

URP入门

Reuben

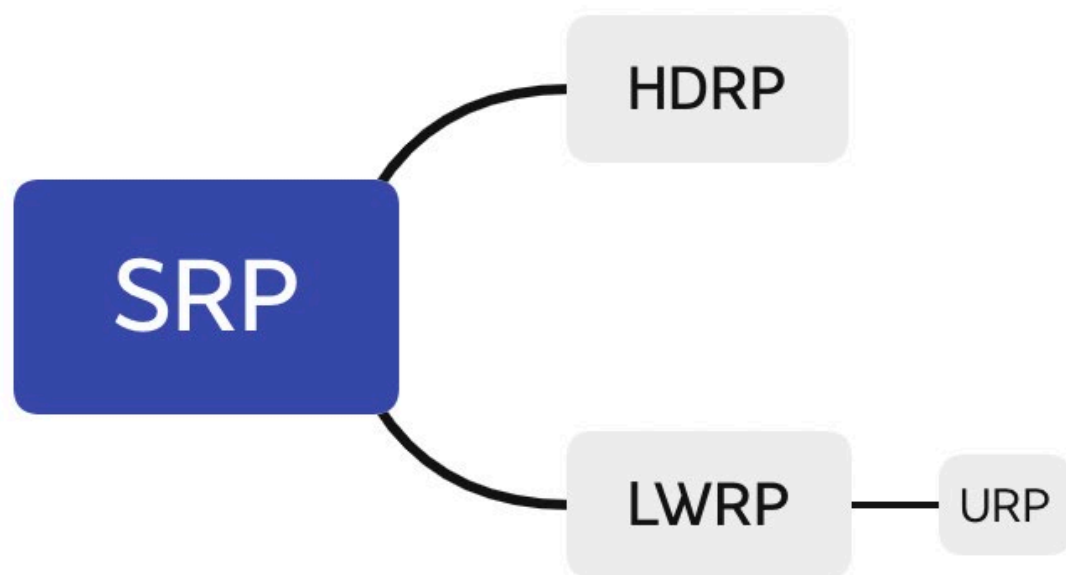
目录

- 01 URP概述
- 02 写一个Lit



URP

URP（Universal Render Pipeline，通用渲染管线）

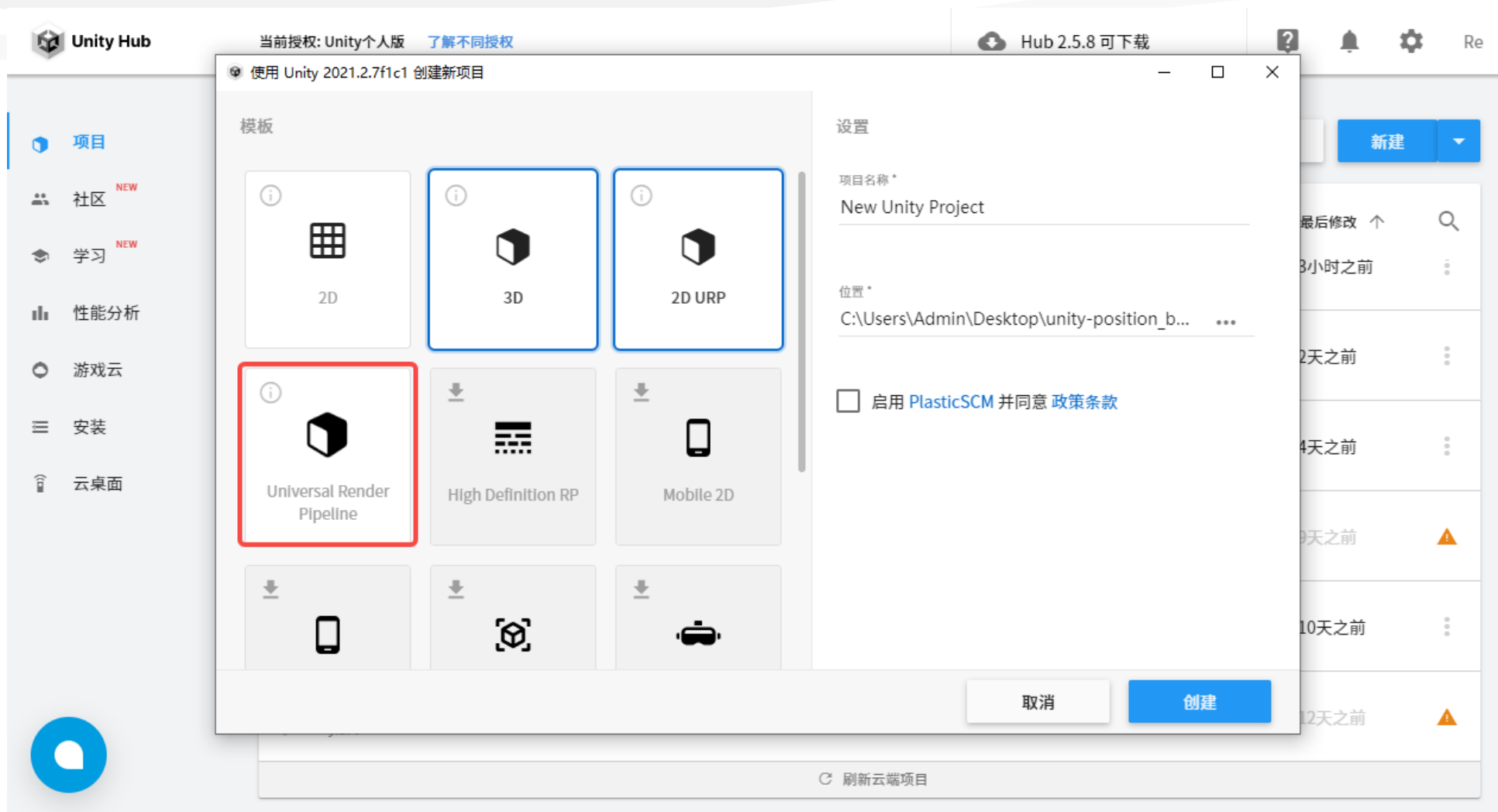


管他是什么，会用就行

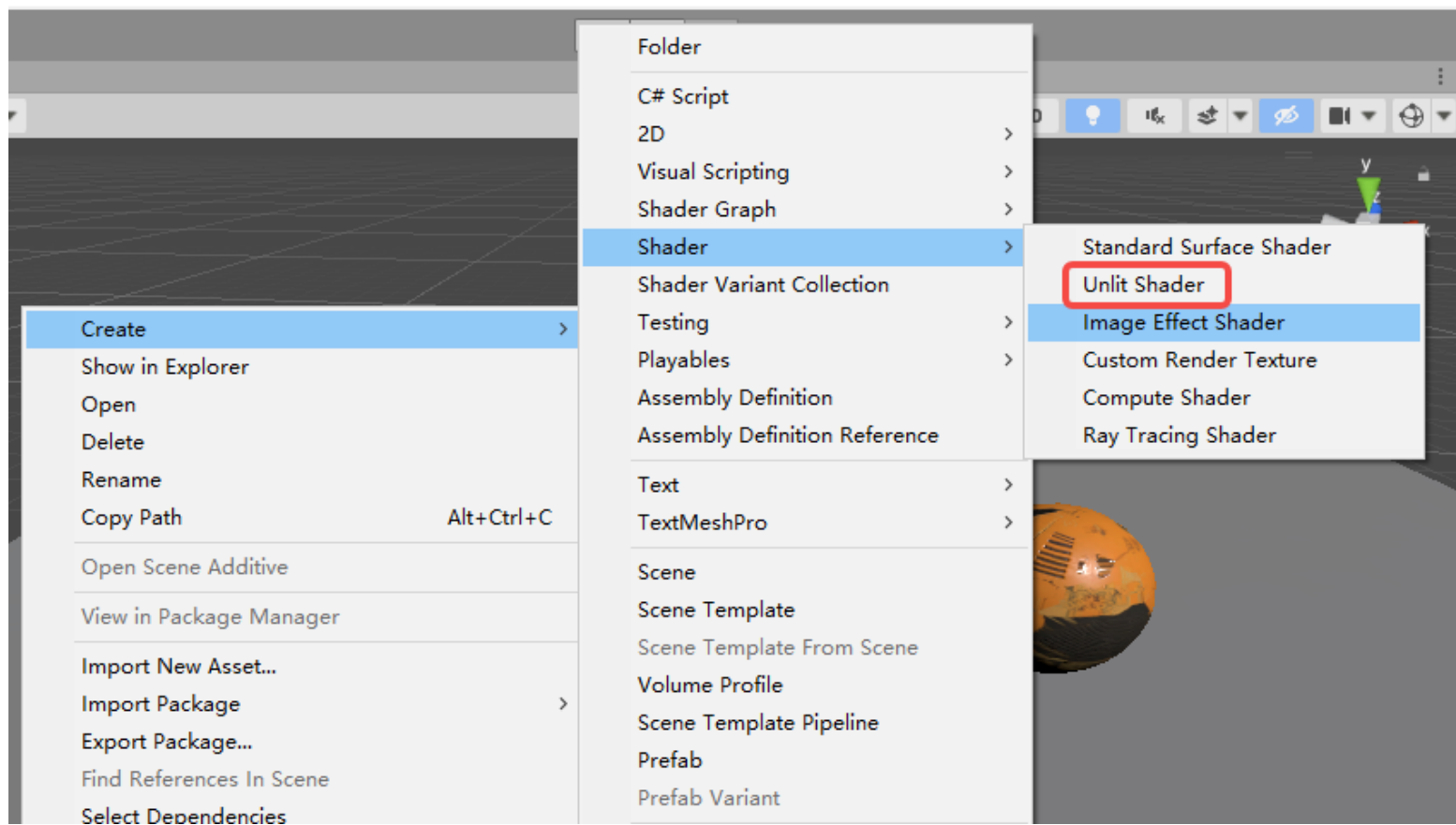
新建一个URP项目

你可以去Package Manager中升级URP，但其实还挺麻烦的，
我的建议是直接下一个URP模版

新建一个URP项目



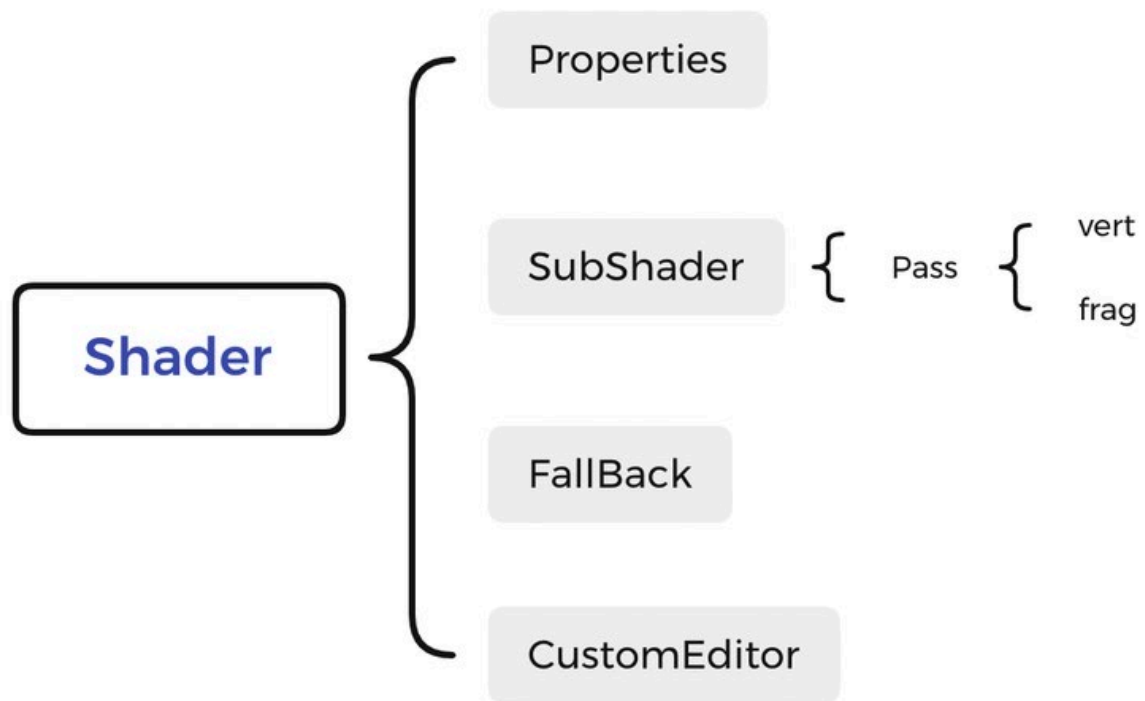
新建一个Shader



新建一个Shader

全删掉，替换成我的，你们可以一步一步跟着抄，但也可以直接用我这个，本次介绍的重点不是怎么写，而是这些API都干了什么

Shader长啥样



Properties

这东西是用来暴露接口，给shader中的属性用的

Unity中shader属性有三个来源，分别是per-instance、Material面板、全局shader属性

优先级per-instance最高，全局属性最低，感兴趣的可以去搜一下 `GetPropertyBlock()` 这个函数

Properties

Properties

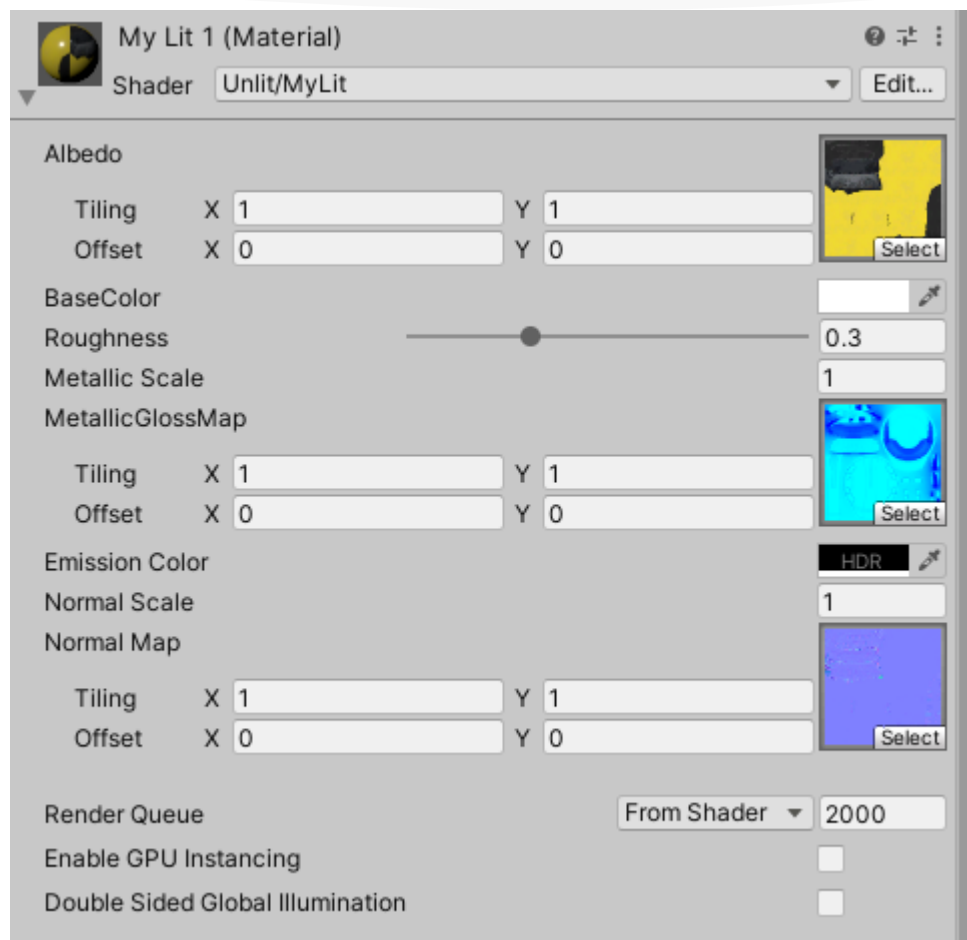
```
{  
    [MainTexture] _BaseMap("Albedo", 2D) = "white" {}  
    [MainColor] _BaseColor("BaseColor", Color) = (1,1,1,1)  
    _Roughness("Roughness", Range(0.0001,1)) = 0.5  
    _Metallic("Metallic Scale", float) = 0.0  
    _MetallicGlossMap("MetallicGlossMap", 2D) = "white" {}  
    [HDR] _EmissionColor("Emission Color", Color) = (0,0,0)  
    _BumpScale("Normal Scale", float) = 1.0  
    [Normal] _BumpMap("Normal Map", 2D) = "bump" {}  
  
    [HideInInspector][NoScaleOffset]unity_Lightmaps("unity_Lightmaps", 2DArray) = "" {}  
