		//Using cacheUniform BufferUpdate the shadow view'sUniform Buffer. ShadowDepthView-
		>ViewUniformBuffer.UpdateUniformBufferImmediate(*ShadowDepthView-
>CachedViewUniformShaderParameters);

		//If there are dependent views, traverse all persistent viewsUBExtension, execute the start rendering
		command. if(DependentView)
		{
			externTSet<IPersistentViewUniformBufferExtension*>
PersistentViewUniformBufferExtensions;

			for(IPersistentViewUniformBufferExtension* Extension:
PersistentViewUniformBufferExtensions)
			{
				Extension->BeginRenderView(DependentView);
			}
		}
	}

	//Drawing for mobile platformsUniform Bufferrenew.
	if(FSceneInterface::GetShadingPath(FeatureLevel) == EShadingPath::Mobile) {

		FMobileShadowDepthPassUniformParameters		ShadowDepthPassParameters;
		SetupShadowDepthPassUniformBuffer(this,		RHICmdList, *ShadowDepthView,
		ShadowDepthPassParameters);
		SceneRenderer->Scene-
> UniformBuffers.MobileCSMShadowDepthPassUniformBuffer.UpdateUniformBufferImmediate(ShadowD
epthPassParameters);

MobileShadowDepthPassUniformBuffer.UpdateUniformBufferImmediate(ShadowDepthPassParameters);

		PassUniformBuffer = SceneRenderer->Scene-
> UniformBuffers.MobileCSMShadowDepthPassUniformBuffer; }


	//Setting up the gridPassThe rendering status.
```

```
        FMeshPassProcessorRenderStateDrawRenderState(*ShadowDepthView, PassUniformBuffer);
        SetStateForShadowDepth(bReflectiveShadowmap, bOnePassPointLightShadow, DrawRenderState);

        SetStateForView(RHICmdList);

        if(CacheMode == SDCM_MovablePrimitivesOnly)
        {
            if(bDoParallelDispatch) {

                BeginShadowRenderPass(RHICmdList,false);
            }

            //The depth of static primitives is copied before rendering the depth of movable primitives.
            CopyCachedShadowMap(RHICmdList, DrawRenderState, SceneRenderer, *ShadowDepthView);

            if(bDoParallelDispatch) {

                RHICmdList.EndRenderPass();
            }
        }

        //  Parallel drawing
        if  (bDoParallelDispatch)
        {
            boolbFlush = CVarRHICmdFlushRenderThreadTasksShadowPass.GetValueOnRenderThread()
> 0
                || CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() >0;
            FScopedCommandListWaitForTasksFlusher(bFlush);

            //Send rendering commands.
            {
                FShadowParallelCommandListSetParallelCommandListSet(*ShadowDepthView,
SceneRenderer, RHICmdList, CVarRHICmdShadowDeferredContexts.GetValueOnRenderThread() >0, !bFlush,
DrawRenderState, *this, BeginShadowRenderPass);

                ShadowDepthPass.DispatchDraw(&ParallelCommandListSet, RHICmdList);
            }
        }
        //  Non-parallel drawing
        else
        {
            ShadowDepthPass.DispatchDraw(nullptr,               RHICmdList);
        }
}
```

From this we can see that the depth rendering Pass and the scene's depth Pass are similar, with only slight differences in some state processing.

5.6.4.3 FProjectedShadowInfo::RenderTranslucencyDepths

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

void FProjectedShadowInfo::RenderTranslucencyDepths(FRHICommandList& RHICmdList, FSceneRenderer*
SceneRenderer)
{
    //Set the transparency depthPassofUniform Buffer.
```

```
        FTranslucencyDepthPassUniformParameters TranslucencyDepthPassParameters;
        SetupTranslucencyDepthPassUniformBuffer(this, RHICmdList,                    * ShadowDepthView,
TranslucencyDepthPassParameters);
        TUniformBufferRef<FTranslucencyDepthPassUniformParameters>                   PassUniformBuffer      =
TUniformBufferRef<FTranslucencyDepthPassUniformParameters>::CreateUniformBufferImmediate(T
ranslucencyDepthPassParameters, UniformBuffer_SingleFrame,
EUniformBufferValidation::None);

        //set upMeshPassProcessorThe rendering status. FMeshPassProcessorRenderState
        DrawRenderState(*ShadowDepthView, {                                                  PassUniformBuffer);

            //Cleaned up shadows and borders.
            RHICmdList.SetViewport(
                X, Y,0.0f,
                //Note that the viewport length and width include
                2*BorderSize. (X + BorderSize *2+ ResolutionX),
                (Y + BorderSize *2+ Resolution Y),1.0f);

            FLinearColor ClearColors[2] = {FLinearColor(0,0,0,0), FLinearColor(0,0,0,0)};
            DrawClearQuadMRT(RHICmdList,true, UE_ARRAY_COUNT(ClearColors), ClearColors, false,1.0f,false,0);


            // Set the viewport for the shadow.
            RHICmdList.SetViewport(
                (X + BorderSize), (Y + BorderSize),0.0f,
                //Note that the viewport length and width includeBorderSize(Not the above
                2*BorderSize). (X + BorderSize + ResolutionX),
                (Y + BorderSize + ResolutionY),1.0f);

            DrawRenderState.SetDepthStencilState(TStaticDepthStencilState<false,
            CF_Always>::GetRHI());
            //The mixed state is superposition.
            DrawRenderState.SetBlendState(TStaticBlendState<

                CW_RGBA, BO_Add, BF_One, BF_One, BO_Add, BF_One, BF_One, CW_RGBA, BO_Add, BF_One,
                BF_One, BO_Add, BF_One, BF_One>::GetRHI());

            //VisibleMeshDrawCommandList. FMeshCommandOneFrameArray
            VisibleMeshDrawCommands; FDynamicPassMeshDrawListContext

TranslucencyDepthContext(DynamicMeshDrawCommandStorage, VisibleMeshDrawCommands,
GraphicsMinimalPipelineStateSet, NeedsShaderInitialisation);

            //Declare Transparent DepthPassprocessor.
            FTranslucencyDepthPassMeshProcessor                TranslucencyDepthPassMeshProcessor(
                SceneRenderer->Scene,
                ShadowDepthView,
                DrawRenderState,
                this,
                &TranslucencyDepthContext);


            //Iterate over all dynamic transparent grid elements
            for(int32 MeshBatchIndex =0; MeshBatchIndex <
DynamicSubjectTranslucentMeshElements.Num(); MeshBatchIndex++)
            {
                constFMeshBatchAndRelevance& MeshAndRelevance =
DynamicSubjectTranslucentMeshElements[MeshBatchIndex];
                constuint64 BatchElementMask = ~0ull;
                //Will be transparentMeshBatchJoinProcessorIn order to convertMeshDrawCommand.
```

```
            TranslucencyDepthPassMeshProcessor.AddMeshBatch(*MeshAndRelevance.Mesh,
BatchElementMask, MeshAndRelevance.PrimitiveSceneProxy);
        }

        //Iterate over all static transparent mesh primitives.
        for(int32 PrimitiveIndex =0; PrimitiveIndex <
SubjectTranslucentPrimitives.Num(); PrimitiveIndex++)
        {
            constFPrimitiveSceneInfo* PrimitiveSceneInfo =
SubjectTranslucentPrimitives[PrimitiveIndex];
            int32 PrimitiveId = PrimitiveSceneInfo->GetIndex(); FPrimitiveViewRelevance
            ViewRelevance = ShadowDepthView-
> PrimitiveViewRelevanceMap[PrimitiveId];

            if(!ViewRelevance.bInitializedThisFrame) {

                // Compute the subject primitive's view relevance since it wasn't cached ViewRelevance =
                PrimitiveSceneInfo->Proxy-
> GetViewRelevance(ShadowDepthView);
            }

            if(ViewRelevance.bDrawRelevance && ViewRelevance.bStaticRelevance) {

                for(int32 MeshIndex =0; MeshIndex < PrimitiveSceneInfo-
> StaticMeshes.Num(); MeshIndex++)
                {
                    constFStaticMeshBatch& StaticMeshBatch = PrimitiveSceneInfo-
> StaticMeshes[MeshIndex];
                    constuint64 BatchElementMask =
StaticMeshBatch.bRequiresPerElementVisibility ? ShadowDepthView-
> StaticMeshBatchVisibility[StaticMeshBatch.BatchVisibilityId] : ~0ull;
                    TranslucencyDepthPassMeshProcessor.AddMeshBatch(StaticMeshBatch,
BatchElementMask, StaticMeshBatch.PrimitiveSceneInfo->Proxy, StaticMeshBatch.Id);
                }
            }
        }

        //The drawing command is actually submitted only when there is a valid
        mesh drawing command. if(VisibleMeshDrawCommands.Num() >0) {

            const boolbDynamicInstancing = IsDynamicInstancingEnabled(ShadowDepthView-
> FeatureLevel);

            FRHIVertexBuffer* PrimitiveIdVertexBuffer = nullptr; //View data to
            grid drawing instructions.
            ApplyViewOverridesToMeshDrawCommands(*ShadowDepthView,
VisibleMeshDrawCommands, DynamicMeshDrawCommandStorage, GraphicsMinimalPipelineStateSet,
NeedsShaderInitialisation);
            //Sort grid drawing instructions.
            SortAndMergeDynamicPassMeshDrawCommands(SceneRenderer->FeatureLevel,
VisibleMeshDrawCommands, DynamicMeshDrawCommandStorage, PrimitiveIdVertexBuffer,1);
            //Submit a mesh drawing command.
            SubmitMeshDrawCommands(VisibleMeshDrawCommands,
GraphicsMinimalPipelineStateSet, PrimitiveIdVertexBuffer,0, bDynamicInstancing,1, RHICmdList);

        }
```

```
    }
}
```

From this we can see that when rendering the shadow of a translucent object, unlike an opaque object, it will only convert FMesh into FMeshDrawCommand through FTranslucencyDepthPassMeshProcessor during the rendering phase. Only when there are valid instructions will the instructions be sorted and actually submitted for drawing.

