<http://www.xionggf.com/articles/graphic/u3d/u3d_built_in_shader_anatomy/underhood_surface_shader.html>Shader "Unlit/BumpedSpecular0"

{

Properties

{

\_Color ("Main Color", Color) = (1.0, 1.0, 1.0, 1.0)

\_MainTex ("Texture", 2D) = "white" {}

\_BumpMap ("Normal Map", 2D) = "bump" {}

}

SubShader

{

Tags { "RenderType" = "Opaque" }

CGPROGRAM

#pragma surface surf Lambert

#pragma target 3.0

sampler2D \_MainTex;

sampler2D \_BumpMap;

fixed4 \_Color;

struct Input

{

float2 uv\_MainTex;

float2 uv\_BumpMap;

};

void surf(Input IN, inout SurfaceOutput o)

{

fixed4 tex = tex2D(\_MainTex, IN.uv\_MainTex);

o.Albedo = tex.rgb \* \_Color.rgb;

o.Alpha = tex.a \* \_Color.a;

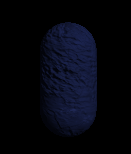
o.Normal = UnpackNormal(tex2D(\_BumpMap, IN.uv\_BumpMap));

}

ENDCG

}

}



我们在不修改代码的情况下，添加一个聚光灯：



和顶点/片元着色器需要包含到一个特定的Pass块不同，表面着色器的CG代码是直接而且也必须写在SubShader块中，unity会在背后为我们生成多个Pass。当然，可以在SubShader以开始处使用Tags来设置该表面着色器使用的标签，然后我们使用CGPROGRAM和ENDCG定义了表面着色器的具体代码。

一个表面着色器中最重要的部分是两个结构体以及它的编译指令。其中，两个结构体是表面着色器中不同函数之间信息传递的桥梁，而编译指令是我们和unity沟通的重要手段。

**编译指令**

编译指令是我们和unity沟通的重要方式，通过它可以告诉unity：“嘿，用这个表面函数设置表面属性，用这个光照模型模拟光照，我不要阴影和环境光，不要雾效”只需要依据代码，我们就可以完成这么多事情

编译指令最重要的作用是指明该表面着色器使用的**表面函数**和**光照函数**，并设置一些可选参数。表面着色器的CG块中第一句代码往往就是它的编译指令，编译指令的一般格式如下：

#pragma surface surfaceFunction lightModel [optionalparams]

其中，#pragma surface用于指明该编译指令是用于定义表面着色器的，在它后面需要指明使用的表面函数（surfaceFunction）和光照模型（lightModel），同时还可以使用一些可选参数来控制表面着色器的一些行为。

表面着色器的有点在于抽象出了“表面”这一概念，与之前遇到的顶点/片元抽象层不同，一个对象的表面属性定义了它的反射率、光滑度、透明度等值。而编译指令中的surfaceFunction就用于定义这些表面属性。surfaceFunction通常就是名为surf的函数（函数名可以是任意的），它的函数格式是固定的：

Void surf(Input IN, inout SurfaceOutput o)

Void surf(Input IN, inout SurfaceOutputStandard o)

Void surf(Input IN, inout SurfaceOutputStandardSpecular o)

其中，后两个是unity 5中由于引入了基于物理的渲染而新添加的两种结构体，SurfaceOutput、SurfaceOutputStandard和SurfaceOutputStandardSpecular都是Unity内置的结构体，它们需要配合不同的光照模型使用。

在表面函数中，会使用输入结构体Input IN来设置各种表面属性。并把这些属性存储在输出结构体SurfaceOutput、SurfaceOutputStandard或SurfaceOutputStandardSpecular中，再传递给光照函数计算光照结果。

**光照函数**

光照函数会使用表面函数中设置的各种表面属性来应用某些光照模型，进而模拟物体表面的光照效果。Unity内置了基于物理的光照模型函数Standard和StandardSpecular（在UnityPBSLighting.cginc文件中被定义），以及简单的非基于物理的光照模型函数Lambert和BlinnPhong（在Lighting.cginc文件中被定义），当然我们也可以定义自己的光照函数。

**其他可选参数**

在编译指令的最后，我们还可以设置一些可选参数，这些可选参数包含了很多非常有用的指令类型，例如，开启/设置透明度混合/透明度测试，指明自定义的顶点和颜色修改函数，控制生成的代码等。

