UE4 MOBILE PBR PIPELINE

TIAN

15 DEC 2021 12:00

TIAN

THE NEED

No matter developing shader in NPR or PBR, a well-understanding of the default shading pipeline is important, especially in Unreal Engine where all the shader things are packed in their source codes. To manipulate the default pipeline or create your own lighting model, it needs a well understanding of the engine shading pipeline.

THE MOBILE BASEPASS SHADER

CODES WITHOUT MACROS

`\Engine\Shaders\Private\MobileBasePassPixelShader.usf` This is the path of the UE4 mobile PBR shader. There are a lot of macro there, such as `MATERIALBLENDING_...`, `MATERIAL_SHADINGMODEL_UNLIT`, `FULLY_ROUGH`, `NONMETAL`, etc.. Let's just remove all of them and see what the code is like.

```
void Main(
        FVertexFactoryInterpolantsVSToPS Interpolants
        , FMobileBasePassInterpolantsVSToPS BasePassInterpolants
        , in float4 SvPosition : SV_Position
        OPTIONAL_IsFrontFace
        , out half4 OutColor    : SV_Target0
        )
{

        ResolvedView = ResolveView();

        SvPosition.y = ResolvedView.BufferSizeAndInvSize.y - SvPosition.y
```

```
{
        float4 ScreenPosition = SvPositionToResolvedScreenPosition
        float3 WorldPosition = BasePassInterpolants.PixelPosition.
        float3 WorldPositionExcludingWPO = BasePassInterpolants.Pi
        CalcMaterialParametersEx(MaterialParameters, PixelMaterial

}


// Store the results in local variables and reuse instead of calli
FGBufferData GBuffer = (FGBufferData)0;
GBuffer.WorldNormal = MaterialParameters.WorldNormal;
GBuffer.BaseColor = GetMaterialBaseColor(PixelMaterialInputs);
GBuffer.Metallic = GetMaterialMetallic(PixelMaterialInputs);
GBuffer.Specular = GetMaterialSpecular(PixelMaterialInputs);
GBuffer.Roughness = GetMaterialRoughness(PixelMaterialInputs);
GBuffer.ShadingModelID = GetMaterialShadingModel(PixelMaterialInpu
half MaterialAO = GetMaterialAmbientOcclusion(PixelMaterialInputs)


GBuffer.GBufferAO = MaterialAO;
// The smallest normalized value that can be represented in IEEE 7
// The code will make the following computation involving roughnes
// Therefore to prevent division by zero on devices that do not su
// must be >= 5.96e-8. We will clamp to 0.015625 because 0.015625^
//
// Note that we also clamp to 1.0 to match the deferred renderer o
// stored in an 8-bit value and thus automatically clamped at 1.0.
GBuffer.Roughness = max(0.015625, GetMaterialRoughness(PixelMateri

FMobileShadingModelContext ShadingModelContext = (FMobileShadingMo
ShadingModelContext.Opacity = GetMaterialOpacity(PixelMaterialInpu


half3 Color = 0;

half CustomData0 = GetMaterialCustomData0(MaterialParameters);
half CustomData1 = GetMaterialCustomData1(MaterialParameters);
InitShadingModelContext(ShadingModelContext, GBuffer, MaterialPara
float3 DiffuseDir = MaterialParameters.WorldNormal;

half IndirectIrradiance;
half3 IndirectColor;
ComputeIndirect(Interpolants, DiffuseDir, ShadingModelContext, Ind
Color += IndirectColor;

half3 SkyDiffuseLighting = GetSkySHDiffuseSimple(MaterialParameter
```

```
Color *= MaterialAO;
IndirectIrradiance *= MaterialAO;

// Shadow
half Shadow = GetPrimaryPrecomputedShadowMask(Interpolants).r;

half NoL = max(0, dot(MaterialParameters.WorldNormal, MobileDirect
half3 H = normalize(MaterialParameters.CameraVector + MobileDirect
half NoH = max(0, dot(MaterialParameters.WorldNormal, H));

// Direct Lighting (Directional light) + IBL
FMobileDirectLighting Lighting = MobileIntegrateBxDF(ShadingModelC
// MobileDirectionalLight.DirectionalLightDistanceFadeMADAndSpecul
Color += (Shadow) * MobileDirectionalLight.DirectionalLightColor.r


// Environment map has been prenormalized, scale by lightmap lumin
half3 SpecularIBL = GetImageBasedReflectionLighting(MaterialParame

Color += SpecularIBL * ShadingModelContext.SpecularColor;

// Local lights
...

        AccumulateLightingOfDynamicPointLight(MaterialParameters,









...

// Skylight
//@mw todo
// TODO: Also need to do specular.
Color += SkyDiffuseLighting * half3(ResolvedView.SkyLightColor.rgb

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

