车漆技术分析

- ◯ 1.基础层
  - ▢ Diffuse Color
    - Lit
      - bsdfData.diffuseColor = ComputeDiffuseColor(surfaceData.baseColor, metallic);
        - Lit.hlsl的411行
    - StackLit
      - bsdfData.diffuseColor = ComputeDiffuseColor(surfaceData.baseColor, metallic);
        - StackLit.hlsl的795行
    - UE
  - ▢ Metallic
    - Lit
      - float metallic = HasFlag(surfaceData.materialFeatures, MATERIALFEATUREFLAGS_LIT_SPECULAR_COLOR | MATERIALFEATUREFLAGS_LIT_SUBSURFACE_SCATTERING | MATERIALFEATUREFLAGS_LIT_TRANSMISSION) ? 0.0 : surfaceData.metallic;
        - Lit.hlsl 第 409 行
    - StackLit
      - float metallic = HasFlag(surfaceData.materialFeatures, MATERIALFEATUREFLAGS_STACK_LIT_SPECULAR_COLOR | MATERIALFEATUREFLAGS_STACK_LIT_SUBSURFACE_SCATTERING | MATERIALFEATUREFLAGS_STACK_LIT_TRANSMISSION) ? 0.0 : surfaceData.metallic;
        - StackLit — 第 793 行
          - 两者完全相同，只有feature flag 的命名前缀区别
    - UE
  - ▢ Smothness
    - Lit

- bsdfData.perceptualRoughness = PerceptualSmoothnessToPerceptualRoughness(surfaceData.perceptualSmoothness);
  - Lit.hlsl的406行
- StackLit
  - bsdfData.perceptualRoughnessA = PerceptualSmoothnessToPerceptualRoughness(surfaceData.perceptualSmoothnessA);
    bsdfData.perceptualRoughnessB = PerceptualSmoothnessToPerceptualRoughness(surfaceData.perceptualSmoothnessB);
    - StackLit.hlsl的789-790行
      - A/B 两叶各自独立，再通过 lobeMix（第 828 行）混合。Coat 模式下 roughness 不立即 clamp（第 880 ~ 883 行 vs 第 874 行），延迟到 VLayering 处理。
- UE

- ▢ 间接光GI部分
  - Lit
    - builtinData.bakeDiffuseLighting *= preLightData.diffuseFGD * GetDiffuseOrDefaultColor(bsdfData, _ReplaceDiffuseForIndirect).rgb;
      - Lit — 第 1336 行（无 diffuseEnergy 项）
  - StackLit
    - builtinData.bakeDiffuseLighting *= preLightData.diffuseFGD * preLightData.diffuseEnergy * GetDiffuseOrDefaultColor(bsdfData, _ReplaceDiffuseForIndirect).rgb;
      - StackLit — 第 3040 行（多乘 diffuseEnergy）
        - VLayering 计算出的 coat 层对 base 层 diffuse 的透射衰减系数，Coat 越厚/IOR 越高，该值越趋近于 0。
  - UE

- ▢ Fresnel0（镜面反射基础色）
  - F0公式，

$$F_0 = \left(\frac{n_t - n_i}{n_t + n_i}\right)^2$$

- nt:折射介质的 IOR（transmittedIor，即材质本身
- ni:入射介质的 IOR（incidentIor，默认 1.0，即空气）

- Lit
  - bsdfData.fresnel0 = HasFlag(surfaceData.materialFeatures, MATERIALFEATUREFLAGS_LIT_SPECULAR_COLOR) ? surfaceData.specularColor : ComputeFresnel0(surfaceData.baseColor, surfaceData.metallic, DEFAULT_SPECULAR_VALUE);
    - Lit.hlsl 第 412 行
      - 介电质硬编码为 DEFAULT_SPECULAR_VALUE（0.04），无法配置 IOR。当F0为0.04时，IOR值为1.5

材质	IOR	F0
空气	1.0	0.0
水	1.33	0.02
玻璃 / Clear Coat	1.5	**0.04**（Lit 的 DEFAULT_SPECULAR_VALUE）
钻石	2.4	0.17
金属（铁）	~2.9	~0.23

- StackLit
  - if (!surfaceData.useProfileIor)
    {
    if (HasFlag(surfaceData.materialFeatures, MATERIALFEATUREFLAGS_STACK_LIT_SPECULAR_COLOR))
    bsdfData.fresnel0 = surfaceData.specularColor;
    else
    bsdfData.fresnel0 = ComputeFresnel0(surfaceData.baseColor, metallic, IorToFresnel0(surfaceData.dielectricIor));
    }
    - CarPaintStackLit.hlsl 第 819 ~ 822 行
      - 介电质通过 dielectricIor 物理换算，可灵活配置。
  - bsdfData.fresnel0 = ConvertF0ForAirInterfaceToF0ForNewTopIor(bsdfData.fresnel0, bsdfData.coatIor);
    - float3 ConvertF0ForAirInterfaceToF0ForNewTopIor(float3 fresnel0, float newTopIor)
      {
      float3 ior = Fresnel0ToIor(min(fresnel0, float3(1,1,1)*0.999)); // guard against 1.0
      return IorToFresnel0(ior, newTopIor);
      }
      - Coat 层对 fresnel0 的二次换算 — 第 870 行