5.6.4.4 RenderShadowDepthMapAtlases

RenderShadowDepthMapsThe very beginning is toRenderShadowDepthMapAtlasescall the CSM shadow atlas, the following is its code analysis:

```
void FSceneRenderer::RenderShadowDepthMapAtlases(FRHICommandListImmediate& RHICmdList) {

    FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

    //Whether parallel drawing is available.
    boolbCanUseParallelDispatch = RHICmdList.IsImmediate() &&
        GRHICommandList.UseParallelAlgorithms() &&
        CVarParallelShadows.GetValueOnRenderThread();

    //Iterate over all shadow map atlases.
    for(int32 AtlasIndex =0; AtlasIndex <
    SortedShadowsForShadowDepthPass.ShadowMapAtlases.Num();      AtlasIndex++)
    {
        constFSortedShadowMapAtlas& ShadowMapAtlas =
        SortedShadowsForShadowDepthPass.ShadowMapAtlases[AtlasIndex];
        //Rendering textures.
        FSceneRenderTargetItem& RenderTarget = ShadowMapAtlas.RenderTargets.DepthTarget-
> GetRenderTargetItem();
        FIntPoint AtlasSize = ShadowMapAtlas.RenderTargets.DepthTarget->GetDesc().Extent;

        //Start shadow renderingPass.
        autoBeginShadowRenderPass = [this, &RenderTarget, &SceneContext](FRHICommandList& InRHICmdList,bool
        bPerformClear)
        {
            ERenderTargetLoadAction DepthLoadAction = bPerformClear?
ERenderTargetLoadAction::EClear : ERenderTargetLoadAction::ELoad;

            //RenderingPassinformation.
            FRHIRenderPassInfoRPInfo(RenderTarget.TargetableTexture,
MakeDepthStencilTargetActions(MakeRenderTargetActions(DepthLoadAction,
ERenderTargetStoreAction::EStore), ERenderTargetActions::Load_Store),                nullptr,
FExclusiveDepthStencil::DepthWrite_StencilWrite);

            //If depth render textures without color are not supported, reallocate the render texture, changing
            itsActionstate. if(!GSupportsDepthRenderTargetWithoutColorRenderTarget) {

                RPInfo.ColorRenderTargets[0].Action =
ERenderTargetActions::DontLoad_DontStore;
                RPInfo.ColorRenderTargets[0].RenderTarget =
SceneContext.GetOptionalShadowDepthColorSurface(InRHICmdList,
RPInfo.DepthStencilRenderTarget.DepthStencilTarget->GetTexture2D()->GetSizeX(),
RPInfo.DepthStencilRenderTarget.DepthStencilTarget->GetTexture2D()->GetSizeY());
```

```
                InRHICmdList.TransitionResource(EResourceTransitionAccess::EWritable,
RPInfo.ColorRenderTargets[0].RenderTarget);
            }
            InRHICmdList.TransitionResource(EResourceTransitionAccess::EWritable,
RPInfo.DepthStencilRenderTarget.DepthStencilTarget);
            InRHICmdList.BeginRenderPass(RPInfo, TEXT("ShadowMapAtlases"));

            if(!bPerformClear)
            {
                InRHICmdList.BindClearMRTValues(false,true,false);
            }
        };

        TArray<FProjectedShadowInfo*,        SceneRenderingAllocator>        ParallelShadowPasses;
        TArray<FProjectedShadowInfo*,        SceneRenderingAllocator>        SerialShadowPasses;

        //Collect the renderings herePass,Toggle rendering with minimizePass.
        for(int32 ShadowIndex =0; ShadowIndex < ShadowMapAtlas.Shadows.Num(); ShadowIndex++)

        {
            FProjectedShadowInfo* ProjectedShadowInfo =
ShadowMapAtlas.Shadows[ShadowIndex];

            const boolbDoParallelDispatch = bCanUseParallelDispatch &&
                (ProjectedShadowInfo->IsWholeSceneDirectionalShadow() ||
CVarParallelShadowsNonWholeScene.GetValueOnRenderThread());

            //  Depending on whether to join different rendering
            if  PassList. (bDoParallelDispatch)
            {
                //Parallel lists.
                ParallelShadowPasses.Add(ProjectedShadowInfo);
            }
            else
            {
                //  Continuous list.
                SerialShadowPasses.Add(ProjectedShadowInfo);
            }
        }

        FLightSceneProxy* CurrentLightForDrawEvent =NULL;

        //Parallel rendering queues.
        if(ParallelShadowPasses.Num() >0) {

            {
                // Clear before going wide.
                BeginShadowRenderPass(RHICmdList,          true);
                RHICmdList.EndRenderPass();
            }

            for(int32 ShadowIndex =0; ShadowIndex < ParallelShadowPasses.Num();
ShadowIndex++)
            {
                FProjectedShadowInfo* ProjectedShadowInfo =
ParallelShadowPasses[ShadowIndex];

                (.....)
```

```
                ProjectedShadowInfo->SetupShadowUniformBuffers(RHICmdList,                Scene);
                ProjectedShadowInfo->TransitionCachedShadowmap(RHICmdList, //Call the    Scene);
                rendering depth interface of the shadow instance.
                ProjectedShadowInfo->RenderDepth(RHICmdList, this, BeginShadowRenderPass,
true);
            }
        }

        CurrentLightForDrawEvent = nullptr;

        //Non-parallel drawing mode.
        if(SerialShadowPasses.Num() >0) {

            boolbForceSingleRenderPass =
CVarShadowForceSerialSingleRenderPass.GetValueOnAnyThread()          !=   0;
            if(bForceSingleRenderPass) {

                BeginShadowRenderPass(RHICmdList,true);
            }

            for(int32 ShadowIndex =0; ShadowIndex < SerialShadowPasses.Num();
ShadowIndex++)
            {
                FProjectedShadowInfo* ProjectedShadowInfo =
SerialShadowPasses[ShadowIndex];

                (.....)

                ProjectedShadowInfo->SetupShadowUniformBuffers(RHICmdList,                Scene);
                ProjectedShadowInfo->TransitionCachedShadowmap(RHICmdList, if(!           Scene);
                bForceSingleRenderPass) {

                    BeginShadowRenderPass(RHICmdList, ShadowIndex ==0);
                }

                //Call the rendering depth interface of the shadow instance.
                ProjectedShadowInfo->RenderDepth(RHICmdList, this, BeginShadowRenderPass,
false);

                if(!bForceSingleRenderPass) {

                    RHICmdList.EndRenderPass();
                }
            }
            if(bForceSingleRenderPass) {

                RHICmdList.EndRenderPass();
            }
        }

        //Converts a render texture to be readable.
        RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
        RenderTarget.TargetableTexture);
    }
}
```

5.6.5 Shadow Application

The previous two sections have analyzed the initialization and rendering logic of shadows in detail. So how is the rendered depth map applied to lighting? This section will reveal the technical details during this period.

5.6.5.1 RenderLights

The application of shadow map still needs toRenderLightsstart the coloring analysis from the interface. Although the logic has been introduced in the previous articleRenderLights,here we mainly focus on the application logic of shadow:

```
void FDeferredShadingSceneRenderer::RenderLights(FRHICommandListImmediate& RHICmdList,
FSortedLightSetSceneInfo &SortedLightSet, ...)
{
	(.....)

	constTArray<FSortedLightSceneInfo, SceneRenderingAllocator> &SortedLights =
	SortedLightSet.SortedLights;
	constint32 AttenuationLightStart = SortedLightSet.AttenuationLightStart;

	(.....)

	{
		SCOPED_DRAW_EVENT(RHICmdList, DirectLighting);

		FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList); EShaderPlatform
		ShaderPlatformForFeatureLevel =
GShaderPlatformForFeatureLevel[FeatureLevel];

		(.....)

		//Handling lighting with shadows.
		{
			SCOPED_DRAW_EVENT(RHICmdList, ShadowedLights);

			(.....)

			boolbDirectLighting = ViewFamily.EngineShowFlags.DirectLighting; bool
			bShadowMaskReadable =false; TRefCountPtr<IPooledRenderTarget>
			TRefCountPtr<IPooledRenderTarget>		ScreenShadowMaskTexture;
								ScreenShadowMaskSubPixelTexture;

			//Iterate over all lights and draw them with shadows and lighting functions.
			for(int32 LightIndex = AttenuationLightStart; LightIndex <
SortedLights.Num(); LightIndex++)
			{
				constFSortedLightSceneInfo& SortedLightInfo = SortedLights[LightIndex]; constFLightSceneInfo&
				LightSceneInfo = *SortedLightInfo.LightSceneInfo;

				const boolbDrawShadows = SortedLightInfo.SortKey.Fields.bShadowed &&
!ShouldRenderRayTracingStochasticRectLight(LightSceneInfo);
				boolbUsedShadowMaskTexture =false;

				(.....)
```

```
//Allocates a screen shading mask texture.
if((bDrawShadows || bDrawLightFunction || bDrawPreviewIndicator) &&
!ScreenShadowMaskTexture.IsValid())
{
    SceneContext.AllocateScreenShadowMask(RHICmdList,
ScreenShadowMaskTexture);
    bShadowMaskReadable =false;

    (......)
}

// Draws the shadow.
if (bDrawShadows)
{
    INC_DWORD_STAT(STAT_NumShadowedLights);

    constFLightOcclusionType OcclusionType =
GetLightOcclusionType(*LightSceneInfo.Proxy);

    (......)
    //Shadow map shading type.
    else// (OcclusionType == FOcclusionType::Shadowmap) {


        (......)

        //Cleaned up shadow occlusion.
        autoClearShadowMask = [&](TRefCountPtr<IPooledRenderTarget>&
InScreenShadowMaskTexture)
        {
            const boolbClearLightScreenExtentsOnly =
CVarAllowClearLightSceneExtentsOnly.GetValueOnRenderThread() &&
SortedLightInfo.SortKey.Fields.LightType != LightType_Directional;
            boolbClearToWhite = !bClearLightScreenExtentsOnly;

            //Shadow rendering pass information.
            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;
            }

            //Convert RenderingPassThe render target.
            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);
                                                //The shadow mask texture is originally completely white.
                                                DrawClearQuad(RHICmdList,true, FLinearColor(1,1,
1,1),false,0,false,0);
                                            }
                                            else
                                            {
                                                LightSceneInfo.Proxy->GetScissorRect(ScissorRect,
View, View.ViewRect);
                                            }
                                        }
                                    }
                                    RHICmdList.EndRenderPass();
                                };

                                //Clean up the screen shadows.
                                ClearShadowMask(ScreenShadowMaskTexture); //
                                    Clears the screen's shadow mask sub-pixel texture.
                                if    (ScreenShadowMaskSubPixelTexture)
                                {
                                    ClearShadowMask(ScreenShadowMaskSubPixelTexture);
                                }

                                //Render shadow casters.
                                RenderShadowProjections(RHICmdList, &LightSceneInfo,
ScreenShadowMaskTexture,      ScreenShadowMaskSubPixelTexture, HairDatas,
bInjectedTranslucentVolume);
                            }

                            //Marks that the shadow mask texture is rendered.
                            bUsedShadowMaskTexture =true;
                        }

                        (.....)

                        //    Use a shadow mask texture.
                        if    (bUsedShadowMaskTexture)
                        {
                            //Copy and resolve the shadow mask texture.
                            RHICmdList.CopyToResolveTarget(ScreenShadowMaskTexture-
> GetRenderTargetItem().TargetableTexture, ScreenShadowMaskTexture-
```

```
> GetRenderTargetItem().ShaderResourceTexture, FResolveParams(FResolveRect()));
                        if(ScreenShadowMaskSubPixelTexture) {

                            RHICmdList.CopyToResolveTarget(ScreenShadowMaskSubPixelTexture-
> GetRenderTargetItem().TargetableTexture, ScreenShadowMaskSubPixelTexture-
> GetRenderTargetItem().ShaderResourceTexture, FResolveParams(FResolveRect()));
                        }

                        // ConversionScreenShadowMaskTextureTo be
                        if readable. (!bShadowMaskReadable)
                        {

RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
ScreenShadowMaskTexture->GetRenderTargetItem().ShaderResourceTexture);
                            if(ScreenShadowMaskSubPixelTexture) {


RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
ScreenShadowMaskSubPixelTexture->GetRenderTargetItem().ShaderResourceTexture);
                            }
                            bShadowMaskReadable =true;
                        }
                    }

                    (......)

                    else
                    {
                        // Calculate standard deferred lighting, Contains shadows.
                        SCOPED_DRAW_EVENT(RHICmdList, StandardDeferredLighting);
                        SceneContext.BeginRenderingSceneColor(RHICmdList,
ESimpleRenderTargetMode::EExistingColorAndDepth,
FExclusiveDepthStencil::DepthRead_StencilWrite,true);

                        //The light map may have been created by the previous light, but it is only available if valid data is written to this light source.
bUsedShadowMaskTextureTo judge rather than useScreenShadowMaskTextureAddress determination.
                        IPooledRenderTarget* LightShadowMaskTexture = nullptr; IPooledRenderTarget*
                        LightShadowMaskSubPixelTexture = nullptr; if(bUsedShadowMaskTexture) {


                            LightShadowMaskTexture = ScreenShadowMaskTexture;
                            LightShadowMaskSubPixelTexture = ScreenShadowMaskSubPixelTexture;
                        }

                        // Rendering deferred lighting, whereLightShadowMaskTextureIt's the light sourceLightSceneInfoShadow
                        if information. (bDirectLighting)
                        {
                            RenderLight(RHICmdList, &LightSceneInfo, LightShadowMaskTexture, false,true);
InHairVisibilityViews,
                        }

                        SceneContext.FinishRenderingSceneColor(RHICmdList);

                        (......)
                    }
                }
            }// shadowed lights
```

```
    }
}
```

5.6.5.2 RenderShadowProjections

In the previous section, RenderShadowProjections was called to render the screen space shadow mask texture for the light source again. The code is as follows:

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

bool FDeferredShadingSceneRenderer::RenderShadowProjections(FRHICommandListImmediate& RHICmdList,
constFLightSceneInfo* LightSceneInfo, IPooledRenderTarget* ScreenShadowMaskTexture, IPooledRenderTarget*
ScreenShadowMaskSubPixelTexture, ...) {

    (......)