```
// NEEDS_BASEPASS_PIXEL_FOGGING is not allowed on mobile for the s

half3 Emissive = GetMaterialEmissive(PixelMaterialInputs);

Color += Emissive;

// On mobile, water (an opaque material) is rendered as trnaslucen
OutColor.rgb = Color * VertexFog.a + VertexFog.rgb;
```

TO VISUALIZE EACH PIECE

It's hard to know what's going on by just watching the codes. So let's just go to the usf file return the specific parameter that you want to see, for instance:

```
963   #if MATERIAL_SHADINGMODEL_DEFAULT_LIT
964   |   OutColor.rgb = ShadingModelContext.SpecularColor;
965   #endif
966   }
```

And go back to the engine, just make some minor changes, such as moving a node in the master material, then click apply, the shader will compile.
Here I used a standard PBR material in a simple lighting environment, containing a directional light, a skylight, and a background plane. The complete Lit output is:

COMPUTEINDIRECT

The pixel shader starting from **Main()** function, and at line 643 `half3 Color = 0;` the color starts adding layer by layer. At first, it is added by `IndirectColor`, calculated from **ComputeIndirect**:

```
{
        //To keep IndirectLightingCache conherence with PC, initialize the
        IndirectIrradiance = 0;
        Color = 0;


        // Indirect Diffuse
#if LQ_TEXTURE_LIGHTMAP
        float2 LightmapUV0, LightmapUV1;
        uint LightmapDataIndex;
        GetLightMapCoordinates(Interpolants, LightmapUV0, LightmapUV1, Lig

        half4 LightmapColor = GetLightMapColorLQ(LightmapUV0, LightmapUV1,
        Color += LightmapColor.rgb * ShadingModelContext.DiffuseColor * Vi
        IndirectIrradiance = LightmapColor.a;
        ...
```

This function samples the lightmap for static mesh with baked light. Notice that for mobile it is `LQ_TEXTURE_LIGHTMAP`. Here I don't have any lightmaps, the result is just black:

ENABLE_SKY_LIGHT

The **IndirectIrradiance** is calculated under **ENABLE_SKY_LIGHT**:

```
#if ENABLE_SKY_LIGHT
                half3 SkyDiffuseLighting = GetSkySHDiffuseSimple(MaterialP
                half3 DiffuseLookup = SkyDiffuseLighting * ResolvedView.Sk
                IndirectIrradiance += Luminance(DiffuseLookup);
#endif
```

Where,

SKYDIFFUSELIGHTING

`half3 SkyDiffuseLighting`

DIFFUSELOOKUP

`half3 DiffuseLookup`

INDIRECTIRRADIANCE

`IndirectIrradiance`

IndirectIrradiance will be used later in **SpecularIBL**

FMOBILEDIRECTLIGHTING

```
FMobileDirectLighting Lighting = MobileIntegrateBxDF(ShadingModelContext,
Color += (Shadow) * MobileDirectionalLight.DirectionalLightColor.rgb * (Li
```

Where Color += the `shadow`, directional light color, `Lighting.Diffuse` and `Lighting.Specular` and the directional light specular scale. Let's output the result of it:

You see that the shading is almost done except lacking of some sky light in the shadow area.
So, what has been done in that two lines of code?
Firstly, lets see the **FMobileDirectLighting Lighting** in `\Engine\Shaders\Private\MobileShadingModels.ush`.

LIGHTING.SPECULAR

- Calculated in `MobileShadingModels.ush`, is in the struct of **FMobileDirectLighting**.
- Which is the specular comes from the light, just like *Phong*.
- Looks like:

`Lighting.Specular`

LIGHTING.DIFFUSE

- Calculated in `MobileShadingModels.ush`, is in the struct of **FMobileDirectLighting**.
- Looks like:

Below is the part **FMobileDirectLighting Lighting** in `\Engine\Shaders\Private\MobileShadingModels.ush`