- UE

2. Clear Coat

Coat模型

Lit

Lit 近似模型

```
if (HasFlag(bsdfData.materialFeatures, MATERIALFEATUREFLAGS_LIT_CLEAR_COAT))
{
    // Apply isotropic GGX for clear coat
    // Note: coat F is scalar as it is a dieletric
    float coatF = F_Schlick(CLEAR_COAT_F0, LdotH) * bsdfData.coatMask;
    // Scale base specular
    specTerm *= Sq(1.0 - coatF);

    // Add top specular
    // TODO: Should we call just D_GGX here ?
    // We use abs(NdotL) to handle the none case of double sided
    float DV = DV_SmithJointGGX(NdotH, abs(NdotL), clampedNdotV, bsdfData.coatRoughness, preLightData.coatPartLambdaV);
    specTerm += coatF * DV;

    // Note: The modification of the base roughness and fresnel0 by the clear coat is already handled in FillMaterialClearCoatData

    // Very coarse attempt at doing energy conservation for the diffuse layer based on NdotL. No science.
    diffTerm *= lerp(1, 1.0 - coatF, bsdfData.coatMask);
}
```

公式本质

F： coatF = coat 层菲涅耳反射率，由F_Schlick(CLEAR_COAT_F0, LdotH)计算，取值为固定值0.04

（1-F)^2：光线穿过 coat 层两次（下行+上行）的透射衰减，这是 近似：实际应是 T_down * T_up，但用(1-F) ^2代替

base：specTerm（base层的完整高光BRDF结果），base 层的 GGX 高光（已含 diffuseColor/fresnel0 等），取值由下层 BRDF 计算

coat：DV，coat 层自身的 GGX 分布函数值，取值：DV_SmithJointGGX，用固定 roughness

$$\text{result} = (1 - F)^2 \cdot \text{base} + F \cdot \text{coat}$$

- 本质问题：(1-F)² 忽略了层间多次弹射，也忽略了折射对 base 层角度和 roughness 的影响。

StackLit

CarPaintStackLit Adding 方程（无限次内反射求和）

```
// Multiple scattering forms
float3 denom = max(0.0, (float3(1.0, 1.0, 1.0) - R10*R12)); //i = new layer, 0 = cumulative top (llab3.1 to 3.4)

// (mean(denom) <= 0.0f)? is not enough to prevent division by 0, see below
//
// float3 m_RGi = (mean(denom) <= 0.0f)? float3(0.0, 0.0, 0.0) : (T0i*R12*T10) / denom; //(llab3.1)
// float3 m_Ri0 = (mean(denom) <= 0.0f)? float3(0.0, 0.0, 0.0) : (T21*R10*T12) / denom; //(llab3.2)
// float3 m_Rr  = (mean(denom) <= 0.0f)? float3(0.0, 0.0, 0.0) : (R10*R12) / denom;
float3 m_RGi = float3(0.0, 0.0, 0.0);
float3 m_Ri0 = float3(0.0, 0.0, 0.0);
float3 m_Rr  = float3(0.0, 0.0, 0.0);

// Energy for transmission terms:
float3 e_T0i = float3(0.0, 0.0, 0.0);
float3 e_Ti0 = float3(0.0, 0.0, 0.0);

// If the denominator of any component of the multiple scattering forms reaches 0, the series becomes meaningless
// as bounces between the interfaces become "trapped" there in that limit, so we leave the terms to 0.
UNITY_UNROLL
for(int j = 0; j < 3; ++j)
{
    if (denom[j] > 0.0)
    {
        m_RGi[j] = (T0i[j]*R12[j]*Ti0[j]) / denom[j]; //(llab3.1)
        m_Ri0[j] = (T21[j]*Ri0[j]*T12[j]) / denom[j]; //(llab3.2)
        m_Rr[j]  = (Ri0[j]*R12[j]) / denom[j];
        // Evaluate the adding operator on the energy for transmission terms:
        e_T0i[j] = (T0i[j]*T12[j]) / denom[j]; //(llab3.3)
        e_Ti0[j] = (T21[j]*Ti0[j]) / denom[j]; //(llab3.4)
    }
}
float  m_rGi = mean(m_RGi);
float  m_ri0 = mean(m_Ri0);
m_rr  = mean(m_Rr);

// Evaluate the adding operator on the energy for reflection terms:
float3 e_RGi = R0i + n_RGi; //(llab3.1)
float3 e_Ri0 = R21 + n_Ri0; //(llab3.2)
```

