### 像素着色器计算过程

注：在条件分支（如果/否则）中，空行表示跳出分支，无空行表示仍然在分支中。

##### 计算材质参数

具体过程略，中间调用了CalcMaterialParameters函数。

half3 BaseColor = GetMaterialBaseColor(MaterialParameters);

half Metallic = GetMaterialMetallic(MaterialParameters);

half Specular = GetMaterialSpecular(MaterialParameters);

half Roughness = GetMaterialRoughness(MaterialParameters);

half Opacity = GetMaterialOpacity(MaterialParameters);

half3 Emissive = GetMaterialEmissive(MaterialParameters);

##### 计算材质漫反射颜色及高光色

half3 DiffuseColor;

half SpecularColor;(nonmetal) / half3 SpecularColor;(metal)

如果材质为非金属

DiffuseColor = BaseColor;

SpecularColor = 0.04;

否则

half DielectricSpecular = 0.08 \* Specular; // 介质高光系数

DiffuseColor = BaseColor – BaseColor \* Metallic; // 1个MAD（乘加指令）运算

SpecularColor = (DielectricSpecular – DielectricSpecular \* Metallic)

+ BaseColor \* Metallic; // 2个MAD运算

如果材质完全粗糙

// 下面是将参数roughness = 1，NoV = 1代入函数EnvBRDFApprox得到的结果

DiffuseColor += Specular \* 0.45;

SpecularColor = 0;

否则

half NoV = max(dot(MaterialParameters.WorldNormal, MaterialParameters.CameraVector), 0);

如果材质为非金属

SpecularColor = EnvBRDFApproxNonmetal(Roughness, Nov);

否则

SpecularColor = EnvBRDFApprox (SpecularColor, Roughness, Nov);

##### 计算环境光（非方向光）作用于像素点的颜色和间接辐射亮度

half3 Color = 0;

half IndirectIrradiance = 0;

如果使用光照贴图

float2 LightmapUV0;

float2 LightmapUV1;

// 根据顶点到像素点的插值获取光照贴图的UV

GetLightMapCoordinates(Interpolants, LightmapUV0, LightmapUV1);

half4 LightmapColor = GetLightMapColorLQ(LightmapUV0, LightmapUV1, MaterialParameters.WorldNormal);

Color += LightmapColor.rgb \* DiffuseColor;

IndirectIrradiance = LightmapColor.a;

否则，如果使用顶点缓存的非方向光照

如果材质混合模式为Masked或Solid（Solid似乎是不透明Opaque？）

// 考虑到不透明物体，计算时需要使用法线

FTwoBandSHVectorRGB PointIndirectLighting;

PointIndirectLighting.R.V = IndirectLightingSHCoefficients[0];

PointIndirectLighting.G.V = IndirectLightingSHCoefficients[1];

PointIndirectLighting.B.V = IndirectLightingSHCoefficients[2];

FTwoBandSHVector DiffuseTransferSH = CalcDiffuseTransferSH(MaterialParameters.WorldNormal,1);

// 计算漫射光时需要使用法线

half3 DiffuseGI = max(half3(0,0,0), DotSH(PointIndirectLighting, DiffuseTransferSH));

IndirectIrradiance = Luminance(DiffuseGI);

Color += DiffuseColor \* DiffuseGI;

否则

// 非方向光作用于半透明物体

// Ambient项被包括在了xyz中

// 已经在CPU中提前除以了PI与SH ambient

half3 PointIndirectLighting = IndirectLightingSHCoefficients[0];

half3 DiffuseGI = PointIndirectLighting;

IndirectIrradiance = Luminance(DiffuseGI);

Color += DiffuseColor \* DiffuseGI;

##### 计算阴影

half Shadow;

如果着色模型不是Unlit（无光照模式下不计算阴影）

Shadow = GetPrimaryPrecomputedShadowMask(Interpolants).r;

如果使用简单动态光照

Shadow = DirectionalLightShadowing;

如果使用可移动的直接光照级联阴影贴图

FPCFSamplerSettings Settings;

Settings.ShadowDepthTexture = View.DirectionalLightShadowTexture;

Settings.ShadowDepthTextureSampler = View.DirectionalLightShadowSampler;

Settings.TransitionScale = View.DirectionalLightShadowTransition;

Settings.ShadowBufferSize = View.DirectionalLightShadowSize;

Settings.bSubsurface = false;

Settings.DensityMulConstant = 0;

Settings.ProjectionDepthBiasParameters = 0;

float4 ShadowPosition;

for (int i = 0; i < MAX\_FORWARD\_SHADOWCASCADES; i++)