    FVisibleLightInfo& VisibleLightInfo = VisibleLightInfos[LightSceneInfo->Id];

    //Calls the parent class to render the shadow caster.
    FSceneRenderer::RenderShadowProjections(RHICmdList, LightSceneInfo,
ScreenShadowMaskTexture, ScreenShadowMaskSubPixelTexture,false,false, ...);

    //Iterate over all the shadows to be cast by visible light sources.
    for(int32 ShadowIndex =0; ShadowIndex < VisibleLightInfo.ShadowsToProject.Num(); ShadowIndex++)

    {
        FProjectedShadowInfo* ProjectedShadowInfo =
        VisibleLightInfo.ShadowsToProject[ShadowIndex];

        //  Handles transparent volume shadows.
        if  (ProjectedShadowInfo->bAllocated
            &&ProjectedShadowInfo->bWholeSceneShadow !
            && ProjectedShadowInfo->bRayTracedDistanceField
            && (!LightSceneInfo->Proxy->HasStaticShadowing() || ProjectedShadowInfo-
> IsWholeSceneDirectionalShadow())) {

            (......)
        }
    }

    //Renders capsule direct shadows.
    RenderCapsuleDirectShadows(RHICmdList, *LightSceneInfo,          ScreenShadowMaskTexture,
VisibleLightInfo.CapsuleShadowsToProject,false);

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

        (......)
    }

    //  Hair shadows.
    if  (HairDatas)
    {
        RenderHairStrandsShadowMask(RHICmdList, Views, LightSceneInfo, HairDatas,
        ScreenShadowMaskTexture);
    }
```

```
    returntrue;
}

bool FSceneRenderer::RenderShadowProjections(FRHICommandListImmediate& RHICmdList,const FLightSceneInfo*
LightSceneInfo, IPooledRenderTarget* ScreenShadowMaskTexture, IPooledRenderTarget*
ScreenShadowMaskSubPixelTexture, ...)
{
    FVisibleLightInfo& VisibleLightInfo = VisibleLightInfos[LightSceneInfo->Id]; FSceneRenderTargets&
    SceneContext = FSceneRenderTargets::Get(RHICmdList);

    //Collect all shadow information for a light source so that only onePassRendering is complete.
    TArray<FProjectedShadowInfo*> DistanceFieldShadows;//Distance Field Shadows.
    TArray<FProjectedShadowInfo*> NormalShadows;//Universal shadows.

    //Collects shadow instances from a light source and puts them into distance fields or universal shadow lists by type.
    for(int32 ShadowIndex =0; ShadowIndex < VisibleLightInfo.ShadowsToProject.Num(); ShadowIndex++)

    {
        FProjectedShadowInfo* ProjectedShadowInfo =
        VisibleLightInfo.ShadowsToProject[ShadowIndex];
        // Collect distance fields or universal shadows.
        if (ProjectedShadowInfo->bRayTracedDistanceField)
        {
            DistanceFieldShadows.Add(ProjectedShadowInfo);
        }
        else
        {
            NormalShadows.Add(ProjectedShadowInfo);
            if(ProjectedShadowInfo->bAllocated && ProjectedShadowInfo-
> RenderTargets.DepthTarget
                && !bMobileModulatedProjections)
            {
                // Converts the resource status to readable.
                RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
ProjectedShadowInfo->RenderTargets.DepthTarget-
> GetRenderTargetItem().ShaderResourceTexture.GetReference());
            }
        }
    }

    //Normal shadow rendering.
    if(NormalShadows.Num() >0) {

        //Rendering shadow occlusion interface.
        autoRenderShadowMask = [&](constFHairStrandsVisibilityViews* HairVisibilityViews)

        {
            //Renders shadows for all views.
            for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                constFViewInfo& View = Views[ViewIndex];

                (.....)

                RHICmdList.SetViewport(View.ViewRect.Min.X, View.ViewRect.Min.Y,0.0f,
View.ViewRect.Max.X, View.ViewRect.Max.Y,1.0f);
                LightSceneInfo->Proxy->SetScissorRect(RHICmdList, View, View.ViewRect);
```

```
                    //Update sceneUniform Buffer. Scene-
                    >UniformBuffers.UpdateViewUniformBuffer(View);

                    //Projects all normal shadow depth buffers onto the scene.
                    for(int32 ShadowIndex =0; ShadowIndex < NormalShadows.Num();
ShadowIndex++)
                    {
                        FProjectedShadowInfo* ProjectedShadowInfo =
NormalShadows[ShadowIndex];

                        if(ProjectedShadowInfo->bAllocated && ProjectedShadowInfo-
> FadeAlphas[ViewIndex] >1.0f/256.0f)
                        {
                            // Renders point light shadows.
                            if (ProjectedShadowInfo->bOnePassPointLightShadow)
                            {
                                ProjectedShadowInfo -
> RenderOnePassPointLightProjection(RHICmdList, ViewIndex, View,
bProjectingForForwardShading, HairVisibilityData);
                            }
                            else //Universal light source shadows.
                            {
                                ProjectedShadowInfo->RenderProjection(RHICmdList, ViewIndex,
&View, this, bProjectingForForwardShading, bMobileModulatedProjections, HairVisibilityData);

                            }
                        }
                    }
                }
            };

            (.....)
            else// Renders normal shadows.
            {
                //Construction RenderingPass Render Textures areScreenShadowMaskTexture.
                information, FRHIRenderPassInfo RPInfo(ScreenShadowMaskTexture-
> 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;

                TransitionRenderPassTargets(RHICmdList, RPInfo); RHICmdList.BeginRenderPass(RPInfo,
                TEXT("RenderShadowProjection"));

                //Render shadow occlusion.
                RenderShadowMask(nullptr);

                RHICmdList.SetScissorRect(false,0,0,0,0);
                RHICmdList.EndRenderPass();
            }

            //Sub-pixel shading.
            if(!bMobileModulatedProjections && ScreenShadowMaskSubPixelTexture &&
```

```
InHairVisibilityViews)
            {
                (......)
            }
        }

        //Distance Field Shadows.
        if(DistanceFieldShadows.Num() >0) {

            (......)
        }

        returntrue;
}
```

It can be seen that RenderShadowProjections can be used to project all shadows associated with the light source into ScreenShadowMaskTexture. In addition, it may also include distance field shadows, sub-pixel shadows, transparent volume shadows, capsule shadows, height field shadows, hair shadows and other types.

5.6.5.3 FProjectedShadowInfo::RenderProjection

To explore the rendering details of ScreenShadowMaskTexture, enter FProjectedShadowInfo::RenderProjection (note that the shadow rendering stage is analyzed in FProjectedShadowInfo::RenderDepthand FProjectedShadowInfo::RenderTranslucencyDepth):

```
void FProjectedShadowInfo::RenderProjection(FRHICommandListImmediate& RHICmdList, int32 ViewIndex,const
FViewInfo* View,constFSceneRenderer* SceneRender,bool bProjectingForForwardShading,bool
bMobileModulatedProjections, ...)const {

        (......)

        FGraphicsPipelineStateInitializer GraphicsPSOInit;
        RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);

        //Check if the shadow view flag is on, if not, return directly.
        constFVisibleLightViewInfo& VisibleLightViewInfo = View-
> VisibleLightInfos[LightSceneInfo->Id]; {

            FPrimitiveViewRelevance ViewRelevance =
            VisibleLightViewInfo.ProjectedShadowViewRelevanceMap[ShadowId];
            if(ViewRelevance.bShadowRelevance ==false) {

                return;
            }
        }

        boolbCameraInsideShadowFrustum;
        TArray<FVector4, TInlineAllocator<8>> FrustumVertices; SetupFrustumForProjection(View,
        FrustumVertices, bCameraInsideShadowFrustum);

        const boolbSubPixelSupport = HairVisibilityData != nullptr; const bool
        bStencilTestEnabled = !bSubPixelSupport;
        const boolbDepthBoundsTestEnabled = IsWholeSceneDirectionalShadow() && GSupportsDepthBoundsTest
&& CVarCSMDepthBoundsTest.GetValueOnRenderThread() !=0&&
```

```
!bSubPixelSupport;

    if(!bDepthBoundsTestEnabled && bStencilTestEnabled) {

        SetupProjectionStencilMask(RHICmdList, View, ViewIndex, SceneRender, FrustumVertices,
bMobileModulatedProjections, bCameraInsideShadowFrustum);
    }

    //Rasterization status.
    GraphicsPSOInit.RasterizerState = (View->bReverseCulling || IsWholeSceneDirectionalShadow()) ?
TStaticRasterizerState<FM_Solid,CM_CCW>::GetRHI() : TStaticRasterizerState<FM_Solid,CM_CW>::GetRHI();


    //Depth-stencil state.
    GraphicsPSOInit.bDepthBounds if       =  bDepthBoundsTestEnabled;
    (bDepthBoundsTestEnabled) {

        GraphicsPSOInit.DepthStencilState               = TStaticDepthStencilState<false,
        CF_Always>::GetRHI();
    }
    else if(bStencilTestEnabled) {

        if(GStencilOptimization) {

            // No depth test or writes, zero the stencil
            // Note: this will disable hi-stencil on many GPUs, but still seems // to be faster. However,
            early stencil still works
            GraphicsPSOInit.DepthStencilState =
                TStaticDepthStencilState< false,
                CF_Always,
                true, CF_NotEqual, SO_Zero, SO_Zero, SO_Zero, false,
                CF_Always, SO_Zero, SO_Zero, SO_Zero, 0xff,0xff

                > ::GetRHI();
        }
        else
        {
            // no depth test or writes, Test stencil for non-zero.
            GraphicsPSOInit.DepthStencilState =
                TStaticDepthStencilState< false,
                CF_Always,
                true, CF_NotEqual, SO_Keep, SO_Keep, SO_Keep, false,
                CF_Always, SO_Keep, SO_Keep, SO_Keep, 0xff,0xff

                > ::GetRHI();
        }
    }
    else
    {
        GraphicsPSOInit.DepthStencilState = TStaticDepthStencilState<false, CF_Always>::GetRHI();

    }

    //Get the mixed state, see the following analysis for the specific logic.
    GraphicsPSOInit.BlendState = GetBlendStateForProjection(bProjectingForForwardShading,
    bMobileModulatedProjections);
    GraphicsPSOInit.PrimitiveType = IsWholeSceneDirectionalShadow() ? PT_TriangleStrip : PT_TriangleList;
```