    [HideInInspector][NoScaleOffset]unity_LightmapsInd("unity_LightmapsInd", 2DArray) = "" {}  
}
```

Properties

我们这个Lit是金属粗糙度工作流的PBR

PBR就是基于物理的渲染，一般有两个工作流，一个是金属粗糙度，一个是镜面光泽度，这两个本质上是一样的，互有优劣，选M/R工作流主要是因为URP模板里的贴图是这个工作流的

Properties



你会发现这个面板的UI还挺丑的，而且宏编译之类的功能也没有，**URP**用了一个叫**CustomEditor**的东西，能让这东西好看点，咱今天就先不用了

SubShader

这个东西有啥用我也不是特别清楚，我只知道可以写LOD
一个SubShader中可以有多多个Pass，咱们这个Lit就只用到两个Pass，
一个是前向渲染，一个是阴影投射

```
SubShader
{
    Tags { "RenderType" = "Opaque" "RenderPipeline" = "UniversalPipeline" }

    Pass
    { ... }
    Pass
    { ... }
}
```

Tags

RenderPipeline用于告诉unity这个SubShader要在URP中运行，感兴趣的同学可以去看UniversalRenderPipeline.cs

```
Shader.globalRenderPipeline = "UniversalPipeline";
```

RenderType用于区分渲染的对象是什么，我没这么用过，好像可以用于替代渲染？

MyForwardPass

```
Pass
{
    Name "MyForwardPass"
    Tags{"LightMode" = "UniversalForward"}

    HLSLPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    //材质关键词
    #pragma shader_feature _NORMALMAP          //使用法线贴图
    #pragma shader_feature _EMISSION          //开启自发光
    //渲染流水线关键词
    #pragma multi_compile _ _ADDITIONAL_LIGHTS_VERTEX _ADDITIONAL_LIGHTS