```
...
struct FMobileDirectLighting
{
        half3 Diffuse;
        half3 Specular;
};
...
Lighting.Specular = ShadingModelContext.SpecularColor * (NoL * CalcSpecula
Lighting.Diffuse = NoL * ShadingModelContext.DiffuseColor;
...
```

Lighting.Specular + Lighting.Diffuse will looks like:

CALCSPECULAR()

```
half CalcSpecular(half Roughness, half NoH)
{
        return (Roughness*0.25 + 0.25) * GGX_Mobile(Roughness, NoH);
}
```

GGX_MOBILE()

and **GGX_Mobile()** is in `\Engine\Shaders\Private\MobileGGX.ush`

```
half GGX_Mobile(half Roughness, float NoH)
{
    // Walter et al. 2007, "Microfacet Models for Refraction through Rough
        float OneMinusNoHSqr = 1.0 - NoH * NoH;
        half a = Roughness * Roughness;
        half n = NoH * a;
        half p = a / (OneMinusNoHSqr + n * n);
        half d = p * p;
        // clamp to avoid overlfow in a bright env
        return min(d, 2048.0);
}
```

SPECULARIBL

Continue scrolling down, here it comes **SpecularIBL**.

```
...
half3 SpecularIBL = GetImageBasedReflectionLighting(MaterialParameters, GB
Color += SpecularIBL * ShadingModelContext.SpecularColor;
...
```

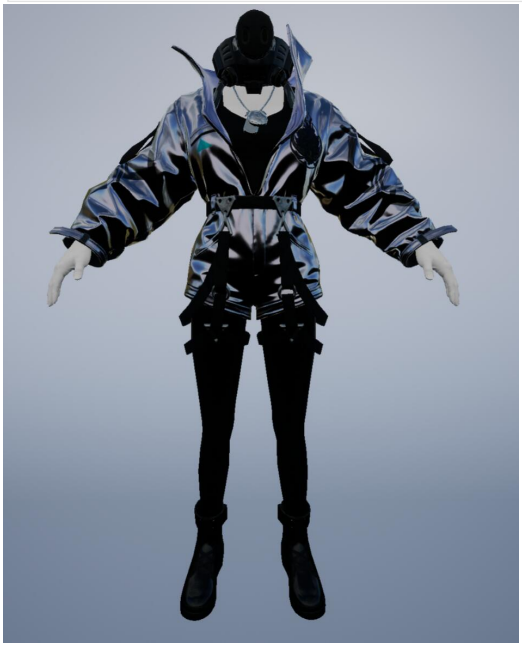
This is the specular created by image-based lighting, showing when your skylight sets to

Which looks like:

Later, **SpecularIBL** is multiplied with **ShadingModelContext.SpecularColor** and add on to the `Color` .

Which looks like:

```
SpecularIBL * ShadingModelContext.SpecularColor
```



Then add it on the `Color` :

```
Color += SpecularIBL * ShadingModelContext.SpecularColor;
```

See the metal necklace, it turns from black to silver with the light of cubemap after adding the `SpecularIBL` .

``

ACCUMULATELIGHTINGOFDYNAMICPOINTLIGHT

After the SpecularIBL, there is a function called **AccumulateLightingOfDynamicPointLight**, which is in charge of some dynamic lights except the directional light. This part calculated the attenuation of lights, NoL, and light color, then multiplied with `Lighting.Diffuse + Lighting.Specular`