{

if (i == (MAX\_FORWARD\_SHADOWCASCADES - 1) // 忽略对最后一张阴影贴图的测试

|| MaterialParameters.ScreenPosition.w < View.DirectionalLightShadowDistances[i])

{

ShadowPosition = mul( float4(MaterialParameters.ScreenPosition.xyw, 1), View.DirectionalLightScreenToShadow[i]);

break; // 阴影位置（ShadowPosition）找到时退出循环

}

}

// 为了透明投影，需要在光照空间将像素深度值裁剪到[0.0f,0.99999f]范围内，

// 因为没有被渲染的阴影深度缓存区域会被置为1.0f。

// 为了得到不被投影的结果，我们需要无视当前渲染像素与平面的前后位置关系，强制进行阴影比较。

float LightSpacePixelDepthForOpaque = min(ShadowPosition.z, 0.99999f);

Settings.SceneDepth = LightSpacePixelDepthForOpaque;

half ShadowMap = Manual2x2PCF(ShadowPosition.xy, Settings);

Shadow = ShadowMap;

##### 计算方向光作用于像素点的颜色

如果着色模型不是Unlit（无光照模式下不计算方向光作用）

half NoL = max(0, dot(MaterialParameters.WorldNormal, View.DirectionalLightDirection));

half RoL = max(0, dot(MaterialParameters.ReflectionVector, View.DirectionalLightDirection));

Color += (Shadow \* NoL) \* View.DirectionalLightColor.rgb

\* (DiffuseColor + SpecularColor \* PhongApprox(Roughness, RoL));

如果材质非完全粗糙

half3 SpecularIBL = GetImageBasedReflectionLighting(MaterialParameters, Roughness);

// 环境贴图已经被预标准化，并经过光照贴图的亮度缩放

Color += SpecularIBL \* IndirectIrradiance \* SpecularColor;

如果启用天空光照

Color += GetSkySHDiffuseSimple(MaterialParameters.WorldNormal) \* View.SkyLightColor.rgb

\* DiffuseColor;

##### 计算自发光

Color += Emissive;

##### 计算顶点雾化

half4 VertexFog = half4(0, 0, 0, 1);

如果启用顶点雾化

如果使用封装的插值

VertexFog = PackedInterpolants[0];

否则

VertexFog = BasePassInterpolants.VertexFog;

##### 应用不着色蒙版

如果着色模型不是Unlit且ES2\_EMULATION（？暂时不知道这个量代表什么）

Color = lerp(Color, DiffuseColor + SpecularColor, View.UnlitViewmodeMask);

##### 处理材质混合模式，确定输出像素颜色

out half4 OutColor; // 像素着色器返回值

如果材质混合模式为半透明

OutColor = half4(Color \* VertexFog.a + VertexFog.rgb, Opacity);

否则，如果材质混合模式为叠加

OutColor = half4(Color \* (VertexFog.a \* Opacity.x), 0.0f);

否则，如果材质混合模式为乘积

half3 FoggedColor = lerp(half3(1, 1, 1), Color, VertexFog.aaa \* VertexFog.aaa);

OutColor = half4(FoggedColor, Opacity);

否则

OutColor.rgb = Color \* VertexFog.a + VertexFog.rgb;

如果启用HDR镶嵌或在Gamma空间输出（像素点颜色值包含Gamma校正）

OutColor.a = 0.0; // 此时场景颜色的alpha不会被使用，所以设为0

否则

OutColor.a = BasePassInterpolants.PixelPosition.w; // 将深度值Z放在16位的alpha值中

如果材质混合模式不是乘积

OutColor.rgba \*= View.ExposureScale.xyzw; // 曝光调整

如果在Gamma空间输出

OutColor.rgb = sqrt( OutColor.rgb );

#### CalcMaterialParameters函数

利用插值与纹理采样为当前像素点计算材质参数

void CalcMaterialParameters(in out FMaterialPixelParameters Parameters, float4 SvPosition,

FIsFrontFace bIsFrontFace, float4 PixelPosition, float4 PixelPositionExcludingShaderOffsets)

{

// 移除视图变换

Parameters.WorldPosition\_CamRelative = PixelPosition.xyz;

Parameters.WorldPosition = PixelPosition.xyz - View.PreViewTranslation.xyz;

// 如果材质使用了无偏移的世界位置表达式，就计算参数；否则，变量已经在这之前被初始化为0了

#if USE\_WORLD\_POSITION\_EXCLUDING\_SHADER\_OFFSETS

Parameters.WorldPosition\_NoOffsets\_CamRelative = PixelPositionExcludingShaderOffsets.xyz;