    #pragma multi_compile _ _MAIN_LIGHT_SHADOWS _MAIN_LIGHT_SHADOWS_CASCADE _MAIN_LIGHT_SHADOWS_SCREEN //主光开启投射阴影
    #pragma multi_compile _ _SHADOWS_SOFT      //开启软阴影
    //Unity定义的关键词
    #pragma multi_compile _ DIRLIGHTMAP_COMBINED //开启 lightmap定向模式
    #pragma multi_compile _ LIGHTMAP_ON         //开启 lightmap
    #pragma multi_compile_fog                   //开启雾效

    //有了这个，就不用写那些采样和声明了
    // #include "Packages/com.unity.render-pipelines.universal/Shaders/LitInput.hlsl"
    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"
    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
```

编译指令

虽然我们这个shader没写编译指令，但还是要提一嘴，这东西是为了提高代码兼容性，根据平台选择HLSL编译器
列一下URP官方的shader（这东西好像不同URP版本有不少变化）

```
#pragma exclude_renderers gles gles3 glcore  
#pragma target 4.5
```


关键词

关键词有三种，材质关键词、渲染流水线关键词、unity定义的关键词，这些东西与shader变体有关

为什么shader要有变体呢？因为shader中执行分支的方式是两个都走一遍，然后抛弃掉错误的结果，这样会大幅降低性能（指数级），所以shader中不怎么用分支，需要用的时候就用宏编译，写变体

关键词

(看注释)

//材质关键词

#pragma shader_feature _NORMALMAP //使用法线贴图

#pragma shader_feature _EMISSION //开启自发光

//渲染流水线关键词

#pragma multi_compile _ _ADDITIONAL_LIGHTS_VERTEX _ADDITIONAL_LIGHTS

#pragma multi_compile _ _MAIN_LIGHT_SHADOWS _MAIN_LIGHT_SHADOWS_CASCADE _MAIN_LIGHT_SHADOWS_SCREEN //主光开启投射阴影