- 公式本质：
  光在两层界面之间无限次弹射的物理精确解（几何级数的闭合式）。

$$R_{total} = R_{0i} + \frac{T_{0i} \cdot R_{12} \cdot T_{i0}}{1 - R_{i0} \cdot R_{12}}$$

  - 具体参数的物理意义

参数	物理含义
R12	当前新界面（第 i 层）对向下光线的**反射率**
T12	当前新界面对向下光线的**透射率** = 1 - R12
R21	当前新界面对向上光线的**反射率**（由下往上看）
T21	当前新界面对向上光线的**透射率**
R0i	当前**累积栈**（所有已处理层）对向下光线的反射率
T0i	当前累积栈对向下光线的透射率
Ri0	当前累积栈对向上光线的反射率
Ti0	当前累积栈对向上光线的透射率
denom = 1 - Ri0·R12	分母，代表**层间多次弹射的能量截止因子**
m_R0i = T0i·R12·Ti0 / denom	通过新界面弹射后的**多次散射累积贡献**
e_R0i = R0i + m_R0i	加入新界面后，整个栈的**总反射率**

    - R12:coat 层用 FresnelUnpolarized，base 层用 F_Schlick(fresnel0)
    - T12:同上，coat 层透射；media 层用 Beer-Lambert exp(-thickness*extinction/cosθ)
    - R21:菲涅耳对称，R21 = R12
    - T21 = T12（对称
    - R0i:迭代累积，初始 = 0
    - T0i:迭代累积，初始 = 1
    - Ri0:迭代累积，初始 = 0
    - Ti0:迭代累积，初始 = 1
    - denom = 1 - Ri0·R12:越接近 0 说明两层之间越容易"困住"光
    - m_R0i = T0i·R12·Ti0 / denom:这是等比级数 ∑(Ri0·R12)^n 的闭合解
    - e_R0i = R0i + m_R0i:最终输出给下一层迭代
  - Ri0 * R12的物理意义
    每一项代表光在上层累积栈（反射率 Ri0）和新界面（反射率 R12）之间来回弹射一次的能量贡献。

分母 `1 - Ri0·R12` 的物理意义

这是整个公式最关键的部分，展开后等于无穷级数：

$$\frac{1}{1 - R_{i0} \cdot R_{12}} = 1 + (R_{i0} \cdot R_{12}) + (R_{i0} \cdot R_{12})^2 + \cdots$$

    - Lit 的公式里没有这个项，相当于强制令分母 = 1（即忽略所有内部弹射），所以 Lit 的模型在高 IOR、厚 coat 的情况下能量不守恒。
- 具体函数：void ComputeAdding(float _cti, float3 V, in BSDFData bsdfData, inout PreLightData preLightData,
  bool calledPerLight = false, bool testSingularity = false, float minRoughness = 0.0)

- 方差/Roughness 传播（Adding 独有）

```
// Evaluate the adding operator on the normalized variance
_s_r0m = s_ti0 + j0i*(s_t0i + s_r12 + m_rr*(s_r12+s_ri0));
float _s_r0i = (r0i*s_r0i + m_r0i*_s_r0m); // e_r0i; -> _s_r0i normalization is done below
float _s_t0i = j12*s_t0i + s_t12 + j12*(s_r12 + s_ri0)*m_rr;
float _s_rim = s_t12 + j12*(s_t21 + s_ri0 + m_rr*(s_r12+s_ri0));
float _s_ri0 = (r21*s_r21 + m_ri0*_s_rim); // e_ri0; -> _s_ri0 normalization is done below
float _s_ti0 = ji0*s_t21 + s_ti0 + ji0*(s_r12 + s_ri0)*m_rr;
_s_r0i = (e_r0i > 0.0) ? _s_r0i/e_r0i : 0.0;
_s_ri0 = (e_ri0 > 0.0) ? _s_ri0/e_ri0 : 0.0;
```

  - 参数及其意义

参数	含义
s_r12 / s_t12	当前界面反射/透射引入的**线性化方差**（roughness 的方差表示）
j12	Jacobian 项，折射时因角度压缩/扩展导致的**立体角变化比** = cosθ_t / cosθ_i * n12
m_rr	多次弹射的方差加权因子
_s_r0i	累积到当前层后，整个栈反射 lobe 的**总方差**

    - 这些项确保了 base 层的有效 roughness 不是原始输入值，而是被 coat 层折射和散射后的物理正确值。Lit 对此的处理（FillMaterialClearCoatData 第 254 ~ 256 行）只是一个一次性的标量近似，完全不考虑折射角度和 Jacobian。