```
	{
		uint32 LocalQuality = GetShadowQuality();

		//Handles shadow quality and shadow resolution.
		if(LocalQuality >1) {

			if(IsWholeSceneDirectionalShadow() && CascadeSettings.ShadowSplitIndex >0) {

				// adjust kernel size so that the penumbra size of distant splits will
better match up with the closer ones
				const floatSizeScale = CascadeSettings.ShadowSplitIndex /
FMath::Max(0.001f, CVarCSMSplitPenumbraScale.GetValueOnRenderThread());
			}
			else if(LocalQuality >2&& !bWholeSceneShadow) {

				staticautoCVarPreShadowResolutionFactor =
IConsoleManager::Get().FindTConsoleVariableDataFloat(TEXT("r.Shadow.PreShadowResolutionFac tor"));

				constint32 TargetResolution = bPreShadow ? FMath::TruncToInt(512*
CVarPreShadowResolutionFactor->GetValueOnRenderThread()):512;

				int32 Reduce =0;

				{
					int32 Res = ResolutionX;

					while(Res < TargetResolution) {

						Res *=2;
						+ +Reduce;
					}
				}

				// Never drop to quality 1 due to low resolution, aliasing is too bad LocalQuality =
				FMath::Clamp((int32)LocalQuality - Reduce,3,5);
			}
		}

		//Binding Vertex Layout
		GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                    =
		GetVertexDeclarationFVector4();
		//Binding Shadow Castingshader.
		BindShadowProjectionShaders(LocalQuality, RHICmdList, GraphicsPSOInit, ViewIndex,
* View, HairVisibilityData, this, bMobileModulatedProjections);

		if(bDepthBoundsTestEnabled) {

			SetDepthBoundsTest(RHICmdList, CascadeSettings.SplitNear,
CascadeSettings.SplitFar, View->ViewMatrices.GetProjectionMatrix());
		}

		RHICmdList.SetStencilRef(0);
	}

	//Perform screen-space drawing.
	if(IsWholeSceneDirectionalShadow())//Full-view directional shadows. {
```

```
            //Set Vertexbuffer.
            RHICmdList.SetStreamSource(0, //Call    GClearVertexBuffer.VertexBufferRHI,              0);
            draw.
            RHICmdList.DrawPrimitive(0,2,              1);
        }
        else
        {
            //    Dynamically create vertex buffers.
            FRHIResourceCreateInfo          CreateInfo;
            FVertexBufferRHIRef VertexBufferRHI = RHICreateVertexBuffer(sizeof(FVector4) * FrustumVertices.Num(),
BUF_Volatile, CreateInfo);
            //Upload data to the vertex buffer.
            void* VoidPtr = RHILockVertexBuffer(VertexBufferRHI,0,sizeof(FVector4) * FrustumVertices.Num(),
            RLM_WriteOnly);
            FPlatformMemory::Memcpy(VoidPtr, FrustumVertices.GetData(),sizeof(FVector4) *
            FrustumVertices.Num());
            RHIUnlockVertexBuffer(VertexBufferRHI);


            //Set up the vertex buffer and draw it.
            RHICmdList.SetStreamSource(0, VertexBufferRHI,0); // Draw the frustum using the projection shader..
            RHICmdList.DrawIndexedPrimitive(GCubeIndexBuffer.IndexBufferRHI,0,0,8,0,12,

1);
            //Releases a vertex buffer.
            VertexBufferRHI.SafeRelease();
        }

        //Clean up the template buffer0.
        if(!bDepthBoundsTestEnabled && bStencilTestEnabled) {

            if(!GStencilOptimization) {

                DrawClearQuad(RHICmdList,false, FLinearColor::Transparent,false,0,true,
0);
            }
        }
}
```

The above code calls several key interfaces:GetBlendStateForProjectionand BindShadowProjectionShaders,which are analyzed below:

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

FRHIBlendState*FProjectedShadowInfo::GetBlendStateForProjection(bool
bProjectingForForwardShading,boolbMobileModulatedProjections)const {

    returnGetBlendStateForProjection(
        GetLightSceneInfo().GetDynamicShadowMapChannel(),
        IsWholeSceneDirectionalShadow(),
        CascadeSettings.FadePlaneLength >0&& !bRayTracedDistanceField,
        bProjectingForForwardShading,
        bMobileModulatedProjections);
}

FRHIBlendState*FProjectedShadowInfo::GetBlendStateForProjection(
    int32 ShadowMapChannel,
```

```
    bool bIsWholeSceneDirectionalShadow,
    bool bUseFadePlane,
    bool bProjectingForForwardShading,
    bool bMobileModulatedProjections)
{
    // Deferred rendering mode has4Channels(RGBA).
    // CSMand per-object shadows are stored in separate channels to allowCSMTransition to precomputed shadows and per-object shadows that can exceed the fade distance.
    // Subsurface shadows require an additional pass.

    FRHIBlendState* BlendState = nullptr;

    // Forward rendering mode
    if (bProjectingForForwardShading)
    {
        (.....)
    }
    else // Deferred Rendering Mode
    {
        // Light attenuation (ScreenShadowMaskTexture)Channel assignment: R:
        // WholeSceneShadows, non SSS
        // G: WholeSceneShadows, SSS
        // B: non WholeSceneShadows, non SSS A: non
        // WholeSceneShadows, SSS
        //
        // Analysis of the above terms:
        // SSS:Subsurface scattering material.
        // non SSS:Opaque object shadows. //
        WholeSceneShadows:Directional LightCSM.
        // non WholeSceneShadows:Spotlight, per-object shadows, transparent lighting, omnidirectional parallel light (omni-directional
lights).

        if(bIsWholeSceneDirectionalShadow)//Full-view directional shadows. {

            //Mixed logic requires matchingFCompareFProjectedShadowInfoBySplitIndexorder. For example, the fading plane blending mode
The method requires shadows to be rendered first.
            //Full-view shadows are only enabledRandGaisle.
            if(bUseFadePlane)//Use a fading plane. {

                // alphaChannels are used to transition between levels. Not requiredBO_Min,becauseBandAChannel to be
                used as transparent shadows. BlendState = TStaticBlendState<CW_RG, BO_Add, BF_SourceAlpha,
BF_InverseSourceAlpha>::GetRHI();
            }
            else//No fading plane is used.
            {
                // CSMThe first level does not require a transition.BO_MinIt is used to combine multiple shadow passes.
                BlendState = TStaticBlendState<CW_RG, BO_Min, BF_One, BF_One>::GetRHI();
            }
        }
        else //Non-panoramic directional light shadows.
        {
            if(bMobileModulatedProjections)//Color modulated shadows. {

                // Color modulated shadows, ignoredAlpha.
                BlendState = TStaticBlendState<CW_RGB, BO_Add, BF_Zero, BF_SourceColor,
BO_Add, BF_Zero, BF_One>::GetRHI();
            }
            else //Non-color modulated shadows.
            {
```

```
                    // BO_MinIt is used to combine multiple shadow passes.
                    BlendState = TStaticBlendState<CW_BA, BO_Min, BF_One, BF_One, BO_Min,
BF_One,                 BF_One>::GetRHI();
                }
            }
        }

        returnBlendState;
}

//Binding Shadow Castingshader.
static void BindShadowProjectionShaders(int32 Quality, FRHICommandList& RHICmdList, FGraphicsPipelineStateInitializer
GraphicsPSOInit, int32 ViewIndex,constFViewInfo& View, constFHairStrandsVisibilityData* HairVisibilityData,const
FProjectedShadowInfo* ShadowInfo,boolbMobileModulatedProjections) {


        if(HairVisibilityData) {

            (......)

            return;
        }

        //  Transparent shadow.
        if  (ShadowInfo->bTranslucentShadow)
        {
            (......)
        }
        //  Full-view directional shadows.
        else  if(ShadowInfo->IsWholeSceneDirectionalShadow())
        {
            //  PCFSoft shadows.
            if(CVarFilterMethod.GetValueOnRenderThread() ==1) {

                if(ShadowInfo->CascadeSettings.FadePlaneLength >0)
                    BindShaderShaders<FShadowProjectionNoTransformVS,
TDirectionalPercentageCloserShadowProjectionPS<5,true> >(RHICmdList, GraphicsPSOInit, ViewIndex, View,
HairVisibilityData, ShadowInfo);
                else
                    BindShaderShaders<FShadowProjectionNoTransformVS,
TDirectionalPercentageCloserShadowProjectionPS<5,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View,
HairVisibilityData, ShadowInfo);
            }
            else if(ShadowInfo->CascadeSettings.FadePlaneLength >0) {

                if(ShadowInfo->bTransmission) {

                    (......)
                }
                else
                {
                    switch  (Quality)
                    {
                    case1:    BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<1,true> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case  2: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<2,true> >(RHICmdList, GraphicsPSOInit, ViewIndex, View,
```

```
HairVisibilityData,            ShadowInfo);break;
                    case    3: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<3,true> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case    4: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<4,true> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case    5: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<5,true> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
ShadowInfo);break;
                    default:
                        check(0);
                    }
                }
            }
            else
            {
                if(ShadowInfo->bTransmission) {

                    (......)
                }
                else
                {
                    switch    (Quality)
                    {
                    case1:     BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<1,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case    2: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<2,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case    3: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<3,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case    4: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<4,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case    5: BindShaderShaders<FShadowProjectionNoTransformVS,
TShadowProjectionPS<5,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
ShadowInfo);break;
                    default:
                        check(0);
                    }
                }
            }
        }
        //Local light sources/per-object shadows.
        else
        {
            if(bMobileModulatedProjections) {

                (......)
            }
            else if(ShadowInfo->bTransmission) {

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

```
            {
                if(CVarFilterMethod.GetValueOnRenderThread() ==1&& ShadowInfo-
> GetLightSceneInfo().Proxy->GetLightType() == LightType_Spot)
                {
                    BindShaderShaders<FShadowVolumeBoundProjectionVS,
TSpotPercentageCloserShadowProjectionPS<5,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View,
HairVisibilityData, ShadowInfo);
                }
                else
                {
                    switch    (Quality)
                    {
                    case1:      BindShaderShaders<FShadowVolumeBoundProjectionVS,
TShadowProjectionPS<1,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case   2: BindShaderShaders<FShadowVolumeBoundProjectionVS,
TShadowProjectionPS<2,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case   3: BindShaderShaders<FShadowVolumeBoundProjectionVS,
TShadowProjectionPS<3,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case   4: BindShaderShaders<FShadowVolumeBoundProjectionVS,
TShadowProjectionPS<4,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
                        ShadowInfo);break;
                    case   5: BindShaderShaders<FShadowVolumeBoundProjectionVS,
TShadowProjectionPS<5,false> >(RHICmdList, GraphicsPSOInit, ViewIndex, View, HairVisibilityData,
ShadowInfo);break;
                    default:
                        check(0);
                    }
                }
            }
        }
}

//Binding Shadow CastingVSandPS.
template<typename VertexShaderType, typename PixelShaderType> static void BindShaderShaders
(FRHICommandList& RHICmdList, FGraphicsPipelineStateInitializer& GraphicsPSOInit, int32 ViewIndex,const
FViewInfo& View,constFHairStrandsVisibilityData* HairVisibilityData,constFProjectedShadowInfo* ShadowInfo)


