Lighting Introduction

In the built-in pipeline , custom shaders that required lighting/shading was usually handled by **Surface Shaders** . These had the option to choose which lighting model to use , either the physically-based **Standard/StandardSpecular** or **Lambert** (diffuse) and **BlinnPhong** (specular) models . You could also write custom lighting models , which you would use if you wanted to produce a toon shaded result for example .

The Universal RP does not support surface shaders , however the ShaderLibrary does provide functions to help handle a lot of the lighting calculations for us . These are contained in **Lighting.hlsl** - (which isn't included automatically with Core.hlsl , it must be included separately) .

There are even functions inside that lighting file that can completely handle lighting for us , including **UniversalFragmentPBR** and **UniversalFragmentBlinnPhong** . These functions are really useful but there is still some setup involved , such as the InputData and SurfaceData structures that need to be passed into the functions .

We'll need a bunch of exposed Properties (which should also be added to the CBUFFER) to be able to send data into the shader and alter it per-material . You can check the templates for the exact properties used - for example , PBRLitTemplate.

There's also keywords that need to be defined before including the **Lighting.hlsl** file , to ensure the functions handle all the calculations we want , such as shadows and baked lighting . It's common for a shader to also include some shader feature keywords (not included below but see template) to be able to toggle features , e.g. to avoid unnecessary texture samples and make the shader cheaper .

```
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS
#pragma multi_compile _ _MAIN_LIGHT_SHADOWS_CASCADE
// Note , v11 changes this to :
// #pragma multi_compile _ _MAIN_LIGHT_SHADOWS
//    _MAIN_LIGHT_SHADOWS_CASCADE _MAIN_LIGHT_SHADOWS_SCREEN

#pragma multi_compile _ _ADDITIONAL_LIGHTS_VERTEX _ADDITIONAL_LIGHTS
#pragma multi_compile_fragment _ _ADDITIONAL_LIGHT_SHADOWS
#pragma multi_compile_fragment _ _SHADOWS_SOFT
#pragma multi_compile _ LIGHTMAP_ON
#pragma multi_compile _ DIRLIGHTMAP_COMBINED
#pragma multi_compile _ LIGHTMAP_SHADOW_MIXING
#pragma multi_compile _ SHADOWS_SHADOWMASK
#pragma multi_compile _ _SCREEN_SPACE_OCCLUSION

#pragma multi_compile_fog
#pragma multi_compile_instancing

// Include Lighting.hlsl
#include "Packages/com.unity.render-pipeline.universal/ShaderLibrary/
    Lighting.hlsl
```

Surface Data & Input Data

Both of these `UniversalFragmentPBR / UniversalFragmentBlinnPhong` functions use two structures to pass data through : `SurfaceData` and `InputData` .

The **SurfaceData** struct is responsible for sampling textures and providing the same inputs as you'd find on the URP/Lit shader . Specifically it contains the following :

```
struct SurfaceData
{
    half3 albedo;
    half3 specular;
    half metallic;
    half smoothness;
    half3 normalTS;
    half3 emission;
    half occlusion;
    half alpha;

    // And added in v10 :
    half clearCoatMask;
    half clearCoatSmoothness;
};
```

Note that you don't need to include this code , as this struct is part of the ShaderLibrary and we can instead include the file it is contained in . Prior to v10 , the struct existed in SurfaceInput.hlsl but the functions in Lighting.hlsl did not actually make use of it .

While you could still use the struct , you would instead need to do :

```
half4 color = UniversalFragmentPBR(inputData, surfaceData.albedo,
    surfaceData.metallic, surfaceData.specular,
    surfaceData.smoothness, surfaceData.occlusion,
    surfaceData.emission, surfaceData.alpha);
```

In v10+ the struct moved to it's own file , SurfaceData.hlsl , and the `UniversalFragmentPBR` function was updated so we can simply pass both structs through instead (for the `UniversalFragmentBlinnPhong` function a SurfaceData version is being added in v12 but current versions will need to split it . Examples shown later) .

```
half4 color = UniversalFragmentPBR(inputData, surfaceData);
```

We can still include **SurfaceInput.hlsl** instead though , as SurfaceData.hlsl will automatically be included by that file too , and it also contains the `_BaseMap` , `_BumpMap` and `_EmissionMap` texture definitions for us and some functions to assist with sampling them . We'll of course still need the Lighting.hlsl include too in order to have access to those functions .