- ▢ Coat参数
  - Lit
    - bsdfData.coatMask = coatMask;
      float ieta = lerp(1.0, CLEAR_COAT_IETA, bsdfData.coatMask);
      bsdfData.coatRoughness = CLEAR_COAT_ROUGHNESS;
      - 所有 Coat 属性全部是硬编码常量，在 GetPreLightData 中直接使用
        #define CLEAR_COAT_ROUGHNESS 0.01
        #define CLEAR_COAT_IETA (1.0 / CLEAR_COAT_IOR)
    - toIor = lerp(1.0, CLEAR_COAT_IOR, bsdfData.coatMask);
      // ...
      preLightData.coatPartLambdaV = GetSmithJointGGXPartLambdaV(clampedNdotV, CLEAR_COAT_ROUGHNESS);
      preLightData.coatIblR = reflect(-V, N);
      preLightData.coatIblF = F_Schlick(CLEAR_COAT_F0, clampedNdotV) * bsdfData.coatMask;
      - #define CLEAR_COAT_IOR 1.5
        #define CLEAR_COAT_F0 0.04
  - StackLit
    - 5个参数自定义

```
void FillMaterialCoatData(float coatPerceptualRoughness, float coatMask, float coatIor, float coatThickness, float3 coatExtinction, inout BSDFData bsdfData)
{
    bsdfData.coatPerceptualRoughness = coatPerceptualRoughness;
    // Hack: A true coatMask would lead to complications like additionnal base lobes unmodified by the coat,
    // in proportion to 1-coatMask. We lerp the ior with 1 here along with thickness to 0 so that Beer-Lambert attenuation
    // is removed as coatMask -> 0. Some series term in ComputeStatistics are also lerped with their proper neutral values, along
    // with pre-integrated FGD terms (to lerp them to 0).
    bsdfData.coatMask       = coatMask;
    bsdfData.coatIor        = lerp(1.0, coatIor, bsdfData.coatMask);
    bsdfData.coatThickness  = coatThickness * bsdfData.coatMask;
    bsdfData.coatExtinction = coatExtinction;
}
```

- 用户可自定义参数：
  1.coatMask
  2.coatIor
  3.coatPerceptualRoughness
  4.coatThinckness
  5.coatExtinction

- Clear Coat对各个属性的影响
  - Coat 的计算分三层，在 ComputeStatistics 中处理，i=0 是界面层，i=1 是媒介层，i=2 是 base 层
  - fresnel0
    - bsdfData.fresnel0 = ConvertF0ForAirInterfaceToF0ForNewTopIor(bsdfData.fresnel0, bsdfData.coatIor);

```
bsdfData.fresnel0 = ConvertF0ForAirInterfaceToF0ForNewTopIor(bsdfData.fresnel0, bsdfData.coatIor);
```

    - float3 ConvertF0ForAirInterfaceToF0ForNewTopIor(float3 fresnel0, float newTopIor)
      {
      float3 ior = Fresnel0ToIor(min(fresnel0, float3(1,1,1)*0.999)); // guard against 1.0
      return IorToFresnel0(ior, newTopIor);
      }

```
float3 ConvertF0ForAirInterfaceToF0ForNewTopIor(float3 fresnel0, float newTopIor)
{
    float3 ior = Fresnel0ToIor(min(fresnel0, float3(1,1,1)*0.999)); // guard against 1.0
    return IorToFresnel0(ior, newTopIor);
}
```

      - TEMPLATE_1_REAL(Fresnel0ToIor, fresnel0, return ((1.0 + sqrt(fresnel0)) / (1.0 - sqrt(fresnel0)))) //转换F0到Ior
      - 输入：材质本身的F0，按照空气/材质界面计算的菲涅尔基础反射率
      - newTopIor = bsdfData.coatIor：
      - 输出：重新换算后的 F0，表示在 coat 介质下 base 层的实际反射率
  - Roughness
    - coat界面层（i=0）的方差计算
      s_r12 = RoughnessToLinearVariance(ClampRoughnessIfHonorLightMinRoughness(bsdfData.coatRoughness, minRoughness)) * bsdfData.coatMask;

```
s_r12 = RoughnessToLinearVariance(ClampRoughnessIfHonorLightMinRoughness(roughness bsdfData.coatRoughness, minRoughness)) * bsdfData.coatMask;
s_t12 = RoughnessToLinearVariance(ClampRoughnessIfHonorLightMinRoughness(roughness bsdfData.coatRoughness, minRoughness) * 0.5 * abs((ctt*n12 - ct1)/(ctt*n12)));
j12   = (ctt/ct1)*n12;

s_r21 = s_r12;
s_t21 = RoughnessToLinearVariance(ClampRoughnessIfHonorLightMinRoughness(roughness bsdfData.coatRoughness, minRoughness) * 0.5 * abs((ct1/n12 - ctt)/(ct1/n12)));
j21   = 1.0/j12;
```