#pragma multi_compile _ _SHADOWS_SOFT //开启软阴影

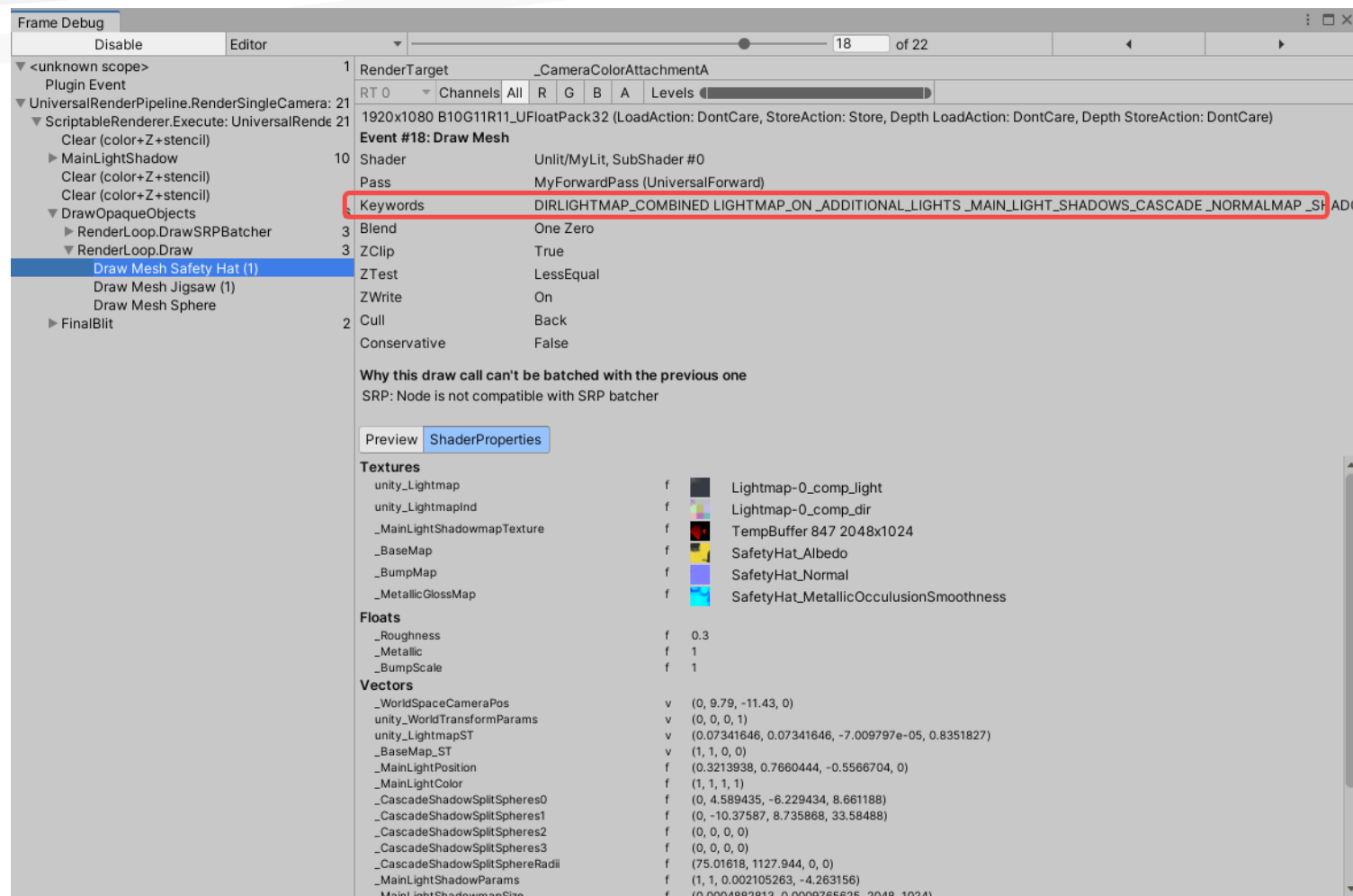
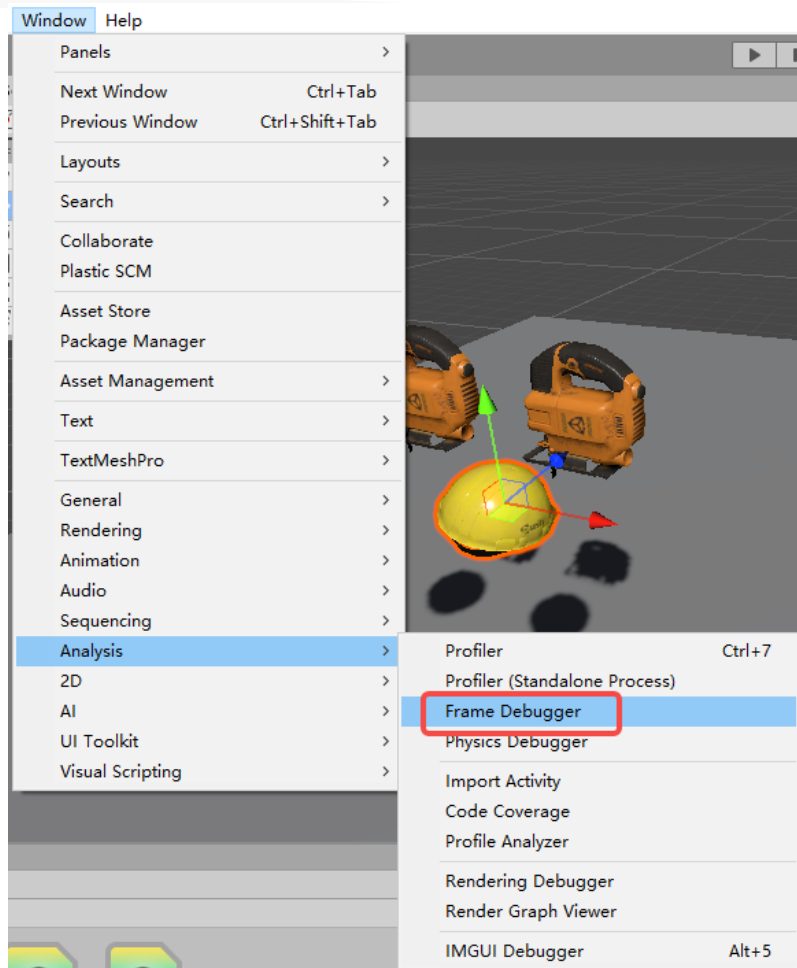
//Unity定义的关键词

#pragma multi_compile _ DIRLIGHTMAP_COMBINED //开启 lightmap定向模式

#pragma multi_compile _ LIGHTMAP_ON //开启 lightmap

#pragma multi_compile_fog //开启雾效

关键词



关键词

可以用Unity自带的工具查一下是否使用了关键词，咱由于没怎么管GUI，所以关键词可能会缺失，如果没有关键词，就用脚本打开

```
public class MatSetting : MonoBehaviour
{
    public Material mat; ☞ Serializable
    ☞ Event function
    void Start()
    {
        mat.EnableKeyword("_NORMALMAP");
    }
}
```

库

一些API库

```
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"  
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
```

顶点函数输入结构体Attributes

模型空间、世界空间、切线空间、齐次裁剪空间等等自行了解

```
struct Attributes
{
    float4 vertex: POSITION;    //模型空间顶点坐标
    float3 normal: NORMAL;    //模型空间法向量
    float4 tangent: TANGENT;    //模型空间切向量
    float2 uv: TEXCOORD0;    //第一套 uv
    float2 uv2: TEXCOORD1;    //lightmap uv
};
```