Parameters.WorldPosition\_NoOffsets = PixelPositionExcludingShaderOffsets.xyz

- View.PreViewTranslation.xyz;

#endif

Parameters.SVPosition = SvPosition;

{

float2 PixelPos = SvPosition.xy - View.ViewRectMin.xy;

Parameters.ScreenPosition.xy = ( PixelPos / View.ViewSizeAndSceneTexelSize.xy - 0.5 )

\* float2(2,-2);

Parameters.ScreenPosition.zw = SvPosition.zw;

Parameters.ScreenPosition.xyz \*= Parameters.ScreenPosition.w;

}

// 像素位置是世界坐标中的位置转化到相对于相机的位置，它的值等价于-CameraVector

Parameters.CameraVector = normalize(-PixelPosition.xyz);

Parameters.LightVector = 0;

Parameters.TwoSidedSign = 1.0f;

#if MATERIAL\_TWOSIDED && HAS\_PRIMITIVE\_UNIFORM\_BUFFER

Parameters.TwoSidedSign \*= View.CullingSign \* Primitive.LocalToWorldDeterminantSign;

#if !MATERIAL\_TWOSIDED\_SEPARATE\_PASS

Parameters.TwoSidedSign \*= GetFloatFacingSign(bIsFrontFace);

#endif

#endif

// 注意这里的Parameters.TangentNormal可能是在世界空间或切线空间中定义的。

Parameters.TangentNormal = GetMaterialNormal(Parameters);

#if MATERIAL\_TANGENTSPACENORMAL

// 为双面材质的背面反转法线

Parameters.TangentNormal \*= Parameters.TwoSidedSign;

#if FEATURE\_LEVEL >= FEATURE\_LEVEL\_SM4

// ES2 will rely on only the final normalize for performance

Parameters.TangentNormal = normalize(Parameters.TangentNormal);

#endif

// 切线空间转换到世界空间后执行标准化能够提升剪切基（UV层到世界空间会导致剪切）的质量

// 这里的TangentToWorld是矩阵TBN（顶点的切线、副法线、法线组成的矩阵）

// TransformTangentVectorToWorld 函数等价于mul(InTangentVector, TangentToWorld)

Parameters.WorldNormal = normalize(TransformTangentVectorToWorld(Parameters.TangentToWorld,

Parameters.TangentNormal));

#else

// 这里我们不需要支持双面材质

Parameters.TangentNormal = Parameters.WorldNormal = normalize(Parameters.TangentNormal);

#endif

Parameters.ReflectionVector = ReflectionAboutCustomWorldNormal(Parameters, Parameters.WorldNormal,

false);

#if !PARTICLE\_SPRITE\_FACTORY

Parameters.Particle.MotionBlurFade = 1.0f;

#endif // !PARTICLE\_SPRITE\_FACTORY

}

#### EnvBRDFApproxNonmetal函数

环境光作用于非金属表面双向反射方程的近似

half EnvBRDFApproxNonmetal( half Roughness, half NoV )

{

// 本函数等价于EnvBRDFApprox( 0.04, Roughness, NoV )

const half2 c0 = { -1, -0.0275 };

const half2 c1 = { 1, 0.0425 };

half2 r = Roughness \* c0 + c1;

return min( r.x \* r.x, exp2( -9.28 \* NoV ) ) \* r.x + r.y;

}

#### EnvBRDFApprox函数

环境光双向反射方程的近似

half3 EnvBRDFApprox( half3 SpecularColor, half Roughness, half NoV )

{

// BRDF方程中G项的一种替代

const half4 c0 = { -1, -0.0275, -0.572, 0.022 };

const half4 c1 = { 1, 0.0425, 1.04, -0.04 };

half4 r = Roughness \* c0 + c1;

half a004 = min( r.x \* r.x, exp2( -9.28 \* NoV ) ) \* r.x + r.y;

half2 AB = half2( -1.04, 1.04 ) \* a004 + r.zw;

// 任何小于2%的光都是不可见的，因此，它们被认为是阴影。

AB.y \*= saturate( 50.0 \* SpecularColor.g );

return SpecularColor \* AB.x + AB.y;

}

#### CalcDiffuseTransferSH函数

通过计算漫反射传输中的系数，并与球谐函数做卷积，函数中的，有关球谐函数的资料参考“动态全局光照的相关算法整理.docx”

FTwoBandSHVector CalcDiffuseTransferSH(half3 Normal,half Exponent)