- 参数解释

参数	含义
s_r12	coat 界面**反射**引入的线性化方差，= coat roughness 转换后 × coatMask
s_t12	coat 界面**透射**引入的线性化方差，因折射角变化而缩放
j12 = (ctt/cti)*n12	Jacobian，折射前后立体角的压缩/扩展比
cti	入射角余弦（光从上往下，coatNormal·V）
ctt	折射角余弦（通过 Snell 定律从 cti 计算）
n12 = coatIor	coat 的折射率比（n_coat / n_air）

  - 公式展开

$$\sigma^2_{total} = \underbrace{s\_ti0}_{\text{透射方差累积}} + j_{0i} \times \left(\underbrace{s\_t0i}_{\text{上层透射}} + \underbrace{s\_r12}_{\text{base 原始方差}} + \underbrace{m\_rr \times (s\_r12 + s\_ri0)}_{\text{多次弹射方差贡献}} \right)$$

- 折射角计算

```
float sti = sqrt(1.0 - Sq(cti));           // sin(θ_i)
float stt = sti / n12;                      // sin(θ_t) = sin(θ_i)/n  (Snell)
// roughness 越大，折射效果越弱（hack）：
const float scale = clamp((1.0-alpha)*(sqrt(1.0-alpha) + alpha), 0.0, 1.0);
stt = scale*stt + (1.0-scale)*sti;          // roughness 越大 → stt → sti，折射消失
ctt = sqrt(1.0 - stt*stt);                  // cos(θ_t)
```

  - 折射角计算的目的是：确定光进入 coat 层后的真实传播方向，用这个方向去计算 base 层的 roughness 增量（方差）和立体角变化（Jacobian），从而让 base 层的高光形状和亮度反映 coat 层的物理影响。

- Base 层 最终于Roughness的计算

```
s_r12 = RoughnessToLinearVariance(※ClampRoughnessIfHonorLightMinRoughness( roughness: PerceptualRoughnessToRoughness(bsdfData.perceptualRoughnessA), minRoughness));
_s_r0m = s_ti0 + j0i*(s_t0i + s_r12 + m_rr*(s_r12+s_ri0));
preLightData.iblPerceptualRoughness[BASE_LOBEA_IDX] = LinearVarianceToPerceptualRoughness(_s_r0m);
s_r12 = RoughnessToLinearVariance(※ClampRoughnessIfHonorLightMinRoughness( roughness: PerceptualRoughnessToRoughness(bsdfData.perceptualRoughness8), minRoughness));
_s_r0m = s_ti0 + j0i*(s_t0i + s_r12 + m_rr*(s_r12+s_ri0));
preLightData.iblPerceptualRoughness[BASE_LOBEB_IDX] = LinearVarianceToPerceptualRoughness(_s_r0m);
```

  - 参数解释

参数	含义
s_ti0	从上往下穿过所有层到 base 层，透射方向累积的方差
j0i	从顶层到 base 层的累积 Jacobian（立体角缩放）
s_t0i	从顶层到 base 层的透射方差累积
m_rr	多次弹射加权因子
_s_r0m	base 层反射 lobe 的**总方差** = 原始方差 + coat 各层贡献之和
最终 iblPerceptualRoughness	经过 coat 层处理后，base 层对 IBL 实际使用的 perceptualRoughness

- Beer-Lambert

- 媒介层（i=1）

  R12 = float3(0.0, 0.0, 0.0);
  T12 = exp(- bsdfData.coatThickness * bsdfData.coatExtinction / cti);
  R21 = R12;
  T21 = T12;

  公式本质
  T=e ^(−thickness×extinction/cosθ)

  - coatThickness:coat 物理厚度（已乘 coatMask，mask=0 时 thickness=0，T=1，无衰减）
  - coatExtinction:RGB 三通道消光系数，控制吸收的颜色（如偏红色的 coat 会让 extinction.g/b 更大）
  - cti:当前折射角余弦，角度越大（掠射）路径越长，衰减越强
  - T12（输出）:单次穿越媒介层的透射率，会通过 Adding 方程累积进 Ti0

- diffuseEnergy（间接光暗部）
  - 在ComputeAdding末尾

```
#ifdef VLAYERED_DIFFUSE_ENERGY_HACKED_TERM
    // TODO
    // coatMask hack: See FillMaterialCoatData.
    preLightData.diffuseEnergy = lerp(float3(1.0, 1.0, 1.0), Ti0, bsdfData.coatMask);
    // Not correct since these stats are still directional probably too much
    // removed, but with a non FGD term, could actually balance out (as using
    // FGD would lower this)
#else // #ifdef VLAYERED_DIFFUSE_ENERGY_HACKED_TERM
    preLightData.diffuseEnergy = float3(1.0, 1.0, 1.0);
#endif // #ifdef VLAYERED_DIFFUSE_ENERGY_HACKED_TERM #else
```

    - 参数解释

参数	含义
Ti0	经过三层 Adding 迭代后，从顶层到 base 层的**累积向上透射系数**（含 coat 界面 + Beer-Lambert）
coatMask	插值权重，mask=0 时 diffuseEnergy=1（无衰减），mask=1 时完全用 Ti0
diffuseEnergy（输出）	最终乘进 bakeDiffuseLighting，决定间接光能传多少到 base 层