多套UV的原因是第一套UV不是全展的，而Lightmap uv是全展的，感兴趣可以去maya手动展一次

顶点函数输出结构体Varyings

```
struct Varyings
{
    float4 pos: SV_POSITION;           //齐次裁剪空间顶点坐标
    float2 uv: TEXCOORD0;              //纹理坐标
    float3 normalWS: TEXCOORD1;        //世界空间法线
    float3 viewDirWS: TEXCOORD2;       //世界空间视线方向

    #if defined(REQUIRES_WORLD_SPACE_POS_INTERPOLATOR)
        float3 posWS: TEXCOORD3;       //世界空间顶点位置
    #endif #if defined(REQUIRES_WORLD_SPACE_POS_INTERPOLATOR)

    DECLARE_LIGHTMAP_OR_SH(lightmapUV, vertexSH, 4); //声明光照贴图的纹理坐标, 光照贴图名称、球谐光照名称、纹理坐标索引

    #ifdef _NORMALMAP
        float4 tangentWS: TEXCOORD5;    //xyz是世界空间切向量, w是方向
    #endif #ifdef _NORMALMAP

    half4 fogFactorAndVertexLight: TEXCOORD6; //x是雾系数, yzw为顶点光照

    #if defined(REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR)
        float4 shadowCoord: TEXCOORD7;   //阴影坐标
    #endif #if defined(REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR)
};
```

声明属性变量

写在CBUFFER中是为了兼容

```
CBUFFER_START(UnityPerMaterial)
float4 _BaseMap_ST;
half4 _BaseColor;
half _Roughness;
half _Metallic;
half _BumpScale;
CBUFFER_END

TEXTURE2D(_BaseMap);    SAMPLER(sampler_BaseMap);
TEXTURE2D(_DetailNormalMap);    SAMPLER(sampler_DetailNormalMap);
TEXTURE2D(_BumpMap);    SAMPLER(sampler_BumpMap);
TEXTURE2D(_MetallicGlossMap);    SAMPLER(sampler_MetallicGlossMap);
```


顶点函数

WS世界空间
VS视图空间
OS模型空间
CS齐次裁剪空间
TS切线空间
NDC就是透视后
的CS空间

```
Varyings vert(Attributes v)
{
    Varyings o = (Varyings)0;
    VertexPositionInputs vertexInput = GetVertexPositionInputs(v.vertex.xyz); // 获得各个空间下的顶点坐标
    VertexNormalInputs normalInput = GetVertexNormalInputs(v.normal, v.tangent); // 获得各个空间下的法线切线坐标
    float3 viewDirWS = GetCameraPositionWS() - vertexInput.positionWS; // 世界空间视线方向=世界空间相机位置-世界空间顶点位置
    half3 vertexLight = VertexLighting(vertexInput.positionWS, normalInput.normalWS); // 遍历灯光做逐顶点光照 (考虑了衰减)
    half fogFactor = ComputeFogFactor(vertexInput.positionCS.z);

    o.uv = TRANSFORM_TEX(v.uv, _BaseMap); // 获得纹理坐标
    o.normalWS = normalInput.normalWS;
    o.viewDirWS = viewDirWS;

#ifdef _NORMALMAP
    real sign = v.tangent.w * GetOddNegativeScale();
    o.tangentWS = half4(normalInput.tangentWS.xyz, sign);
#endif

    OUTPUT_LIGHTMAP_UV(v.uv2, unity_LightmapST, o.lightmapUV); // lightmap uv
    OUTPUT_SH(o.normalWS.xyz, o.vertexSH); // SH

    o.fogFactorAndVertexLight = half4(fogFactor, vertexLight); // 计算雾效

#ifdef REQUIRES_WORLD_SPACE_POS_INTERPOLATOR
    o.posWS = vertexInput.positionWS;
#endif

#ifdef REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR
    o.shadowCoord = GetShadowCoord(vertexInput);
#endif

    o.pos = vertexInput.positionCS; // 齐次裁剪空间顶点坐标
    return o;
}
```

顶点函数

```
half3 vertexLight = VertexLighting(vertexInput.positionWS, normalInput.normalWS); //遍历灯光做逐顶点光照 (考虑了衰减)
```

这个函数只有在开启额外灯光，并且额外灯光是逐顶点光照时才会有用，不然是纯黑（咱这个Lit用不到）

```
#ifdef _NORMALMAP
    real sign = v.tangent.w * GetOddNegativeScale();
    o.tangentWS = half4(normalInput.tangentWS.xyz, sign);
#endif #ifdef _NORMALMAP
```

这个函数的返回值是 1 或者 -1，用于确定切线的方向

正戏正式开始

前面的东西和URP源码基本一致，只是略有删减，毕竟初始化这些东西其实应该都是一致的

片元函数

第一步仍是初始化，主要是贴图采样

```
//初始化视线
half3 viewDirWS = SafeNormalize(i.viewDirWS);
//初始化法线
half4 n = SAMPLE_TEXTURE2D(_BumpMap, sampler_BumpMap, i.uv); //采集切线空间法线
half3 normalTS = UnpackNormalScale(n, _BumpScale);
#ifdef _NORMALMAP
    float sgn = i.tangentWS.w; // should be either +1 or -1
    float3 bitangent = sgn * cross(i.normalWS.xyz, i.tangentWS.xyz); //次切线
    half3x3 tangentToWorld = half3x3(i.tangentWS.xyz, bitangent.xyz, i.normalWS.xyz); //TBN矩阵
    i.normalWS = TransformTangentToWorld(normalTS, tangentToWorld);
#else
    i.normalWS = i.normalWS;
#endif
i.normalWS = NormalizeNormalPerPixel(i.normalWS);
```

片元函数

```
// 切线法线
half4 n = SAMPLE_TEXTURE2D(_BumpMap, sampler_BumpMap, coord2:i.uv); //采
half3 normalTS = UnpackNormalScale(n, _BumpScale);
#ifdef _NORMALMAP
    float sgn = i.tangentWS.w; // should be either +1 or -1
```