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

The **InputData** struct is used to pass some extra things through that are required for lighting calculations . In v10 , includes the following :

```
struct InputData
{
    float3 positionWS;
    half3 normalWS;
    half3 viewDirectionWS;
    float4 shadowCoord;
    half fogCoord;
    half3 vertexLighting;
    half3 bakedGI;
    float2 normalizedScreenSpaceUV;
    half4 shadowMask;
};
```

Again , we don't need to include this code as it's already in Input.hlsl and that's automatically included when we include Core.hlsl anyway .

Since the lighting functions use these structs , we'll need to create them and set each variable it contains . To be more organised , we should do this in separate functions then call them in the fragment shader . The exact contents of the functions can vary slightly depending on what is actually needed for the lighting model .

For now I'm leaving the functions blank to first better see how the file is structured . The next few sections will go through the contents of the `InitializeSurfaceData` and `InitializeInputData` functions .

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

// Attributes, Varyings, Texture definitions etc.
// ...

// Functions
// ...

// SurfaceData & InputData
void InitializeSurfaceData(Varyings IN, out SurfaceData surfaceData){
    surfaceData = (SurfaceData)0; // avoids "not completely initalized" errors
    // ...
}

void InitializeInputData(Varyings IN, half3 normalTS, out InputData inputData) {
    inputData = (InputData)0; // avoids "not completely initalized" errors
    // ...
}

// Vertex Shader
// ...

// Fragment Shader
half4 LitPassFragment(Varyings IN) : SV_Target
{
    // Setup SurfaceData
    SurfaceData surfaceData;
    InitializeSurfaceData(IN, surfaceData);

    // Setup InputData
    InputData inputData;
    InitializeInputData(IN, surfaceData.normalTS, inputData);

    // Lighting Model, e.g.
    half4 color = UniversalFragmentPBR(inputData, surfaceData);

    // or
    // half4 color = UniversalFragmentBlinnPhong(inputData, surfaceData); // v12 only
    // half4 color = UniversalFragmentBlinnPhong(inputData, surfaceData.albedo, half4(surfaceDat
    // surfaceData.smoothness, surfaceData.emission, surfaceData.alpha);