  - 使用在builtinData.bakeDiffuseLighting

```
// pre-integration and energy term.
// preLightData.diffuseEnergy will be 1,1,1 if no vlayering or no VLAYERED_DIFFUSE_ENERGY_HACKED_TERM
// Note: When baking reflection probes, we approximate the diffuse with the fresnel0
builtinData.bakeDiffuseLighting += preLightData.diffuseFGD * preLightData.diffuseEnergy * GetDiffuseOrDefaultColor(bsdfData, _ReplaceDiffuseForIndirect).rgb;
```

- coat 反射系数（直接光高光）
  - coat 界面 Fresnel，ComputeStatistics i=0 第 1552 ~ 1553 行

    n12 = GetCoatEta(bsdfData); //n2/n1;
    R0 = FresnelUnpolarized(cti, n12, 1.0);

  - 非极化FresnelUnpolarized

- 背景：
  自然光（太阳光、灯光）是非偏振光，可以分解为两种正交偏振分量：
  ·p 偏振（parallel）：电场振动平行于入射平面
  ·s 偏振（senkrecht，德语"垂直"）：电场振动垂直于入射平面
  两种偏振的菲涅耳反射率不同，自然光的总反射率是两者平均。
- 代码：

```
float FresnelUnpolarized(in float ct1, in float n1, in float n2)
{
    float cti = ct1;
    float st2 = (1.0 - Sq(cti));
    float nr  = n2/n1;
    if(nr == 1.0) { return 0.0; }

    if(Sq(nr)*st2 <= 1.0) {
        float ctt = sqrt(1.0 - Sq(nr)*st2) ;
        float tpp = (nr*cti-ctt) / (nr*cti + ctt);
        float tps = (cti-nr*ctt) / (nr*ctt + cti);
        return 0.5 * (tpp*tpp + tps*tps);
    } else {
        return 0.0;
    }
}
```

- 公式本质：

$$r_p = \frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1 \cos\theta_t}, \quad r_s = \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_t + n_2 \cos\theta_i}$$

$$R_{unpolarized} = \frac{1}{2}(r_p^2 + r_s^2)$$

- 结论：Schlick 用 F0 的单一标量无法捕捉这种差异，FresnelUnpolarized 的 s/p 平均才能正确描述自然光在介电质 coat 层上的真实反射行为，这对物理正确的 VLayering 能量分配至关重要。

2.1. Coat计算后的合成

准备

阶段一：ComputeAdding 输出中间结果

- ComputeAdding 执行完毕后，所有 coat 效果都被"烘"进 preLightData 的以下字段
  - vLayerEnergyCoeff[TOP]
    - 含义：coat 层反射系数，用于 coat 高光
    - 来源：Adding 方程 m_R0i
  - vLayerEnergyCoeff[BOTTOM]
    - 含义：base 层经 coat 折射后的等效菲涅耳，用于 base 高光
    - 来源：Adding 方程累积的 e_R0i

- layeredRoughnessT/B[0/1]
  - 含义：coat 影响后 base 层的实际各向异性 roughness
  - 来源：方差传播方程 _s_r0m
- iblPerceptualRoughness[]
  - 含义：各叶（coat/A/B）用于 IBL 采样的 perceptualRoughness
  - 来源：　方差传播方程，LinearVarianceToPerceptualRoughness
- diffuseEnergy
  - 含义：coat 对 diffuse 的透射衰减系数
  - 来源：lerp(1, Ti0, coatMask)
    - 汇总公式：

$$Ti0 \approx \underbrace{(1 - R_{coat}(coatIor, \theta))}_{\text{coatIor, coatMask, NdotV}} \times \underbrace{e^{-thickness \cdot extinction / \cos\theta}}_{\text{coatThickness, coatExtinction, coatMask, NdotV}} \times \underbrace{\frac{1 - R_{base}(fresnel0)}{1 - R_{upward} \cdot R_{base}}}_{\text{fresnel0（metallic）}}$$

      - 降低diffuseEnergy的参数优先级
        - coatThickness + coatExtinction — Beer-Lambert 可以把 Ti0 压到接近 0
          - extinction = 0（三通道全0）：coat 完全透明无色，不吸收任何光，T=1，diffuseEnergy 不受媒介层影响。

典型车漆设置：

coat 颜色效果	extinction 设置	透射颜色
无色透明 coat	(0, 0, 0)	不变
偏黄色 coat（金色车漆）	(0, 0, 0.5) 蓝色吸收	偏黄
偏红色 coat（深红车漆）	(0, 0.3, 0.3) 绿蓝吸收	偏红
偏蓝色 coat（珠光蓝）	(0.3, 0.3, 0) 红绿吸收	偏蓝
深色 coat（整体压暗）	(1, 1, 1) 均匀吸收	整体变暗