这个函数的作用是采集的法线先标准化（具体的操作是先将 x y 映射到 $(-1, 1)$ ，在求 z （法线的模为 1），再整体乘法线系数

片元函数

法线贴图为什么是蓝色的呢？是因为现在的法线贴图存储的都是切线空间法线，在光滑处的值为（0，0，1），转化为RGB是（127，127，255）正好是蓝色

如果使用法线贴图，就用BTN矩阵将贴图中的切线空间法线转化为世界空间法线，这样就可以让低模使用高模的法线（貌似美术就是这么制作法线贴图的）

片元函数

```
//初始化阴影
float4 shadowCoord = float4(0, 0, 0, 0);
#if defined(REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR)
    shadowCoord = i.shadowCoord;
#elif defined(MAIN_LIGHT_CALCULATE_SHADOWS) #if defined(REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR)
    shadowCoord = TransformWorldToShadowCoord(i.posWS);
#endif
#endif #elif defined(MAIN_LIGHT_CALCULATE_SHADOWS)

Light mainLight = GetMainLight(shadowCoord); //获取带阴影的主光

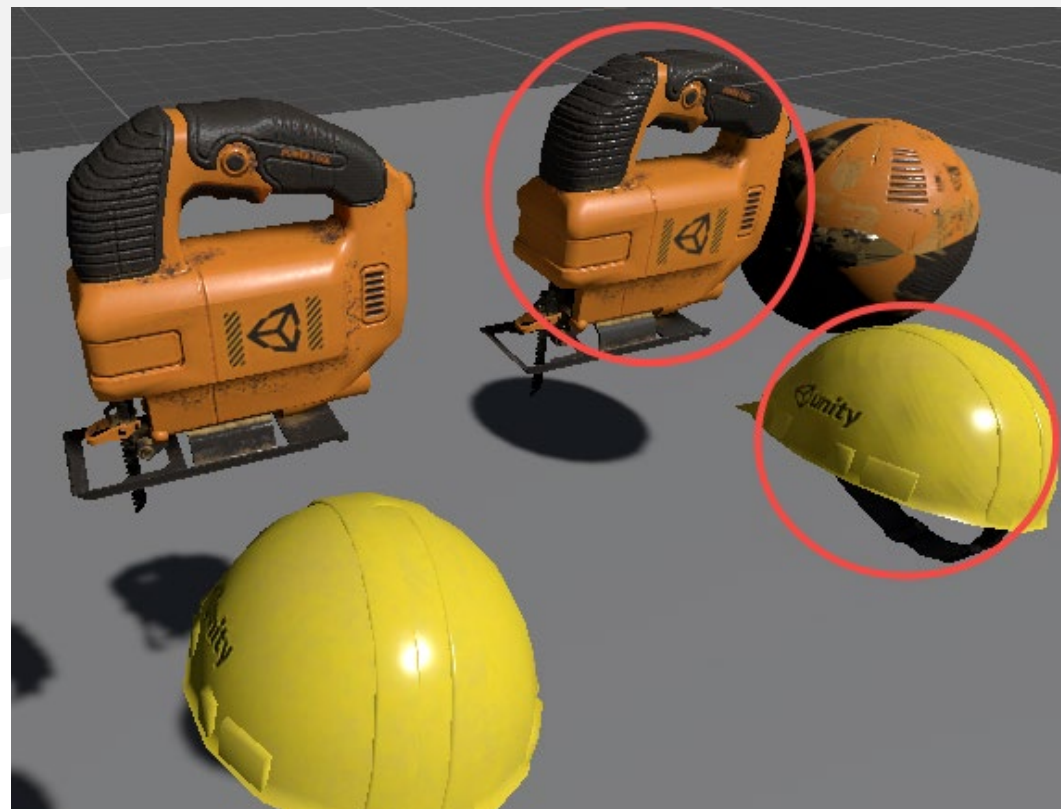
float3 L = normalize(mainLight.direction);
half3 H = normalize(viewDirWS + L); //半向量
float VoH = max(0.001, saturate(dot(viewDirWS, H)));
float NoV = max(0.001, saturate(dot(i.normalWS, viewDirWS)));
float NoL = max(0.001, saturate(dot(i.normalWS, L)));
float NoH = saturate(dot(i.normalWS, H));

half3 radiance = mainLight.color * (mainLight.shadowAttenuation * NoL); //获取光强
```

片元函数

我还真没自信能讲好PBR
(要不你们先去看看浅墨的白皮书?)

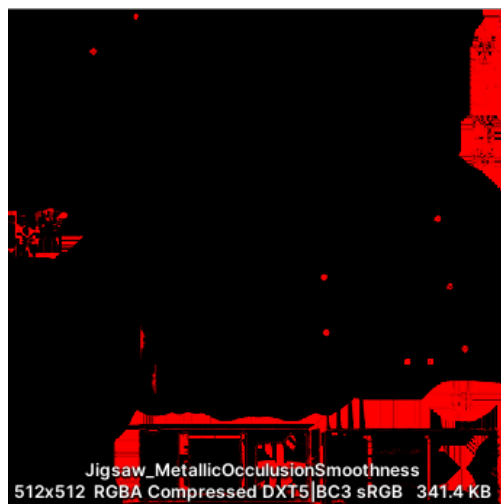
直接光Diffuse是兰伯特
直接光Specular是cook-Torrance
间接光Diffuse是Lightmap
没做间接光高光Specular
先呈现一下效果吧, 我感觉还挺对的



片元函数

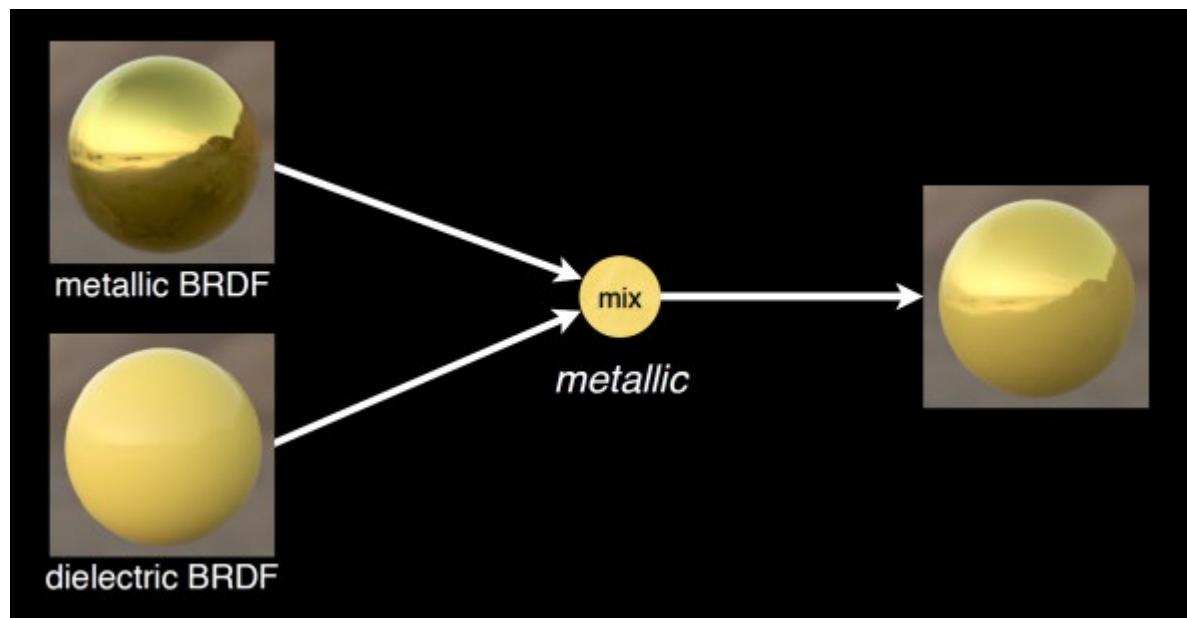
```
//初始化金属度粗糙度  
half4 specGloss = SAMPLE_TEXTURE2D(_MetallicGlossMap, sampler_MetallicGlossMap, coord2:i.uv);  
half metallic = specGloss.r * _Metallic;  
half roughness = specGloss.a * _Roughness;
```