{
    TShaderRef<VertexShaderType> VertexShader = View.ShaderMap-
> GetShader<VertexShaderType>();
    TShaderRef<PixelShaderType> PixelShader = View.ShaderMap->GetShader<PixelShaderType>
();

    //Bindingshaderand render status.
    GraphicsPSOInit.BoundShaderState.VertexShaderRHI          = VertexShader.GetVertexShader();
    GraphicsPSOInit.BoundShaderState.PixelShaderRHI           =  PixelShader.GetPixelShader();
    SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);

    //set upshaderparameter. VertexShader->SetParameters(RHICmdList, View, ShadowInfo);
    PixelShader->SetParameters(RHICmdList, ViewIndex, View, HairVisibilityData, ShadowInfo);


}
```

5.6.5.4 FShadowProjectionNoTransformVS and TShadowProjectionPS

As can be seen in the previous section, different types of shadows will call different VS and PS. Taking the representative panoramic shadow as an example, we analyze the FShadowProjectionNoTransformVS and TShadowProjectionPS used in C++ and Shader logic:

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

//Shadow CastingVS.
classFShadowProjectionNoTransformVS: public FShadowProjectionVertexShaderInterface {

	DECLARE_SHADER_TYPE(FShadowProjectionNoTransformVS,Global); public:

	(.....)

	static void ModifyCompilationEnvironment(constFGlobalShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
	{
		FShadowProjectionVertexShaderInterface::ModifyCompilationEnvironment(Parameters, OutEnvironment);

		//Not usedUSE_TRANSFORM.
		OutEnvironment.SetDefine(TEXT("USE_TRANSFORM"), (uint32)0);
	}

	static bool ShouldCompilePermutation(constFGlobalShaderPermutationParameters& Parameters)

	{
		returntrue;
	}

	//set upviewofUniform Buffer.
	void SetParameters(FRHICommandList& RHICmdList, FRHIUniformBuffer* ViewUniformBuffer) {

		FGlobalShader::SetParameters<FViewUniformShaderParameters>(RHICmdList,
RHICmdList.GetBoundVertexShader(), ViewUniformBuffer);
	}

	void SetParameters(FRHICommandList& RHICmdList,constFSceneView& View,const
	FProjectedShadowInfo*)
	{
		FGlobalShader::SetParameters<FViewUniformShaderParameters>(RHICmdList,
RHICmdList.GetBoundVertexShader(), View.ViewUniformBuffer);
	}
};

//Shadow CastingPS.
template<uint32 Quality,boolbUseFadePlane=false,boolbModulatedShadows =false,bool bUseTransmission =false,bool
SubPixelShadow =false>
class TShadowProjectionPS : public FShadowProjectionPixelShaderInterface {

	DECLARE_SHADER_TYPE(TShadowProjectionPS,Global); public:


	(.....)

	TShadowProjectionPS(constShaderMetaType::CompiledShaderInitializerType& Initializer):
```

```
        FShadowProjectionPixelShaderInterface(Initializer)
    {
        ProjectionParameters.Bind(Initializer); ShadowFadeFraction.Bind(Initializer.ParameterMap,TEXT(
        "ShadowFadeFraction")); ShadowSharpen.Bind(Initializer.ParameterMap,TEXT("ShadowSharpen"));
        TransmissionProfilesTexture.Bind(Initializer.ParameterMap,

TEXT("SSProfilesTexture"));
        LightPosition.Bind(Initializer.ParameterMap, TEXT("LightPositionAndInvRadius"));

    }

    static bool ShouldCompilePermutation(constFGlobalShaderPermutationParameters& Parameters)

    {
        returnIsFeatureLevelSupported(Parameters.Platform, ERHIFeatureLevel::SM5);
    }

    //Modify the macro definition.
    static void ModifyCompilationEnvironment(constFGlobalShaderPermutationParameters& Parameters,
FShaderCompilerEnvironment& OutEnvironment)
    {

FShadowProjectionPixelShaderInterface::ModifyCompilationEnvironment(Parameters,OutEnvironment);

        OutEnvironment.SetDefine(TEXT("SHADOW_QUALITY"), Quality); OutEnvironment.SetDefine(TEXT(
        "SUBPIXEL_SHADOW"), (uint32)(SubPixelShadow ?1:
0));
        OutEnvironment.SetDefine(TEXT("USE_FADE_PLANE"), (uint32)(bUseFadePlane ?1:0));
        OutEnvironment.SetDefine(TEXT("USE_TRANSMISSION"), (uint32)(bUseTransmission ?1:
0));
    }

    void SetParameters(
        FRHICommandList& RHICmdList,
        int32    ViewIndex,
        const FSceneView&       View,
        const    FHairStrandsVisibilityData*                HairVisibilityData,
        const    FProjectedShadowInfo* ShadowInfo)
    {
        FRHIPixelShader* ShaderRHI = RHICmdList.GetBoundPixelShader();

        //Here we will set many settings related to shadow renderingshaderBinding. (See the
        following code) FShadowProjectionPixelShaderInterface::SetParameters(RHICmdList,              ViewIndex,      View,
HairVisibilityData, ShadowInfo);

        const boolbUseFadePlaneEnable = ShadowInfo->CascadeSettings.FadePlaneLength >0;

        //set upshadervalue.
        ProjectionParameters.Set(RHICmdList, this, View, ShadowInfo, HairVisibilityData, bModulatedShadows,
        bUseFadePlaneEnable);
        constFLightSceneProxy& LightProxy = *(ShadowInfo->GetLightSceneInfo().Proxy);
        SetShaderValue(RHICmdList, ShaderRHI, ShadowFadeFraction, ShadowInfo-
> FadeAlphas[ViewIndex] );
        SetShaderValue(RHICmdList, ShaderRHI, ShadowSharpen, LightProxy.GetShadowSharpen()
* 7.0f+1.0f);
        SetShaderValue(RHICmdList, ShaderRHI, LightPosition,
FVector4(LightProxy.GetPosition(),1.0f/ LightProxy.GetRadius());
```

```
            //Binding Deferred LightsUniform Buffer. auto
            DeferredLightParameter =
            GetUniformBufferParameter<FDeferredLightUniformStruct>();
            if(DeferredLightParameter.IsBound()) {

                SetDeferredLightParameters(RHICmdList, ShaderRHI,                DeferredLightParameter,
&ShadowInfo->GetLightSceneInfo(), View);
            }

            FScene* Scene = nullptr;

            if(View.Family->Scene) {

                Scene = View.Family->Scene->GetRenderScene();
            }

            //Bind a texture resource.
            FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList); {

                constIPooledRenderTarget* PooledRT =
GetSubsufaceProfileTexture_RT((FRHICommandListImmediate&)RHICmdList);

                if(!PooledRT)
                {
                    PooledRT = GSystemTextures.BlackDummy;
                }

                constFSceneRenderTargetItem&  Item  =  PooledRT->GetRenderTargetItem();
                SetTextureParameter(RHICmdList, ShaderRHI, TransmissionProfilesTexture,
Item.ShaderResourceTexture);
            }
        }

protected:
        LAYOUT_FIELD(FShadowProjectionShaderParameters,          ProjectionParameters);
        LAYOUT_FIELD(FShaderParameter,          ShadowFadeFraction);
        LAYOUT_FIELD(FShaderParameter,          ShadowSharpen);
        LAYOUT_FIELD(FShaderParameter,          LightPosition);
        LAYOUT_FIELD(FShaderResourceParameter,              TransmissionProfilesTexture);
};

//There are many things related to setting up shadow castingshaderBinding parameters.
void FShadowProjectionShaderParameters::Set(FRHICommandList& RHICmdList, FShader* Shader, constFSceneView&
View,constFProjectedShadowInfo* ShadowInfo,const FHairStrandsVisibilityData* HairVisibilityData,bool
bModulatedShadows,bool bUseFadePlane)

{
        FRHIPixelShader* ShaderRHI = RHICmdList.GetBoundPixelShader();

        //Set the scene texture.
        SceneTextureParameters.Set(RHICmdList,              ShaderRHI,      View.FeatureLevel,
        ESceneTextureSetupMode::All);

        constFIntPoint ShadowBufferResolution = ShadowInfo->GetShadowBufferResolution();

        //    Shadow offset and size.
        if    (ShadowTileOffsetAndSizeParam.IsBound())
        {
```

```
            FVector2DInverseShadowBufferResolution(1.0f/ ShadowBufferResolution.X,1.0f/
            ShadowBufferResolution.Y);
            FVector4ShadowTileOffsetAndSize(
                  (ShadowInfo->BorderSize + ShadowInfo->X) * InverseShadowBufferResolution.X, (ShadowInfo-
                  >BorderSize + ShadowInfo->Y) * InverseShadowBufferResolution.Y, ShadowInfo->ResolutionX
                                                *InverseShadowBufferResolution.X,
                  ShadowInfo->ResolutionY       *InverseShadowBufferResolution.Y);
            SetShaderValue(RHICmdList, ShaderRHI,           ShadowTileOffsetAndSizeParam,
            ShadowTileOffsetAndSize);
      }

      //  Sets the transformation matrix from screen coordinates to shadow depth texture
      if  coordinates. (bModulatedShadows)
      {
            constFMatrix ScreenToShadow = ShadowInfo->GetScreenToShadowMatrix(View,0,0,
ShadowBufferResolution.X, ShadowBufferResolution.Y);
            SetShaderValue(RHICmdList, ShaderRHI, ScreenToShadowMatrix, ScreenToShadow);
      }
      else
      {
            constFMatrix ScreenToShadow = ShadowInfo->GetScreenToShadowMatrix(View);
            SetShaderValue(RHICmdList, ShaderRHI, ScreenToShadowMatrix, ScreenToShadow);
      }

      //  Shadow transition scaling.
      if  (SoftTransitionScale.IsBound())
      {
            const floatTransitionSize = ShadowInfo->ComputeTransitionSize();

            SetShaderValue(RHICmdList, ShaderRHI, SoftTransitionScale, FVector(0,0,1.0f/ TransitionSize));

      }

      //  Shadow buffer size.
      if  (ShadowBufferSize.IsBound())
      {
            FVector2DShadowBufferSizeValue(ShadowBufferResolution.X,
            ShadowBufferResolution.Y);

            SetShaderValue(RHICmdList, ShaderRHI, ShadowBufferSize,
                        FVector4(ShadowBufferSizeValue.X, ShadowBufferSizeValue.Y,1.0f/
ShadowBufferSizeValue.X,1.0f/ ShadowBufferSizeValue.Y));
      }

      //------Processing shadow depth texture------

      FRHITexture* ShadowDepthTextureValue; if(ShadowInfo-
      >RenderTargets.DepthTarget) {

            //If a depth texture exists for the shadow, it is used asPSofShadowDepthTextureValue.
            ShadowDepthTextureValue = ShadowInfo->RenderTargets.DepthTarget-
> GetRenderTargetItem().ShaderResourceTexture.GetReference(); }

      //Transparent shadows have no depth texture.
      else
      {
            ShadowDepthTextureValue = GSystemTextures.BlackDummy-
> GetRenderTargetItem().ShaderResourceTexture.GetReference();
```

```
        }

        //The sampler of the shadow depth texture, note that it is point sampling, and isClamp
        model. FRHISamplerState* DepthSamplerState =
        TStaticSamplerState<SF_Point,AM_Clamp,AM_Clamp,AM_Clamp>::GetRHI();

//Set up the shadow depth texture and sampler.
SetTextureParameter(RHICmdList, ShaderRHI, ShadowDepthTexture,
ShadowDepthTextureSampler, DepthSamplerState, ShadowDepthTextureValue);
        if(ShadowDepthTextureSampler.IsBound()) {

                RHICmdList.SetShaderSampler(
                        ShaderRHI,
                        ShadowDepthTextureSampler.GetBaseIndex(),
                        DepthSamplerState
                ) ;
        }

        //Depth offset and fade plane offset.
        SetShaderValue(RHICmdList, ShaderRHI, ProjectionDepthBias, FVector4(ShadowInfo-
> GetShaderDepthBias(), ShadowInfo->GetShaderSlopeDepthBias(), ShadowInfo-
> GetShaderReceiverDepthBias(), ShadowInfo->MaxSubjectZ - ShadowInfo->MinSubjectZ));
        SetShaderValue(RHICmdList, ShaderRHI, FadePlaneOffset, ShadowInfo-
> CascadeSettings.FadePlaneOffset);

        if(InvFadePlaneLength.IsBound() && bUseFadePlane) {

                SetShaderValue(RHICmdList, ShaderRHI, InvFadePlaneLength,1.0f/ShadowInfo-
> CascadeSettings.FadePlaneLength); }



        //    The direction or position of the light source.
        if    (LightPositionOrDirection.IsBound())
        {
                constFVector LightDirection = ShadowInfo->GetLightSceneInfo().Proxy-
> GetDirection();
                constFVector LightPosition = ShadowInfo->GetLightSceneInfo().Proxy-
> GetPosition();
                const boolbIsDirectional = ShadowInfo->GetLightSceneInfo().Proxy->GetLightType() == LightType_Directional;

                SetShaderValue(RHICmdList, ShaderRHI, LightPositionOrDirection, bIsDirectional ? FVector4(LightDirection,0
) : FVector4(LightPosition,1));
        }

        (......)

        //Per-object shadow parameters.
        SetShaderValue(RHICmdList, ShaderRHI, PerObjectShadowFadeStart, ShadowInfo-
>PerObjectShadowFadeStart);
        SetShaderValue(RHICmdList, ShaderRHI, InvPerObjectShadowFadeLength, ShadowInfo-
>InvPerObjectShadowFadeLength); }
```