          - 跟 thickness 的关系，两者共同决定最终吸收量，需要配合使用：
            T = exp(-thickness * extinction / cosθ)

情况	结果
extinction=(1,1,1), thickness=0.1	T ≈ 0.9，轻微压暗
extinction=(1,1,1), thickness=1.0	T ≈ 0.37，明显压暗
extinction=(1,1,1), thickness=3.0	T ≈ 0.05，极度压暗
extinction=(0,0,1), thickness=1.0	T=(1, 1, 0.37)，蓝色被压，整体偏黄

        - coatIor — IOR 越高，coat 界面反射越多，透射越少

- metallic（via fresnel0）— 金属 fresnel0 高，base 界面透射少
- coatMask — 同时控制上面三层的强度
- NdotV — 视角越掠射，每一层衰减都更严重

- energyCompensationFactor[COAT]
  - 含义：coat 叶的多次散射能量补偿
  - 来源：GetEnergyCompensationFactor(reflectivity, IorToFresnel0(coatIor))

- 直接光
  阶段二：直接光合成（BSDF 函数）
  - specular(直接光)合成：

$$\text{specR} = \underbrace{(N\cdot L)_{base}\cdot bottomF\cdot \text{lerp}(DV_A, DV_B, \text{lobeMix})}_{\text{base 高光（含 coat 压缩的 F0 和 roughness）}} + \underbrace{(N\cdot L)_{coat}\cdot R_{top}\cdot DV_{coat}}_{\text{coat 自身高光}}$$

    - 参数：bottomF
      贡献：coat 不存在时 = F_Schlick(fresnel0, LdotH)；coat 存在时 = lerp(原始F, vLayerEnergyCoeff[BOTTOM], coatMask)，即被 coat 换算过的 F
    - 参数：DV[BASE]
      贡献：用的是经方差传播后的 layeredRoughnessT/B，不是原始 roughness
    - 参数：vLayerEnergyCoeff[TOP]
      贡献：coat 层自身反射系数，来自 Adding 方程
    - 参数：DV[COAT]
      贡献：coat 层自己的 GGX，用 layeredCoatRoughness（可调，非固定常量）
    - DV = D × V，是 GGX BRDF 中两个项的组合
      - 细分主题 1

D — 法线分布函数（Normal Distribution Function）

描述微表面法线的统计分布，决定**高光的形状和大小**。

$$D(\alpha, N\cdot H) = \frac{\alpha^2}{\pi\left((N\cdot H)^2(\alpha^2-1)+1\right)^2}$$

roughness 越小 → D 越尖锐（镜面高光），roughness 越大 → D 越扁平（漫反射高光）。

      - 细分主题 2

V — 几何遮蔽/阴影项（Visibility / Shadowing-Masking）

描述微表面之间的自遮挡，roughness 越大微表面越容易互相遮挡。

HDRP 用的是 Smith Joint GGX 的组合形式：

$$V(N\cdot L, N\cdot V, \alpha) = \frac{0.5}{N\cdot L\cdot \Lambda(N\cdot V) + N\cdot V\cdot \Lambda(N\cdot L)}$$

- 细分主题 3

$$\text{specular} = \underbrace{F}_{\text{菲涅耳颜色}} \times \underbrace{DV}_{\text{形状+遮蔽}} \times (N \cdot L)$$

- diffuse(直接光)合成：

$$\text{diffR} = \underbrace{\frac{1}{\pi}}_{\text{Lambert}} \times \underbrace{\text{diffuseEnergy}}_{\text{coat 唯一贡献}} \times \underbrace{(N_{base} \cdot L)}_{\text{base 法线角度}}$$

  - coat对diffuse的唯一贡献就是diffuseEnergy的贡献

coat 参数	贡献方式	贡献量
coatMask	控制 diffuseEnergy 的插值权重，=0 时无影响，=1 时完全使用 Ti0	主开关
coatIor	coat 界面反射率 R_coat 越高，Ti0 中的 (1-R_coat) 越小	中等
coatThickness	Beer-Lambert 指数衰减的厚度系数，越厚衰减越多	**最强**
coatExtinction	Beer-Lambert 的消光系数，控制 RGB 各通道衰减量和颜色偏移	**最强且带色**
NdotV（视角）	影响折射角 cti，掠射时路径更长，Beer-Lambert 衰减更严重	视角相关

  - coat 对直接光 diffuse 没有颜色叠加，只有衰减

- 间接光
  阶段三：间接光合成（EvaluateBSDF_Env）
  - 具体代码

```
L = preLD.rgb * preLightData.specularFGD[i];
// TODOENERGY: If vlayered, should be done in ComputeAdding with FGD formulation for IBL. Same for LTC actually
// Incorrect, but just for now:
L *= preLightData.energyCompensationFactor[i];
L *= preLightData.hemiSpecularOcclusion[i];

UpdateLightingHierarchyWeights(preLightData.lobeReflectionWeight[i], tempWeight[i]);
smallestHierarchyWeight = min(smallestHierarchyWeight, preLightData.lobeReflectionWeight[i]);
envLighting += L * tempWeight[i];
```