    // or something custom

    // Handle Fog
    color.rgb = MixFog(color.rgb, inputData.fogCoord);
    return color;
}
```

It's also not too important that the functions are void as far as I'm aware . We could instead return the struct itself . I kinda prefer it that way , but I thought I'd try keeping it more consistent with how the URP/Lit shader code looks .

If you want to organise thins further , we could also move all the functions to **separate.hlsl** files and use a `#include` for it . This would also allow you to reuse that code for multiple shaders , and the Meta pass if you need to support that (discussed in more detail in a later section) . At the very least , I'd recommend having a hlsl file containing `InitializeSurfaceData` and it's required fcuntions / texture definitions .

InitializeInputData

As mentioned previously , our `InitializeInputData` function needs to set each of the variables inside the InputData struct , but this mainly obtaining the data passed through from the vertex stage and using some macros and functions (e.g. in order to handle transformations between spaces) .

This struct can also be the same for all lighting models , though I'm sure you could leave some parts out , e.g. if you aren't supporting baked lighting or the shadowMask . It is important to note that everything in the InputData struct needs to be initialised , so the first line in the function sets everything to 0 initally to avoid errors . You'll need to be careful then to not miss anything important though . It also helps prevent the shader breaking if an extra variable is added to the struct in future updates to the ShaderLibrary .

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

#if SHADER_LIBRARY_VERSION_MAJOR < 9
// These functions were added in URP v9.x versions, if we want to support URP versions before, w
// If you're in v10 you could remove this if you don't care about supporting prior versions.
// (Note, also using GetWorldSpaceViewDir in Vertex Shader)

// Computes the world space view direction (pointing towards the viewer) .
float3 GetWorldSpaceViewDir(float3 positionWS)
{
    if (unity_OrthoParams.w == 0)
    {
        // Perspective
        return _WorldSpaceCameraPos - positionWS;
    }
    else
    {
        // Orthographic
        float4x4 viewMat = GetWorldToViewMatrix();
        return viewMat[2].xyz;
    }
}

half3 GetWorldSpaceNormalizeViewDir(float3 positionWS)
{
    float3 viewDir = GetWorldSpaceViewDir(positionWS);
    if (unity_OrthoParams.w == 0)
    {
        // Perspective
        return half3(normalize(viewDir));
    }
    else
    {
        // Orthographic
        return half3(viewDir);
    }
}
#endif

void InitializeInputData(Varyings input, half3 normalTS, out InputData inputData)
{
    inputData = (InputData)0; // avoids "not completely initialized" errors

    inputData.positionWS = input.positionWS;

    #ifdef_NORMALMAP
        half3 viewDirWS = half3(input.normalWS.w, input.tangentWS.w, input.bitangentWS.w);
        inputData.normalWS = TransformTangentToWorld(normalTS, half3x3(input.tangentWS.xyz,
        input.bitangentWS.xyz, input.normalWS.xyz));
    #else
```

```
        half3 viewDirWS = GetWorldSpaceNormalizeViewDir(inputData.positionWS);
        inputData.normalWS = input.normalWS;
    #endif

    inputData.normalWS = NormalizeNormalPerPixel(inputData.normalWS);
    viewDirWS = SafeNormalize(viewDirWS);

    inputData.viewDirectionWS = viewDirWS;

    #if defined(REQUIRES_VERTEX_SHADOW_COORD_INTERPOLATOR)
        inputData.shadowCoord = input.shadowCoord;
    #elif defined(MAIN_LIGHT_CALCULATE_SHADOWS)
        inputData.shadowCoord = TransformWorldToShadowCoord(inputData.positionWS);
    #else
        inputData.shadowCoord = float4(0, 0, 0, 0);
    #endif

    // Fog
    #ifdef _ADDITIONAL_LIGHTS_VERTEX
        inputData.fogCoord = input.fogFactorAndVertexLight.x;
        inputData.vertexLighting = input.fogFactorAndVertexLight.yzw;
    #else
        inputData.fogCoord = input.fogFactor;
        inputData.vertexLighting = half3(0, 0, 0);
    #endif

    /* in v11/v12?, could use this :
    #ifdef _ADDITIONAL_LIGHTS_VERTEX
        inputData.fogCoord = InitializeInputDataFog(float4(inputData.positionWS, 1.0), input.fog
        inputData.vertexLighting = input.fogFactorAndVertexLight.yzw;
    #else
        inputData.fogCoord = InitializeInputDataFog(float4(inputData.positionWS, 1.0), input.fog
        inputData.vertexLighting = half3(0, 0, 0);
    #endif
    // Which currently just seems to force re-evaluating fog per fragment
    */

    inputData.bakedGI = SAMPLE_GI(input.lightmapUV, input.vertexSH, inputData.normalWS);
    inputData.normalizedScreenSpaceUV = GetNormalizedScreenSpaceUV(input.positionCS);
    inputData.shadowMask = SAMPLE_SHADOWMASK(input.lightmapUV);
}
```

It's a bit difficult to go through every function here , so I hope most of this is self-explanatory . The only thing that might not be that clear is the normalizedScreenSpaceUV which is currently only used to sample the **Screen Space Ambient Occlusion** texture later . If you don't need support that you could leave it out , but it also doesn't hurt to include it . If unused , the compiler will likely remove it anyway .

Also in case it's not clear , *`bakedGI`* refers to the **Baked Global Illumination** (baked lighting) and *`shadowMask`* refers specifically to when that is set to Shadowmask mode as an additional shadow mask texture is then used . The `SAMPLE_GI` and `SAMPLE_SHADOWMASK` macros will change when compiled depending on specific keywords . You can find those functions in Lighting.hlsl (split/moved to GlobalIllumination.hlsl in v12), and Shadows.hlsl of the URP ShaderLibrary .

Simple Lighting

The URP/SimpleLit shader uses the `UniversalFragmentBlinnPhone` function from Lighting.hlsl , which uses the **Lambert** and **Blinn-Phong** lighting models . If you aren't familiar with them I'm sure there are better resources online , but I'll attempt to explain them quickly :

Lambert models a perfectly **diffuse** surface , where light is reflected in all directions . This involves a **dot product** between the **light direction** and **normal vector** (both normalised) .

Phong models the **specular** part of the surface , where light is reflected more when the **view direction** aligns with the light vector reflected by the normal . Blinn-Phong is a slight alteration where instead of a reflected vecotr , it uses a **half vector** between the light vector and view direction which is more computationally efficient .