The above is a detailed analysis of the logic of FShadowProjectionNoTransformVS and TShadowProjectionPS at the C++ layer. Let's turn to their corresponding Shader codes:

```
// Engine\Shaders\Private\ShadowProjectionVertexShader.usf

#include"Common.ush"

#ifndefUSE_TRANSFORM
	#defineUSE_TRANSFORM 1
#endif

#if	USE_TRANSFORM
	float4 StencilingGeometryPosAndScale;
#endif

//VSMain entrance.
void Main(in float4 InPosition : ATTRIBUTE0, out float4 OutPosition : SV_POSITION) {

	#ifUSE_TRANSFORM//Use transformations. //
		Convert object position to clip space.
		float3 WorldPosition = InPosition.xyz * StencilingGeometryPosAndScale.w +
		StencilingGeometryPosAndScale.xyz;
		OutPosition = mul(float4(WorldPosition,1), View.TranslatedWorldToClip);
	#else	//No transformation is used.
		//	The object's position is already in clip space, no further transformation is needed.
		OutPosition = float4(InPosition.xyz,1);
	#endif
}


// Engine\Shaders\Private\ShadowProjectionPixelShader.usf

(.....)

float	ShadowFadeFraction;
float	ShadowSharpen;
float4	LightPositionAndInvRadius;

float	PerObjectShadowFadeStart;
float	InvPerObjectShadowFadeLength;

float4	LightPositionOrDirection;
float	ShadowReceiverBias;

#ifUSE_FADE_PLANE || SUBPIXEL_SHADOW float
			FadePlaneOffset;
	float	InvFadePlaneLength;
	uint bCascadeUseFadePlane;
#endif

#if	USE_PCSS
	// PCSS specific parameters. //
			- x: tan(0.5 * Directional Light Angle) in shadow projection space;
	//	- y: Max filter size in shadow tile UV space. PCSSParameters;
	float4
#endif

float4	ModulatedShadowColor;
float4	ShadowTileOffsetAndSize;
```

```
(.....)

//PSMain entrance.
void Main(in float4 SVPos : SV_POSITION,
                out float4 OutColor : SV_Target0)
{
    #ifUSE_FADE_PLANE
    const boolbUseFadePlane=true;
    #endif

    constFPQMPContext PQMPContext = PQMPInit(SVPos.xy);
    float2 ScreenUV = float2(SVPos.xy * View.BufferSizeAndInvSize.zw); floatSceneW =
    CalcSceneDepth(ScreenUV);

    (......)

    //Calculate screen space/shadow space/world space coordinates.
    float4 ScreenPosition = float4(((ScreenUV.xy - View.ScreenPositionScaleBias.wz ) /
View.ScreenPositionScaleBias.xy) * SceneW, SceneW,1);
    float4 ShadowPosition = mul(ScreenPosition, ScreenToShadowMatrix); float3 WorldPosition
    = mul(ScreenPosition, View.ScreenToWorld).xyz;

    //Calculate the coordinates of the shadow space (where the shadow clip space coordinates need to be divided byw,
    to convert to screen space for shadows) floatShadowZ = ShadowPosition.z; ShadowPosition.xyz /=
    ShadowPosition.w;

#if  MODULATED_SHADOWS
    ShadowPosition.xy      * = ShadowTileOffsetAndSize.zw;
    ShadowPosition.xy      + = ShadowTileOffsetAndSize.xy;
#endif

    // PCSSSoft shadows.
#if  USE_PCSS
    float3 ScreenPositionDDX = DDX(ScreenPosition.xyz); float3
    ScreenPositionDDY = DDY(ScreenPosition.xyz);
    float4 ShadowPositionDDX = mul(float4(ScreenPositionDDX,0), ScreenToShadowMatrix); float4
    ShadowPositionDDY = mul(float4(ScreenPositionDDY,0), ScreenToShadowMatrix);
    #if SPOT_LIGHT_PCSS
        ShadowPositionDDX.xyz -= ShadowPosition.xyz * ShadowPositionDDX.w;
        ShadowPositionDDY.xyz -= ShadowPosition.xyz * ShadowPositionDDY.w;
    #endif
#endif

    //Adjust the depth of shadow space (light space) to0.99999f,Because the shadow depth buffer cannot be cleared1It can also force pixels in the light space far plane to
not be occluded.
    floatLightSpacePixelDepthForOpaque = min(ShadowZ,0.99999f); //SSSThe depth cannot be
    adjusted because the subsurface gradient must extend beyond the far plane. float
    LightSpacePixelDepthForSSS = ShadowZ;

    //Per-object shadows are transitioned before being clipped, starting with the plane1The end plane is0.
    floatPerObjectDistanceFadeFraction =1.0f- saturate((LightSpacePixelDepthForSSS - PerObjectShadowFadeStart) *
    InvPerObjectShadowFadeLength);

    //Initialize shadow and projection parameters.
    floatShadow =1; float
    SSSTransmission =1;

    floatBlendFactor =1;
```

```
//Unfiltered shadow casting.
#ifUNFILTERED_SHADOW_PROJECTION {

        //Directly compare the pixel depth of the adjusted light space (shadow space) and the shadow depth of the corresponding pixel coordinatesr
        Channel value. //If the former < the latter, it means it is not blocked.Shadow=1;on the contrary,Shadow=0.
        Shadow = LightSpacePixelDepthForOpaque < Texture2DSampleLevel(ShadowDepthTexture,
ShadowDepthTextureSampler, ShadowPosition.xy,0).r;
}
//Transparent shadow.
#elifAPPLY_TRANSLUCENCY_SHADOWS {

        Shadow = CalculateTranslucencyShadowing(ShadowPosition.xy, ShadowZ);
}
// PCSSSoft shadows.
#elifUSE_PCSS
{
        FPCSSSamplerSettings Settings;

        #ifSPOT_LIGHT_PCSS
        {
                floatCotanOuterCone = DeferredLightUniforms.SpotAngles.x * rsqrt(1.-
DeferredLightUniforms.SpotAngles.x * DeferredLightUniforms.SpotAngles.x);
                floatWorldLightDistance = dot(DeferredLightUniforms.Direction,
DeferredLightUniforms.Position - WorldPosition);
                Settings.ProjectedSourceRadius =0.5* DeferredLightUniforms.SourceRadius *
CotanOuterCone/WorldLightDistance;
                Settings.TanLightSourceAngle =          0;
        }
        #else
        {
                Settings.ProjectedSourceRadius          =0;
                Settings.TanLightSourceAngle =          PCSSParameters.x;
        }
        #endif
        Settings.ShadowDepthTexture = ShadowDepthTexture;
        Settings.ShadowDepthTextureSampler = ShadowDepthTextureSampler;
        Settings.ShadowBufferSize = ShadowBufferSize;
        Settings.ShadowTileOffsetAndSize = ShadowTileOffsetAndSize;
        Settings.SceneDepth = LightSpacePixelDepthForOpaque;
        Settings.TransitionScale          = SoftTransitionScale.z;
        Settings.MaxKernelSize          = PCSSParameters.y;
        Settings.SvPosition          = SVPos.xy;
        Settings.PQMPContext          = PQMPContext;
        Settings.DebugViewportUV          =ScreenUV;

        Shadow = DirectionalPCSS(Settings, ShadowPosition.xy, ShadowPositionDDX.xyz,
        ShadowPositionDDY.xyz);
}
//CustomPCFshadow.
#else
{
        #ifSHADING_PATH_DEFERRED && !FORWARD_SHADING FGBufferData GBufferData =
                GetGBufferData(ScreenUV); const boolbIsDirectional =
                LightPositionOrDirection.w ==0;
                constfloat3 LightDirection = bIsDirectional ? -LightPositionOrDirection.xyz :
normalize(LightPositionOrDirection.xyz - WorldPosition);
                const floatNoL = saturate(dot(GBufferData.WorldNormal, LightDirection));
```

```
            #endif

            FPCFSamplerSettings Settings;

            Settings.ShadowDepthTexture = ShadowDepthTexture;
            Settings.ShadowDepthTextureSampler = ShadowDepthTextureSampler;
            Settings.ShadowBufferSize = ShadowBufferSize;
            #ifSHADING_PATH_DEFERRED && !FORWARD_SHADING
            Settings.TransitionScale = SoftTransitionScale.z *
            lerp(ProjectionDepthBiasParameters.z,1.0, NoL);
            #else
            Settings.TransitionScale                = SoftTransitionScale.z;
            #endif
            Settings.SceneDepth         = LightSpacePixelDepthForOpaque;
            Settings.bSubsurface          = false;
            Settings.bTreatMaxDepthUnshadowed          =false;
            Settings.DensityMulConstant =0;
            Settings.ProjectionDepthBiasParameters                  =0;

            //CustomPCFshadow.
            Shadow = ManualPCF(ShadowPosition.xy, Settings);
      }
      #endif    // !USE_PCSS

      #ifUSE_FADE_PLANE || SUBPIXEL_SHADOW if
        (bUseFadePlane)
      {
            // Create a blend factor which is one before and at the fade plane, and lerps to zero at the far plane.

            BlendFactor =1.0f- saturate((SceneW - FadePlaneOffset) * InvFadePlaneLength);
      }
      #endif

      //Subsurface shadows.
      #ifFEATURE_LEVEL >= FEATURE_LEVEL_SM4 && !FORWARD_SHADING &&
!APPLY_TRANSLUCENCY_SHADOWS

            FGBufferData GBufferData = GetGBufferData(ScreenUV);

            BRANCH
            if(bIsSubsurfaceCompatible && IsSubsurfaceModel(GBufferData.ShadingModelID)) {

                  floatOpacity = GBufferData.CustomData.a;
                  floatDensity =-.05f*log(1- min(Opacity,.999f)); if
                  (GBufferData.ShadingModelID == SHADINGMODELID_HAIR ||
GBufferData.ShadingModelID == SHADINGMODELID_EYE )
                  {
                        Opacity    =  1;
                        Density    =  1;
                  }

                  floatSquareRootFilterScale = lerp(1.999f,0, Opacity); intSquareRootFilterScaleInt =int
                  (SquareRootFilterScale) +1;

                  #if UNFILTERED_SHADOW_PROJECTION
                        floatShadowMapDepth = Texture2DSampleLevel(ShadowDepthTexture,
ShadowDepthTextureSampler, ShadowPosition.xy,0).x;
                        SSSTransmission = CalculateSubsurfaceOcclusion(Density,
```

```
LightSpacePixelDepthForSSS, ShadowMapDepth.xxx).x;
                #else

                    // default code path
                    FPCFSamplerSettings Settings;

                    Settings.ShadowDepthTexture = ShadowDepthTexture;
                    Settings.ShadowDepthTextureSampler          = ShadowDepthTextureSampler;
                    Settings.ShadowBufferSize          =          ShadowBufferSize;
                    Settings.TransitionScale          = SoftTransitionScale.z;
                    Settings.SceneDepth = LightSpacePixelDepthForSSS +
ProjectionDepthBiasParameters.x;
                    Settings.bSubsurface =true; Settings.bTreatMaxDepthUnshadowed =false;
                    Settings.DensityMulConstant = Density * ProjectionDepthBiasParameters.w;
                    Settings.ProjectionDepthBiasParameters = ProjectionDepthBiasParameters.xw;


                #ifUSE_TRANSMISSION
                    if(GBufferData.ShadingModelID == SHADINGMODELID_SUBSURFACE_PROFILE) {

                        SSSTransmission = CalcTransmissionThickness(ScreenPosition.xyz,
LightPositionAndInvRadius.xyz, GBufferData, Settings);
                    }
                    else
                #endif
                    {
                        SSSTransmission = ManualPCF(ShadowPosition.xy, Settings);
                    }
                #endif
            }

        #endif

        //If not usedPCSSFor soft shadows, useShadowSharpenTo adjust the intensity of the shadow, similar toHalf Lambertapproach.
        #if   !USE_PCSS
            Shadow = saturate( (Shadow -0.5) * ShadowSharpen +0.5);
        #endif

        //Transition shadow.0is covered by shadows,1Is not occluded. No need to return color, unless the modulated shadow
        mode needs to write the scene color. floatFadedShadow = lerp(1.0f, Square(Shadow), ShadowFadeFraction *
        PerObjectDistanceFadeFraction);

#if   FORWARD_SHADING
        (.....)
#else
        //Point light shadows are written tob
        aisle. OutColor.b =      EncodeLightAttenuation(FadedShadow); =1;
        OutColor.rga
        //SSSShadow simulation, while not accurate enough, at least there is
        shadowing. OutColor.a = OutColor.b;
#endif

#ifUSE_FADE_PLANE || SUBPIXEL_SHADOW //
            in the case ofCSM,Then outputAlphaChannels to
        if    mix. (bUseFadePlane)
        {
            OutColor.a = BlendFactor;
        }
```

```
#endif

    //Modulate shadows.
#if MODULATED_SHADOWS
    OutColor.rgb = lerp(ModulatedShadowColor.rgb, float3(1,1,1), FadedShadow); OutColor.a =0;

#endif
}

(……)
```