{

FTwoBandSHVector Result = SHBasisFunction(Normal);

// 这些式子是球谐函数每阶对应的缩放因子，球谐函数的每一项与这些因子做卷积

// max(0,cos(theta))^Exponent

half L0 = 2 \* PI / (1 + 1 \* Exponent);

half L1 = 2 \* PI / (2 + 1 \* Exponent);

// 每个波形的系数乘以合适的波形缩放因子

Result.V.x \*= L0;

Result.V.yzw \*= L1;

return Result;

}

#### SHBasisFunction函数

求解一阶球谐函数，有关球谐函数的资料参考“动态全局光照的相关算法整理.docx”。

FTwoBandSHVector SHBasisFunction(half3 InputVector)

{

FTwoBandSHVector Result;

Result.V.x = 0.282095f; //

Result.V.y = -0.488603f \* InputVector.y; //

Result.V.z = 0.488603f \* InputVector.z; //

Result.V.w = -0.488603f \* InputVector.x; //

return Result;

}

#### DotSH函数

计算两个投影在标量函数上的球谐函数的点积，得到的值是这两个球谐函数的积在所有方向上的积分。

half DotSH(FTwoBandSHVector A,FTwoBandSHVector B)

{

half Result = dot(A.V, B.V);

return Result;

}

half3 DotSH(FTwoBandSHVectorRGB A,FTwoBandSHVector B)

{

half3 Result = 0;

Result.r = DotSH(A.R,B);

Result.g = DotSH(A.G,B);

Result.b = DotSH(A.B,B);

return Result;

}

#### Luminance函数

计算光亮度，MaterialFloat的类型可能是half或者float

MaterialFloat Luminance( MaterialFloat3 LinearColor )

{

return dot( LinearColor, MaterialFloat3( 0.3, 0.59, 0.11 ) );

}

#### GetPrimaryPrecomputedShadowMask函数

half GetPrimaryPrecomputedShadowMask(FVertexFactoryInterpolantsVSToPS Interpolants)

{

#if STATICLIGHTING\_TEXTUREMASK && STATICLIGHTING\_SIGNEDDISTANCEFIELD

// 获取距离场中的数据

half DistanceField = Texture2DSample(StaticShadowTexture, StaticShadowTextureSampler,

GetShadowMapCoordinate(Interpolants)).r;

// 通过缩放和偏置距离计算阴影因子

half ShadowFactor=saturate( DistanceField\*DistanceFieldParameters.x+DistanceFieldParameters.y);

return ShadowFactor \* ShadowFactor;

#elif MOVABLE\_DIRECTIONAL\_LIGHT

// 需要在检查光照贴图之前执行这一步，因为我们也许会同时使用光照贴图和方向光

return 1;

#elif HQ\_TEXTURE\_LIGHTMAP || LQ\_TEXTURE\_LIGHTMAP

// 将有光照贴图但没有阴影贴图的对象标记为阴影

// 被光源影响，且被认为完全被阴影遮挡的对象，在STATICLIGHTING\_TEXTUREMASK==1时不会被渲染

return 0;

#else

// 默认情况下标记为无阴影

return 1;

#endif

}

#### ReflectionAboutCustomWorldNormal函数

根据世界空间法线计算反射光线。

MaterialFloat3 ReflectionAboutCustomWorldNormal(FMaterialPixelParameters Parameters,

MaterialFloat3 WorldNormal, bool bNormalizeInputNormal)

{

if (bNormalizeInputNormal)

{

WorldNormal = normalize(WorldNormal);

}

return -Parameters.CameraVector + WorldNormal \* dot(WorldNormal, Parameters.CameraVector) \* 2.0;

}

#### PhongApprox函数

Phong光照模型的近似

half PhongApprox( half Roughness, half RoL )

{

half a = Roughness \* Roughness; // 1个MUL（乘法）指令

half a2 = a \* a; // 1个MUL指令

float rcp\_a2 = rcp(a2); // 1个RCP（求倒数）指令

//half rcp\_a2 = exp2( -6.88886882 \* Roughness + 6.88886882 );

// 球面高斯方程的近似: pow( x, n ) ~= exp( (n + 0.775) \* (x - 1) )

// Phong光照模型: n = 0.5 / a2 - 0.5

// 0.5 / ln(2), 0.275 / ln(2)

half c = 0.72134752 \* rcp\_a2 + 0.39674113; // 1个MAD指令

return rcp\_a2 \* exp2( c \* RoL - c ); // 2个MAD指令，1个EXP2（求平方）指令，1个MUL指令

// 本函数总共使用7个GPU指令

}