While it can be useful to know how to calculate these lighting models , they can be handled for us by the functions in the URP ShaderLibrary . The `UniversalFragmentBlinnPhong` function uses both the `LightingLambert` and `LightingSpecular` (blinn-phong model) functions included in Lighting.hlsl , which are :

```
half3 LightingLambert(half3 lightColor, half3 lightDir, half3 normal)
{
    half NdotL = saturate(dot(normal, lightDir));

    return lightColor * NdotL;
}

half3 LightingSpecular(half3 lightColor, half3 lightDir, half3 normal, half3 viewDir, half4 spec
{
   float3 halfVec = SafeNormalize(float3(lightDir) + float3(viewDir));
   half NdotH = half(saturate(dot(normal, halfVec)));
   half modifier = pow(NdotH, smoothness);
   half3 specularReflection = specular.rgb * modifier;

   return lightColor * specularReflection;
}
```

We could call these functions by including Lighting.hlsl , or copy the code out , but since the `UniversalFragmentBlinnPhong` does it for us we can use that instead . We need the two structs to pass into it though . The `InitializeInputData` function we went through in the section above , but for the

`InitializeSurfaceData` function , it can vary slightly depending on what we need to support (Blinn-Phong doesn't use the metallic like PBR for example) . I'm using the following :

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

// Textures, Samplers
// (note, BaseMap, BumpMap and EmissionMap is being defined by the SurfaceInput.hlsl include)
TEXTURE2D(_SpecGlossMap);
SAMPLER(sampler_SpecGlossMap);

// functions
half4 SampleSpecularSmoothness(float2 uv, half alpha, half4 specColor, TEXTURE2D_PARAM(specMap,
{
    half4 specularSmoothness = half4(0.0h, 0.0h, 0.0h, 0.0h);
    #ifdef _SPECGLOSSMAP
        specularSmoothness = SAMPLE_TEXTURE2D(specMap, sampler_specMap, uv) * specColor;
    #elif defined(_SPECULAR_COLOR)
        specularSmoothness = specColor;
    #endif

    #ifdef _GLOSSINESS_FROM_BASE_ALPHA
        specularSmoothness.a = exp2(10 * alpha + 1);
    #else
        specularSmoothness.a = exp2(10 * specularSmoothness.a + 1);
    #endif

    return specularSmoothness;
}

void InitializeSurfaceData(Varyings IN, out SurfaceData surfaceData)
{
    surfaceData = (SurfaceData)0; // avoids "not completely initialized" errors

    half4 baseMap = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv);

    #ifdef _ALPHATEST_ON
        // Alpha Clipping
        clip(baseMap.a - _Cutoff);
    #endif

    half4 diffuse = baseMap * _BaseColor * IN.color;
    surfaceData.albedo = diffuse.rgb;
    surfaceData.normalTS = SampleNormal(IN.uv, TEXTURE2D_ARGS(_BumpMap, sampler_BumpMap));
    surfaceData.emission = SampleEmission(IN.uv, _EmissionColor.rgb, TEXTURE2D_ARGS(_EmissionMap

    half4 specular = SampleSpecularSmoothness(IN.uv, diffuse.a, _SpecColor,
        TEXTURE2D_ARGS(_SpecGlossMap, sampler_SpecGlossMap)); surfaceData.specular = specular.rg
        surfaceData.smoothness = specular.a * _Smoothness;
}
```

As mentioned previously , in the fragment shader we can then call all these functions :

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

// ...
half4 LitPassFragment(Varyings IN) : SV_Target
{
    // Setup SurfaceData
    SurfaceData surfaceData;
    InitializeSurfaceData(IN, surfaceData);

    // Setup InputData
    InputData inputData;
    InitializeInputData(IN, surfaceData.normalTS, inputData);

    // Simple Lighting (Lambert a& BlinnPhong)
    // half4 color = UniversalFragmentBlinnPhong(inputData, surfaceData); // v12 only
    half4 color = UniversalFragmentBlinnPhong(inputData, surfaceData.albedo, half4(surfaceData.s
        surfaceData.smoothness, surfaceData.emission, surfaceData.alpha);

    color.rgb = MixFog(color.rgb, inputData.fogCoord);

    return color;
}
```

For a full example , see the URP SimpleLitTemplate .