我们这个Lit使用M / R workflow，贴图的R G A通道如下



金属度

迪士尼PBR其实就是将一个金属和非金属通过金属度进行线性插值，所以迪士尼PBR中baseColor同时具有金属的反射率颜色和非金属的漫反射颜色

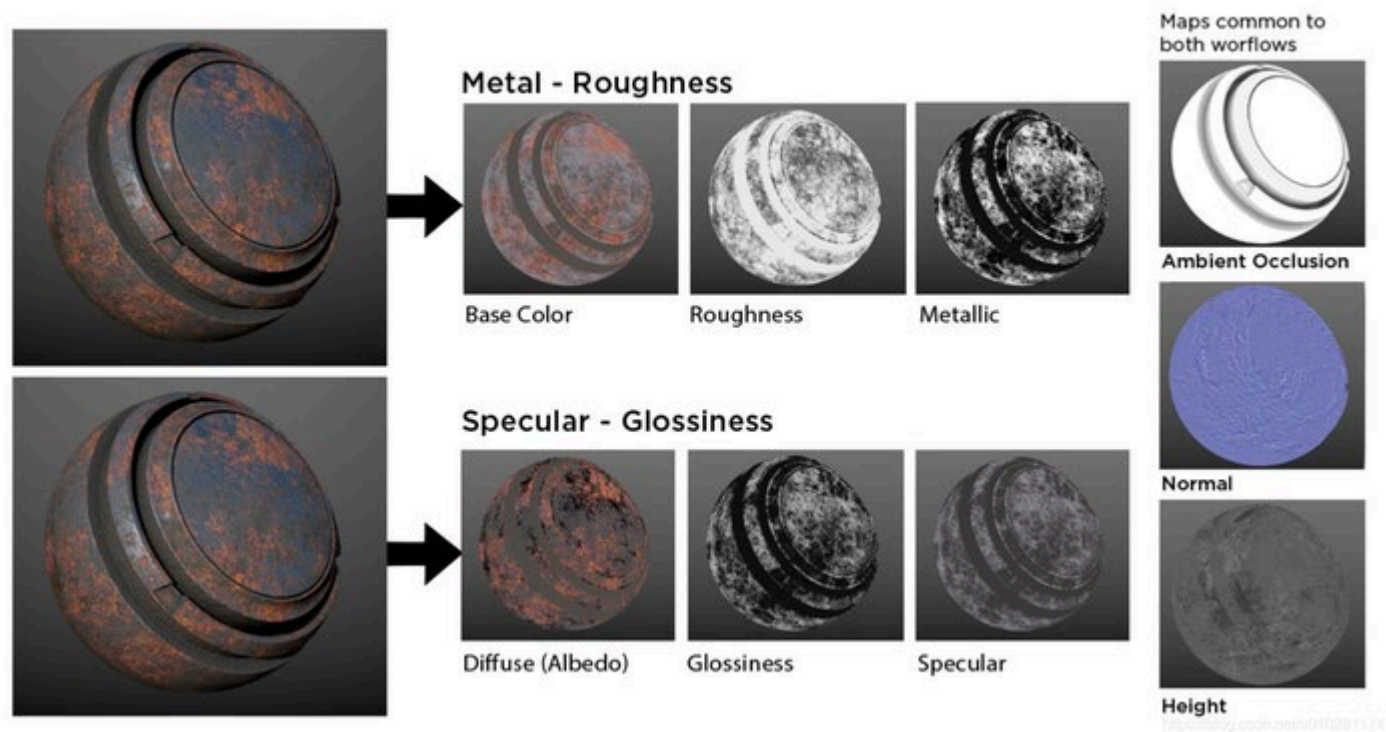


片元函数

M / R 工作流的baseColor是什么东西呢？
是非导体的反照率Albedo或金属F0，不是光照信息
所以要求brdfDiffuse和brdfSpecular
(PBR公式不保真)


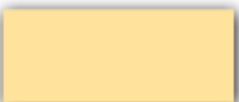
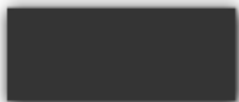
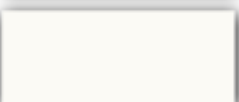
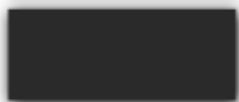
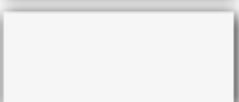
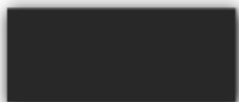
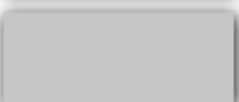
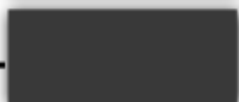
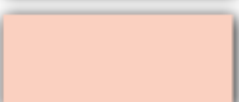
```
//菲涅尔项 F Schlick Fresnel
float3 F_Schlick = F0 + (1-F0) * pow(1 - VoH, 5.0);
float3 Kd = (1-F_Schlick)*(1-metallic);
float3 brdfDiffuse = albedo * Kd;

half3 F0 = lerp(half3(0.04, 0.04, 0.04), albedo, metallic);
```



F0

F0是0度入射的菲涅尔反射值，非金属是灰色，是一个Float，金属是镜面反射颜色，是一个Float3

Values don't drastically deviate in value		Plastic Common dielectrics 55-64 sRGB		Gold sRGB (255,226,155)
		Rusted Metal sRGB (52,52,52)		Silver sRGB (252,250,245)
		Water sRGB (43,43,43)		Aluminium sRGB (245,246,246)
		Ice sRGB (41,41,41)		Iron sRGB (196,199,199)
		Glass sRGB (57,57,57)		Copper sRGB (250,208,192)

漫反射

$$L_o = \int_{\Omega} f_r L_i \cos\theta_i d\omega_i = \int_{\Omega} (k_d \frac{c}{\pi} + k_s \frac{DFG}{4\cos\theta_i \cos\theta_o}) L_i \cos\theta_i d\omega_i$$

$$k_d \frac{c}{\pi}$$

这是漫反射，用的是兰伯特，但没除PI（UE4的1光强等于unity的3.14光强，unity官方为了和方便美术，所以没除PI）

```
float3 diffuseColor = brdfDiffuse * radiance;
```

菲涅尔

F项： 给定入射光线角度和材质信息，求反射的比例
(水面垂直的看通透，斜看如镜子)

$$F_{\text{Schlick}} = F_0 + (1 - F_0)(1 - \cos \theta_d)^5$$

```
//菲涅尔项 F Schlick Fresnel  
float3 F_Schlick = F0 + (1-F0) * pow(1 - VoH, 5.0);
```

“将麦克斯韦电磁方程组拆成矢量形式，在切线坐标系下推出，展开截断到五次方，就能得到这个公式”——某大佬是这样告诉我的

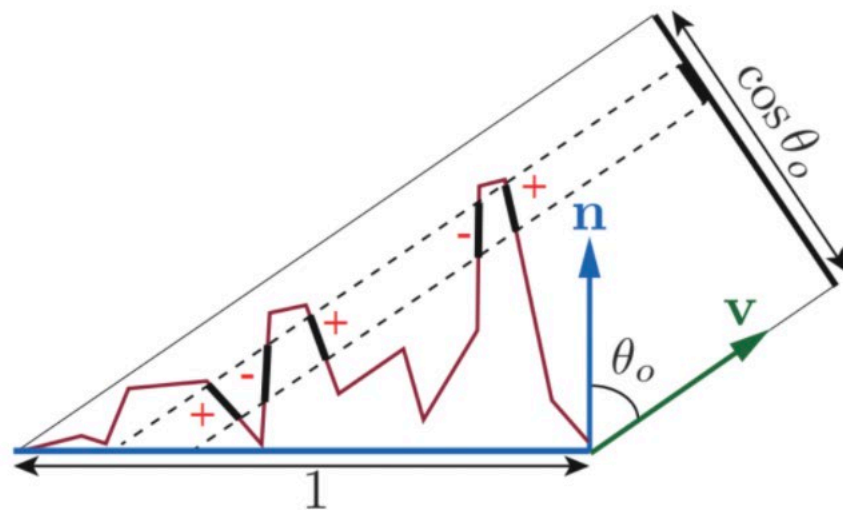
法线分布函数

D项：物体表面抽象成无数个微观理想镜面，D项用于描述其分布
D项分布符合公式：

$$\int D(\mathbf{m}) (\mathbf{n} \cdot \mathbf{m}) d\omega_m = \mathbf{v} \cdot \mathbf{n}$$