  - 三叶(COAT/BASE_A/BASE_B)各自独立采样IBL后ADD

三个叶（COAT / BASE_A / BASE_B）各自**独立采样 IBL 后 ADD**：

$$\text{envLighting} = \sum_{i} \underbrace{\text{preLD}[i]}_{\text{各叶方向采样}} \times \underbrace{\text{specularFGD}[i]}_{\text{含 coat 换算的 F0 和 lobeMix 权重}} \times \underbrace{\text{energyCompensation}[i]}_{\text{多散射补偿}}$$

    - 叶:COAT
      - iblR方向：coat 法线反射方向
        iblPerceptualRoughness：bsdfData.coatPerceptualRoughness（原始，不经方差传播）
        specularFGD：F * (1 - lobeMix) * ...（coat 层 FGD）

- 叶:BASE_A
  - iblR方向：base 法线反射方向（折射修正）
    iblPerceptualRoughness：经方差传播后的 roughness_A
    specularFGD：vLayerEnergyCoeff[BOTTOM] * FGD * (1-lobeMix)
- 叶:BASE_b
  - iblR方向：同上
    iblPerceptualRoughness：经方差传播后的 roughness_B
    specularFGD：vLayerEnergyCoeff[BOTTOM] * FGD * lobeMix
- 间接光 diffuse 合成（ModifyBakedDiffuseLighting）

```
// Premultiply baked (possibly with back facing added) diffuse lighting information with diffuse
// pre-integration and energy term.
// preLightData.diffuseEnergy will be 1,1,1 if no vlayering or no VLAYERED_DIFFUSE_ENERGY_HACKED_TERM
// Note: When baking reflection probes, we approximate the diffuse with the fresnel0
builtinData.bakeDiffuseLighting *= preLightData.diffuseFGD * preLightData.diffuseEnergy * GetDiffuseOrDefaultColor(bsdfData, _ReplaceDiffuseForIndirect).rgb;

// The lobe specific specular occlusion data, along with the result of the screen space occlusion sampling
// will be computed in PreLightData.
```

  - diffuseEnergy 再次乘进间接光，这是暗部过暗的根源
- ▢ 最终输出
  阶段四：PostEvaluateBSDF 最终输出
  - 代码

```
// For now, we use (bsdfData.diffuseColor * preLightData.diffuseEnergy) directly.
float3 GTAOMultiBounceTintBase = (bsdfData.diffuseColor * preLightData.diffuseEnergy);
GetApplyScreenSpaceDiffuseOcclusionForDirect(GTAOMultiBounceTintBase, preLightData.screenSpaceAmbientOcclusion, directAmbientOcclusion, lighting);

// Subsurface scattering mode
float3 modifiedDiffuseColor = GetModifiedDiffuseColorForSSS(bsdfData);

// Compute diffuseOcclusion combining both the effects of the occlusion from data and from screen space, the
// later which has already been sampled in GetPreLightData (also see comments about GTAOMultiBounceTintBase
// above).
float3 diffuseOcclusion = GetDiffuseOcclusionGTAOMultiBounce(bsdfData.ambientOcclusion, preLightData.screenSpaceAmbientOcclusion, GTAOMultiBounceTintBase);

// Apply the albedo to the direct diffuse lighting (only once). The indirect (baked) diffuse lighting has
// already had the albedo applied in ModifyBakedDiffuseLighting() but we now also apply the diffuse occlusion
// on it. Specular occlusion has been applied per lobe during specular lighting evaluations before.
// We also add emissive since it is not merged with bakeDiffuseLighting in ModifyBakedDiffuseLighting.
// (also of lit deferred EncodeToGBuffer function).
lightLoopOutput.diffuseLighting = (modifiedDiffuseColor * lighting.direct.diffuse) + (builtinData.bakeDiffuseLighting * diffuseOcclusion) + builtinData.emissiveColor;

lightLoopOutput.specularLighting = lighting.direct.specular + lighting.indirect.specularReflected;
```

  - float3 GTAOMultiBounceTintBase = (bsdfData.diffuseColor * preLightData.diffuseEnergy);
    - diffuseEnergy 第三次出现，作为 GTAOMultiBounce 的 tint 基底，影响 AO 的颜色计算。

◯ 3.各向异性

- ▢ 1

◯ 4.虹彩(Iridescence)

- ▢ 1

◯ 5.能量补偿(GGX多散射)

- ▢ 1

◯ 6.HazyGloss(朦胧高光)

- ▢ 1

◯ 7.SSS

- 1

8.折射(Refraction)

- 1