PBR Lighting

The URP/Lit shader uses a more accurate **Physically Based Rendering** (PBR) model , which is based on Lambert and a Minimalist CookTorrance model . The exact implementation is slightly diffrernt according to the ShaderLibrary . If interested , you can find how it's implemented by looking at the `LightingPhysicallyBased` function in Lighting.hlsl and the DirectBRDFSpecular function in BRDF.hlsl .

We don't necessarily need to understand how it's implemented to use it though , we can just call the `UniversalFragmentPBR` function . As mentioned previously in v10+ it takes the two structs , InputData and SurfaceData . We've already discussed creating the `InitializeInputData` function in a couple sections above . For the `InitializeSurfaceData` we'll use :

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

// Textures, Samplers
// (note, BaseMap, BumpMap and EmissionMap is being defined
// by the SurfaceInput.hlsl include)
TEXTURE2D(_MetallicSpecGlossMap);
SAMPLER(sampler_MetallicSpecGlossMap);
TEXTURE2D(_OcclusionMap);
SAMPLER(sampler_OcclusionMap);

// Functions
half4 SampleMetallicSpecGloss(float2 uv, half albedoAlpha)
{
    half4 specGloss;
    #ifdef _METALLICSPECGLOSSMAP
        specGloss = SAMPLE_TEXTURE2D(_MetallicSpecGlossMap, uv)
        #ifdef _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            specGloss.a = albedoAlpha * _Smoothness;
        #else
            specGloss.a *= _Smoothness;
        #endif
    #else
        #if _SPECULAR_SETUP
            specGloss.rgb = _SpecColor.rgb;
        #else
            specGloss.rgb = _Metallic.rrr;
        #endif

        #ifdef _SMOOTHNESS_TEXTURE_ALBEDO_CHANNEL_A
            specGloss.a = albedoAlpha * _Smoothness;
        #else
            specGloss.a = _Smoothness;
        #endif
    #endif

    return specGloss;
}

half SampleOcclusion(float2 uv)
{
    #ifdef _OCCLUSIONMAP
        #if defined(SHADER_API_GLES)
            return SAMPLE_TEXTURE2D(_OcclusionMap,
            sampler_OcclusionMap, uv).g;
        #else
            half occ = SAMPLE_TEXTURE2D(_OcclusionMap,
            sampler_OcclusionMap, uv).g;
        #endif
    #else
        return 1.0;
    #endif
}

void InitializeSurfaceData(Varyings IN, out SurfaceData surfaceData)
{
    // avoids "not completely initialized" errors
    surfaceData = (SurfaceData)0;

    half4 albedoAlpha = SampleAlbedoAlpha(IN.uv, TEXTURE2D_ARGS(
        _BaseMap, sampler_BaseMap));
    surfaceData.alpha = Alpha(albedoAlpha.a, _BaseColor, _Cutoff);
    surfaceData.albedo = albedoAlpha.rgb * _BaseColor.rgb *
        IN.color.rgb;

    surfaceData.normalTS = SampleNormal(IN.uv, TEXTURE2D_ARGS(
        _BumpMap, sampler_BumpMap));
    surfaceData.emission = SampleEmission(IN.uv, _EmissionColor.rgb,
        TEXTURE_ARGS(_EmissionMap, sampler_EmissionMap));
    surfaceData.occlusion = SampleOcclusion(IN.uv);

    half4 specGloss = SampleMetallicSpecGloss(IN.uv, albedoAlpha.a);
    #if _SPECULAR_SETUP
        surfaceData.metallic = 1.0h;
        surfaceData.specular = specGloss.rgb;
    #else
        surfaceData.metallic = specGloss.r;
        surfaceData.specular = half3(0.0h, 0.0h, 0.0h);
    #endif

    surfaceData.smoothness = specGloss.a;
}
```

Then in the fragment shader :

```
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary
    /Lighting.hlsl"
// ...
half4 LitPassFragment(Varyings IN) : SV_Target
{
    // Setup SurfaceData
    SurfaceData surfaceData;
    InitializeSurfaceData(IN, surfaceData);

    // Setup InputData
    InputData inputData;
    InitializeInputData(IN, surfaceData.normalTS, inputData);

    // PBR Lighting
    half4 color = UniversalFragmentPBR(inputData, surfaceData);

    // Fog
    color.rgb = MixFog(color.rgb, inputData.forCoord);

    return color;
}
```