M是微表面法线

这个公式的含义是微表面的投影面积与宏观表面的投影面积相同



法线分布函数

D项

$$\alpha = \text{Roughness}^2$$

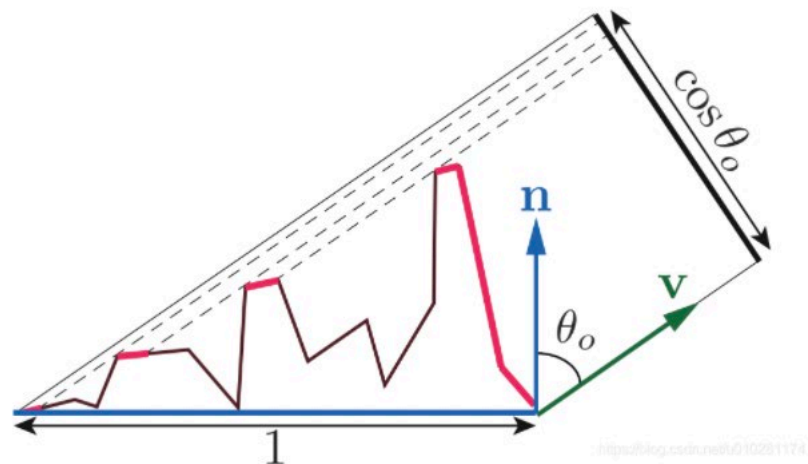
$$D(\mathbf{h}) = \frac{\alpha^2}{\pi ((\mathbf{n} \cdot \mathbf{h})^2 (\alpha^2 - 1) + 1)^2}$$

```
//法线分布项 D NDF GGX
float a = roughness * roughness;
float a2 = a * a;
float d = (NoH * a2 - NoH) * NoH + 1;
float D_GGX = a2 / (PI * d * d);
```


几何函数

G项

$$\int_{\Omega} G_1(\mathbf{m}, \mathbf{v}) D(\mathbf{m}) (\mathbf{v} \cdot \mathbf{m})^+ d\mathbf{m} = \mathbf{v} \cdot \mathbf{n}$$



这是第一个约束，表示宏观投影结果等于正面的投影结果之和
 $(\mathbf{v} \cdot \mathbf{m})^+$

这个表示将结果最小到0

其实还有其他约束，比如确定表面的轮廓，但咱没涉及

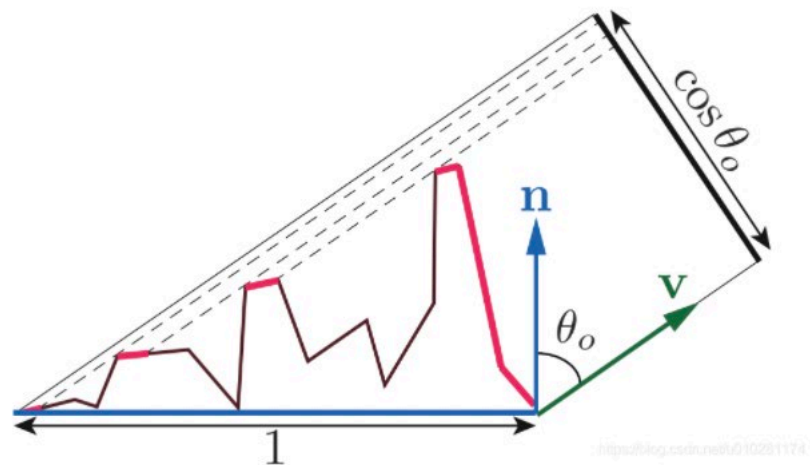
几何函数

G项

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

$$G_1(\mathbf{v}) = \frac{\mathbf{n} \cdot \mathbf{v}}{(\mathbf{n} \cdot \mathbf{v})(1 - k) + k}$$

$$G(\mathbf{l}, \mathbf{v}, \mathbf{h}) = G_1(\mathbf{l}) G_1(\mathbf{v})$$



//几何项 G

```
float k = (roughness + 1) * (roughness + 1) / 8;
```

```
float GV = NoV / (NoV * (1-k) + k);
```

```
float GL = NoL / (NoL * (1-k) + k);
```

```
float G_GGX = GV * GL;
```

片元函数

高光brdf

$$\frac{DFG}{4\cos\theta_i\cos\theta_o}$$

```
float3 brdf = F_Schlick * D_GGX * G_GGX / (4 * NoV * NoL);
```

直接光高光

```
float3 specularColor = brdf * radiance * PI;
```

片元函数

```
//间接光 diffuse
float3 indirectDiffuse = 0;
//lightmap间接光
#ifdef LIGHTMAP_ON
    float3 lm = SampleLightmap(i.lightmapUV, i.normalWS);
    indirectDiffuse.rgb = lm * albedo * Kd;
#endif #ifdef LIGHTMAP_ON
```

间接光只用了Lightmap

```
//计算雾效
color = MixFog(color, i.fogFactorAndVertexLight.x);
```

雾效

片元函数

结果:

```
float3 color = diffuseColor+specularColor+indirectDiffuse;  
//计算雾效  
color = MixFog(color, i.fogFactorAndVertexLight.x);  
return float4(color, 1);
```

阴影投射

加了一个阴影投射的Pass，
这个Pass比较简单，片元函数也没有返回值，我就直接用了URP自带的

```
Pass
{
    Name "MyShadowCaster"
    Tags{"LightMode" = "ShadowCaster"}

    ZWrite On
    ZTest LEqual
    Cull[Cull]

    HLSLPROGRAM
    #pragma only_renderers gles gles3 glcore d3d11
    #pragma target 2.0

    //-----
    // GPU Instancing
    #pragma multi_compile_instancing

    // -----
    // Material Keywords
    #pragma shader_feature_local_fragment _ALPHATEST_ON
    #pragma shader_feature_local_fragment _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A

    // -----
    // Universal Pipeline keywords

    // This is used during shadow map generation to differentiate between directional and punctual light
    #pragma multi_compile_vertex _ _CASTING_PUNCTUAL_LIGHT_SHADOW

    #pragma vertex ShadowPassVertex
    #pragma fragment ShadowPassFragment

    #include "Packages/com.unity.render-pipelines.universal/Shaders/LitInput.hlsl"
    #include "Packages/com.unity.render-pipelines.universal/Shaders/ShadowCasterPass.hlsl"
    ENDHLSL
}
```

谢谢

突然感觉我讲这东西有点贻笑大方了，PBR后续还是找个大佬专门讲吧，嘤嘤嘤