From the above, we can see that when drawing ScreenShadowMaskTexture, it is done in screen space, and ScreenShadowMask will be superimposed multiple times (depending on the number of shadow instances of the light source), and different channels store different types of shadow information.

5.6.5.5 RenderLight

This section will analyze how to use the light source and ScreenShadowMaskTexture information. Go directly to RenderLight to analyze the shadow-related logic:

```
void FDeferredShadingSceneRenderer::RenderLight(FRHICommandList& RHICmdList,const FLightSceneInfo*
LightSceneInfo, IPooledRenderTarget* ScreenShadowMaskTexture, ...) {

    (……)

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

        (……)

        //Parallel light.
        if(LightSceneInfo->Proxy->GetLightType() == LightType_Directional) {

            (……)

            else
            {
                FDeferredLightPS::FPermutationDomain            PermutationVector;

                (……)

                //Declaring a lit pixel shaderFDeferredLightPSExamples.
                TShaderMapRef< FDeferredLightPS >PixelShader( View.ShaderMap,
PermutationVector );
                GraphicsPSOInit.BoundShaderState.PixelShaderRHI =
PixelShader.GetPixelShader();
                (……)
                SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);

                //WillScreenShadowMaskTextureBind to the pixel shader. PixelShader-
                >SetParameters(RHICmdList, View, LightSceneInfo,
ScreenShadowMaskTexture, (bHairLighting) ? &RenderLightParams : nullptr);
            }
```

```
            (.....)

            //Draws a full screen rectangle for calculating directional lighting in screen
            space. 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  //  Non-parallel light.
        {
            (.....)
        }
    }
}
```

As can be seen above, it is bound to the shader by calling itPixelShader->SetParameters.

ScreenShadowMaskTextureThe following is an analysis of the binding process:

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

void FDeferredLightPS::SetParameters(FRHICommandList& RHICmdList,constFSceneView& View, constFLightSceneInfo*
LightSceneInfo, IPooledRenderTarget* ScreenShadowMaskTexture, ...) {

    FRHIPixelShader* ShaderRHI = RHICmdList.GetBoundPixelShader();
    SetParametersBase(RHICmdList, ShaderRHI, View, ScreenShadowMaskTexture, LightSceneInfo-
>Proxy->GetIESTextureResource(), RenderLightParams);

    (.....)
}

void FDeferredLightPS::SetParametersBase(FRHICommandList& RHICmdList, FRHIPixelShader* ShaderRHI,const
FSceneView& View, IPooledRenderTarget* ScreenShadowMaskTexture, ...) {

    if(LightAttenuationTexture.IsBound()) {

        //BindingScreenShadowMaskTexturearriveLightAttenuationTexture
        SetTextureParameter(
            RHICmdList,
            ShaderRHI,
            LightAttenuationTexture, LightAttenuationTextureSampler,
            TStaticSamplerState<SF_Point,AM_Wrap,AM_Wrap,AM_Wrap>::GetRHI(),
            ScreenShadowMaskTexture? ScreenShadowMaskTexture-

> GetRenderTargetItem().ShaderResourceTexture : GWhiteTexture->TextureRHI
            ) ;
    }

    (.....)
}
```

As can be seen above,**the name of the light source's shadow occlusion texture in the shader is: LightAttenuationTexture, and the sampler is called: LightAttenuationTextureSampler.**

5.6.5.6 Shadow Shader Logic

Search for the keyword LightAttenuationTexture in all Shader files, and it is not difficult to find the relevant logic in common.ush:

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

(.....)

//Declare shadow mask variables.
Texture2D               LightAttenuationTexture;
SamplerState            LightAttenuationTextureSampler;

(.....)

//Gets the square of the light attenuation (i.e. shadowing) of a pixel.
float4GetPerPixelLightAttenuation(float2 UV) {

    returnSquare(Texture2DSampleLevel(LightAttenuationTexture,
LightAttenuationTextureSampler, UV,0)); }


(.....)
```

Continue to trackGetPerPixelLightAttenuationthe interface, and finally find the only call, in DeferredLightPixelShaders.usf:

```
// Engine\Shaders\Private\DeferredLightPixelShaders.usf

(.....)

void DeferredLightPixelMain(
#ifLIGHT_SOURCE_SHAPE >0
    float4 InScreenPosition : TEXCOORD0,
#else
    float2    ScreenUV                  :   TEXCOORD0,
    float3    ScreenVector              :   TEXCOORD1,
# endif
    float4    SVP                   :   SV_POSITION,
    out float4      OutColor            : SV_Target0
    )
{
    constfloat2 PixelPos = SVPos.xy; OutColor =0;


    (.....)

    const floatSceneDepth = CalcSceneDepth(InputParams.ScreenUV); FDeferredLightData
    LightData = SetupLightDataForStandardDeferred();

    (.....)
```

```
        floatSurfaceShadow =1.0f;
        //Note that theGetPerPixelLightAttenuation(InputParams.ScreenUV)AsGetDynamicLighting offloat4
LightAttenuationActual parameters.
        constfloat4 Radiance = GetDynamicLighting(DerivedParams.WorldPosition,
DerivedParams.CameraVector, ScreenSpaceData.GBuffer,                ScreenSpaceData.AmbientOcclusion,
ScreenSpaceData.GBuffer.ShadingModelID, LightData,
GetPerPixelLightAttenuation(InputParams.ScreenUV), RectTexture, Dither, uint2(InputParams.PixelPos),
SurfaceShadow);
        const floatAttenuation              =ComputeLightProfileMultiplier(DerivedParams.WorldPosition,
 DeferredLightUniforms.Position,             - DeferredLightUniforms.Direction,
 DeferredLightUniforms.Tangent);

        OutColor += (Radiance * Attenuation) * OpaqueVisibility;

        (......)

        OutColor.rgba *= GetExposure();
}
```

GetDynamicLightingNote that the formal parameter abovefloat4 LightAttenuationis called for GetPerPixelLightAttenuation(InputParams.ScreenUV)assignment. To understand the specific calculation logic of this parameter, we need to go deeperGetDynamicLighting:

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

float4GetDynamicLighting(..., float4 LightAttenuation, ..., inoutfloatSurfaceShadow) {

        //Here is calledGetDynamicLightingSplit(See analysis below) FDeferredLightingSplit
        SplitLighting = GetDynamicLightingSplit(
                WorldPosition, CameraVector, GBuffer, AmbientOcclusion, ShadingModelID, LightData,
                LightAttenuation, Dither, SVPos, SourceTexture, SurfaceShadow);


        returnSplitLighting.SpecularLighting + SplitLighting.DiffuseLighting;
}

FDeferredLightingSplitGetDynamicLightingSplit(..., float4 LightAttenuation, ..., inout floatSurfaceShadow)

{
        (......)

        BRANCH
        if( LightMask >0) {

                FShadowTerms Shadow;
                Shadow.SurfaceShadow      =  AmbientOcclusion;

                (......)

                //Get the shadow data item, which will be passed in hereLightAttenuation,Will continue to track later
                GetShadowTerms. GetShadowTerms(GBuffer, LightData, WorldPosition, L, LightAttenuation, Dither,
Shadow);

                SurfaceShadow = Shadow.SurfaceShadow;

                if( Shadow.SurfaceShadow + Shadow.TransmissionShadow >0)
```

```
        {
            const boolbNeedsSeparateSubsurfaceLightAccumulation =
UseSubsurfaceProfile(GBuffer.ShadingModelID);
            float3 LightColor = LightData.Color;

            (.....)

            float3 LightingDiffuse = Diffuse_Lambert( GBuffer.DiffuseColor ) * Lighting;
            LightAccumulator_AddSplit(LightAccumulator, LightingDiffuse,0.0f,0,
LightColor * LightMask * Shadow.SurfaceShadow, bNeedsSeparateSubsurfaceLightAccumulation);

            (.....)
        }
    }

    returnLightAccumulator_GetResultSplit(LightAccumulator);
}

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

    floatContactShadowLength =0.0f;
    const floatContactShadowLengthScreenScale = View.ClipToView[1][1] * GBuffer.Depth;

    BRANCH
    if(LightData.ShadowedBits) {

        //  Remap the light attenuation buffer. (SeeShadowRendering.cpp)
        //  LightAttenuationThe data layout is: LightAttenuation.x:
        //                                      Full-view directional shadows.
        //  LightAttenuation.y:                 Panoramic parallel lightSSSshadow.
        //  LightAttenuation.z:                 Lighting functions + per-object shadows.
        //  LightAttenuation.w:                 By ObjectSSSshadow.