```
FMobileDirectLighting Lighting = MobileIntegrateBxDF(ShadingModelContext,
Color += min(65000.0, (Attenuation) * LightColorAndFalloffExponent.rgb * (
```

`AccumulateLightingOfDynamicPointLight`

Then scroll down, there is another **ENABLE_SKY_LIGHT** macro, which does this:

This line uses of the `SkyDiffuseLighting` we got in the last **ENABLE_SKY_LIGHT** macro part before, and multiplied with skylight color , also multiplied with `ShadingModelContext.DiffuseColor * MaterialAO`.
We already seen **ShadingModelContext.SpecularColor** in the `SpecularIBL` part, and here we see **ShadingModelContext.DiffuseColor**. Read below to find what's their function is:

DIFFUSECOLOR, SPECPREENVBRDF, SPECULARCOLOR

```
struct FMobileShadingModelContext
{
        half Opacity;
        half3 DiffuseColor;
#if NONMETAL
        half SpecularColor;
#else
        half3 SpecularColor;
#endif
```

SHADINGMODELCONTEXT.DIFFUSECOLOR

- Calculated in `MobileShadingModels.ush`.
- `ShadingModelContext.DiffuseColor = GBuffer.BaseColor - GBuffer.BaseColor * GBuffer.Metallic;`
- Which indicates the effect of metallic value. The higher the metallic is, the darker the `DiffuseColor` is.
- Looks like:

`ShadingModelContext.DiffuseColor`

SHADINGMODELCONTEXT.SPECPREENVBRDF

- Calculated in `MobileShadingModels.ush`.
- `ShadingModelContext.SpecularColor = (DielectricSpecular - DielectricSpecular * GBuffer.Metallic) + GBuffer.BaseColor * GBuffer.Metallic;`

`ShadingModelContext.SpecPreEnvBrdf`

SHADINGMODELCONTEXT.SPECULARCOLOR

- Calculated in `MobileShadingModels.ush` .
- Which is **ShadingModelContext.SpecPreEnvBrdf** calculated in **EnvBRDFApprox**
- Which 'means we have plausible Fresnel and roughness behavior for image based lighting', according to Physically Based Shading on Mobile

`ShadingModelContext.SpecularColor`

Below is the relative codes of **ShadingModelContext.SpecularColor** and **ShadingModelContext.DiffuseColor** in `\Engine\Shaders\Private\MobileShadingModels.ush`

```
        half DielectricSpecular = 0.08 * GBuffer.Specular;
        ShadingModelContext.SpecularColor = (DielectricSpecular - Dielectr
        ShadingModelContext.SpecPreEnvBrdf = ShadingModelContext.SpecularC
        ShadingModelContext.DiffuseColor = GBuffer.BaseColor - GBuffer.Bas
        ...
        ShadingModelContext.SpecularColor = GetEnvBRDF(ShadingModelContext
```

ENVBRDFAPPROX

Below is the function **GetEnvBRDF()** in `\Engine\Shaders\Private\MobileShadingModels.ush`

```
half3 GetEnvBRDF(half3 SpecularColor, half Roughness, half NoV)
    {
        return EnvBRDFApprox(SpecularColor, Roughness, NoV);
    }
```

Below is **EnvBRDFApprox()** in `\Engine\Shaders\Private\BRDF.ush` . You can read Physically Based Shading on Mobile to see about this approximate environment BRDF method.

```
{
    // [ Lazarov 2013, "Getting More Physical in Call of Duty: Black O
    // Adaptation to fit our G term.
    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;

    // Anything less than 2% is physically impossible and is instead c
    // Note: this is needed for the 'specular' show flag to work, sinc
    AB.y *= saturate( 50.0 * SpecularColor.g );

    return SpecularColor * AB.x + AB.y;
}
```

ADD EMISSION

The last step is to adding emission on the `Color` if the material has this input.

```
half3 Emissive = GetMaterialEmissive(PixelMaterialInputs);
Color += Emissive;
```

NEXT STEP

Once finished the disassembly of the codes, we know what each part looks like and what it functions, we can start to rebuild it on other platforms such as Maya, SP, 3dMax, etc. Also, if you want to make custom non physically based shading, it's important to know the PBR pipeline first, so that you know where you should revise and write your own bits, and how to insert a custom lighting model.

Tian

HOME PAGE