        //From the approximateGBufferPass to get static shadows.
        floatUsesStaticShadowMap = dot(LightData.ShadowMapChannelMask, float4(1,1,1,
1));
        //Restore static shadows.
        floatStaticShadowing = lerp(1, dot(GBuffer.PrecomputedShadowFactors,
LightData.ShadowMapChannelMask), UsesStaticShadowMap);

        if(LightData.bRadialLight)//Radial Light {

            //Restore shadow data.
            //Surface shadow = (lighting function + per-object shadow) * static shadow.
            Shadow.SurfaceShadow = LightAttenuation.z * StaticShadowing;
            Shadow.TransmissionShadow = LightAttenuation.w * StaticShadowing;
            Shadow.TransmissionThickness = LightAttenuation.w;
        }
        else//Non-radial light source (parallel light)
        {
            // Remapping the light attenuation buffer (see ShadowRendering.cpp) // Also fix up the
            fade between dynamic and static shadows
            // to work with plane splits rather than spheres.

            floatDynamicShadowFraction = DistanceFromCameraFade(GBuffer.Depth, LightData,
WorldPosition, View.WorldCameraOrigin);
            // For a directional light, fade between static shadowing and the whole scene
```

```
dynamic shadowing based on distance + per object shadows
                Shadow.SurfaceShadow = lerp(LightAttenuation.x, StaticShadowing,
DynamicShadowFraction);
                // Fade between SSS dynamic shadowing and static shadowing based on distance
                Shadow.TransmissionShadow = min(lerp(LightAttenuation.y, StaticShadowing,
DynamicShadowFraction), LightAttenuation.w);

                Shadow.SurfaceShadow *= LightAttenuation.z;
                Shadow.TransmissionShadow *= LightAttenuation.z;

                // Need this min or backscattering will leak when in shadow which cast by non
perobject shadow(Only for directional light)
                Shadow.TransmissionThickness = min(LightAttenuation.y, LightAttenuation.w);
            }

            FLATTEN
            if(LightData.ShadowedBits >1&& LightData.ContactShadowLength >0) {

                ContactShadowLength = LightData.ContactShadowLength *
(LightData.ContactShadowLengthInWS?1.0f: ContactShadowLengthScreenScale);
            }
        }

        //Contact shadow.
#if     SUPPORT_CONTACT_SHADOWS
        if((LightData.ShadowedBits <2&& (GBuffer.ShadingModelID == SHADINGMODELID_HAIR))
            || GBuffer.ShadingModelID == SHADINGMODELID_EYE)
        {
            ContactShadowLength =0.2* ContactShadowLengthScreenScale;
        }

        #if   MATERIAL_CONTACT_SHADOWS
            ContactShadowLength =0.2* ContactShadowLengthScreenScale;
        #endif

        BRANCH
        if(ContactShadowLength >0.0) {

            floatStepOffset = Dither -0.5;
            floatContactShadow = ShadowRayCast( WorldPosition + View.PreViewTranslation, L, ContactShadowLength,8
, StepOffset );

            Shadow.SurfaceShadow *= ContactShadow;

            FLATTEN
            if( GBuffer.ShadingModelID == SHADINGMODELID_HAIR ||                GBuffer.ShadingModelID ==
            SHADINGMODELID_EYE )
            {
                const boolbUseComplexTransmittance =
(LightData.HairTransmittance.ScatteringComponent & HAIR_COMPONENT_MULTISCATTER) >0;
                if(!bUseComplexTransmittance) {

                    Shadow.TransmissionShadow *= ContactShadow;
                }
            }
            else
                Shadow.TransmissionShadow *= ContactShadow *0.5+0.5;
        }
```

```
#endif

    Shadow.HairTransmittance = LightData.HairTransmittance;
    Shadow.HairTransmittance.OpaqueVisibility = Shadow.SurfaceShadow;
}
```

When calculating surface shadows, directly sample the value of LightAttenuationTexture once, restore the parameters of various shadows, and finally apply them to lighting calculations.

5.6.5.7 Summary of Shadow Application

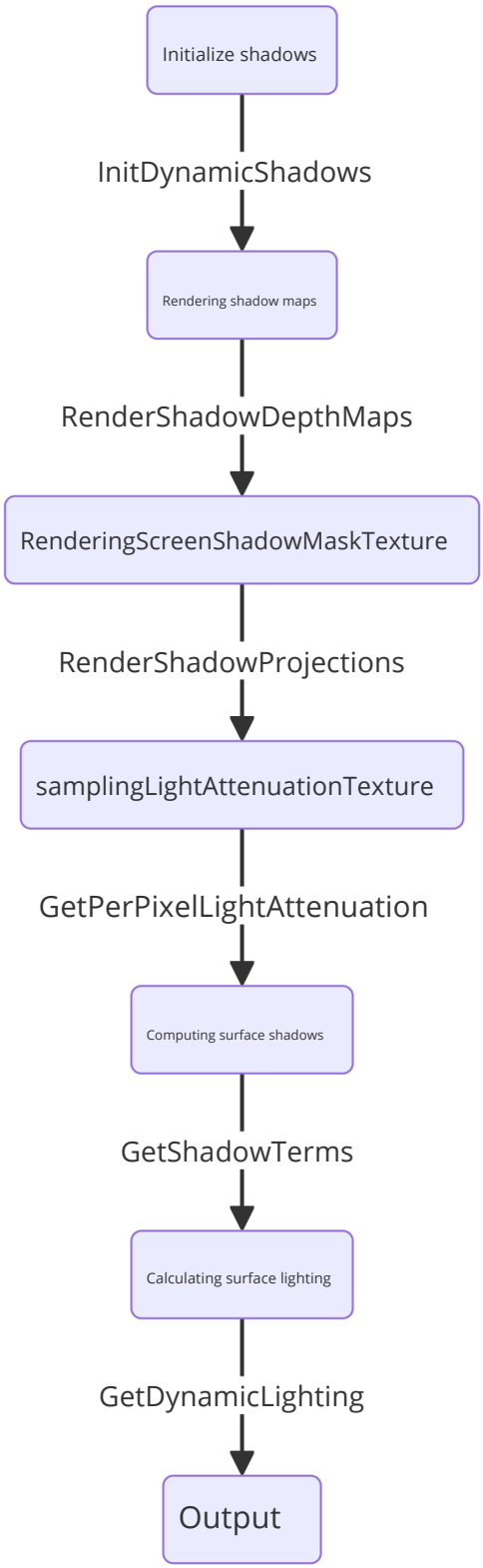
There are several important steps and additional notes to focus on during the shadow application phase:

- The ScreenShadowMaskTexture is rendered once for each light that has shadows enabled and has valid shadows.
- Before rendering ScreenShadowMaskTexture, ClearShadowMask will be called to clear ScreenShadowMaskTexture to ensure that ScreenShadowMaskTexture is in its original state (all white).
- Call RenderShadowProjections to project all shadow instances under the light source and overlay them onto the ScreenShadowMaskTexture in**the screen space**.The purpose of this is similar to deferred shading, merging multiple shadows into the view space of the View in advance, so that only one sampling is needed in the subsequent lighting stage to obtain shadow information, improving rendering efficiency. The disadvantage is that an extra Pass is
  required to merge the shadows of the light source, which increases the Draw Call and a small amount of video memory consumption.
- In addition to regular depth map shadows, ScreenShadowMaskTexture may also contain distance field shadows, subsurface shadows, transparent volume shadows, capsule shadows, height field shadows, hair shadows, RT shadows, etc. Each of its channels has a special purpose
  for subdividing and storing different types of shadow information.
- The shader that renders ScreenShadowMaskTexture supports three filtering methods when filtering the shadow map: no filtering, PCSS, and custom PCF.
- In the shader of lighting calculation, ScreenShadowMaskTexture is bound to the variable of LightAttenuationTexture (light attenuation texture), that is, the ScreenShadowMaskTexture of the C++ layer is actually the LightAttenuationTexture of the lighting shader.
- In the lighting shader file DeferredLightPixelShaders.usf, when calculating dynamic lighting, the value of GetPerPixelLightAttenuation() is called only once (that is, the LightAttenuationTexture is sampled only once) to calculate the lighting result of the object surface affected by the shadow.

5.6.6 UE Shadow Summary

The complexity of UE's shadow system far exceeded the author's original imagination. The shadow chapter only analyzes the main rendering process and basic technology of shadows, and does not include other types of shadows. It has more than 20,000 words, which is far beyond the original conception and plan.

The following figure summarizes the main process and key calls of shadow rendering:



The most outstanding feature is the addition of the rendering step of ScreenShadowMaskTexture, which makes full use of the geometric data, special blending mode and sophisticated channel allocation of GBuffer to render the complex shadow data of the light source into the light

attenuation texture of the screen space in advance, so that when calculating the lighting, only the light attenuation texture needs to be sampled once to reconstruct the surface shadow.

It should be pointed out that adding an additional Pass to render ScreenShadowMaskTexture will increase Draw Calls and increase video memory consumption, which may result in performance degradation for scenes with simple lighting and shadows.

5.7 Summary

This article mainly explains the rendering process and main algorithms of UE's direct light and shadow, so that readers can have a general understanding of the rendering of light and shadow. As for more technical details and principles, readers need to study the UE source code themselves.

This article is more than 50,000 words long, making it the longest article in this series, but it also took the shortest time to write. May is indeed a productive month.

Just yesterday (2021.5.27), the long-awaited UE5 finally released a preview version. The author downloaded it and briefly experienced it as soon as possible to appreciate the charm of UE5's nextgeneration lighting and shadow.

UE5 cool startup screen.

The author may take the time to write a UE5 special to analyze the technical principles and implementation details of Lumen and Nanite used in it.

5.7.1 Thoughts on this article

As usual, this article also arranges some small thoughts to help understand and deepen the mastery and understanding of UE BasePass and LightingPass:

- Please briefly describe the types of light sources and the main rendering process of UE. Please
- briefly describe the types of UE's shadows and the main rendering process. Please fix the
- problem that transparent objects cannot be illuminated by area lights in the official version of UE.
- Expand the number of UE lighting channels.
- Please implement per-grid virtual light sources so that each grid can be lit individually, and support opaque, masked, and translucent objects.

References

- Unreal Engine 4 Sources Unreal
- Engine 4 Documentation
- Rendering and Graphics
- Graphics Programming Overview
- Materials
- Unreal Engine 4 Rendering
- Rendering - Schematic Overview
- UE4 Render System Sheet
- Lighting the Environment
- Shadow Casting
- Learn the principles and implementation of PBR from the shallow to the deep
- Real Shading in Unreal Engine 4
- Lighting the Environment
- Reflective Shadow Map
- RSM System
- HOW UNREAL RENDERS A FRAME

- Unreal Frame Breakdown
- LaTeX/Mathematics
- DECIMA ENGINE: ADVANCES IN LIGHTING AND AA
- Real-Time Polygonal-Light Shading with Linearly Transformed Cosines
- Cascaded Shadow Maps

https://github.com/pe7yu