1. 自定义修改函数：除了表面函数和光照模型外，表面着色器还可以支持其他两种自定义的函数：**顶点修改函数**和**颜色修改函数，**顶点修改函数允许我们自定义一些顶点属性，例如，把顶点颜色传递给表面函数，或是修改顶点位置，实现某些顶点动画。最后的颜色修改函数则可以在颜色绘制到屏幕前，最后一次修改颜色值，例如实现自定义的雾效
2. 阴影：我们可以通过一些指令来控制和阴影相关的代码。例如，addshadow参数会为表面着色器生成一个阴影投射的pass。通常情况下，unity可以直接在fallback中找到通用的光照模式为ShadowCaster的pass，从而将物体正确地渲染到深度和阴影纹理中，但对于一些进行了顶点动画、透明度测试的物体，我们需要对阴影的投影进行特殊处理，来为它们产生正确的阴影。Fullforwardshadows参数则可以在前向渲染路径中支持所有光源类型的阴影。默认情况下，unity只支持最重要的平行光的阴影，如果我们需要让点光源或聚光灯在前向渲染中也可以有阴影，就可以欠佳这个参数，相反，我们也可以使用noshadow参数来禁用阴影
3. 透明度混合和透明度测试：alpha和alphatest
4. 光照：比如noambient参数会告诉unity不要应用任何环境光照或光照探针
5. 控制代码生成：一些指令还可以控制由表面着色器自动生成的代码，默认情况下，unity会为一个表面着色器生成相应的前向渲染路径、延迟渲染路径使用的pass，这会导致生成shader文件比较大，如果我们确定该表面着色器只会在某些渲染路径中使用，就可以exclude\_path:deferred, exclude\_path:forward和exclude\_path:prepass来告诉unity不需要为某些渲染路径生成代码

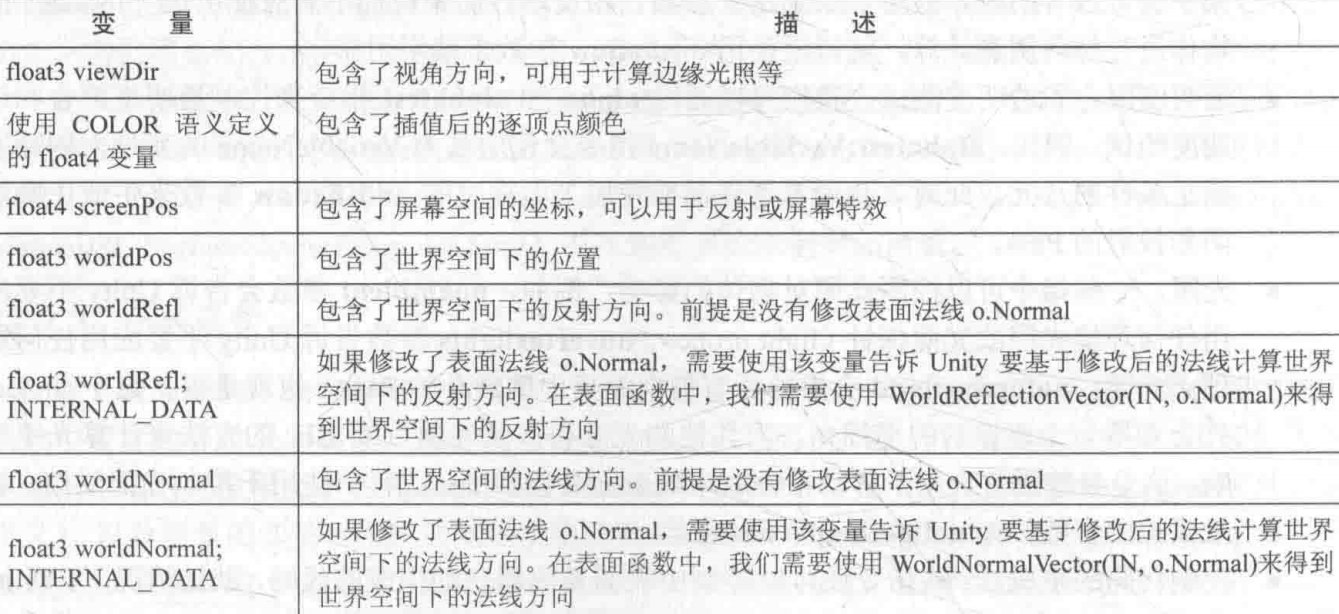
**两个结构体**

上面我们可以看到，表面着色器支持最多自定义4中关键的函数：表面函数（用于设置各种表面性质，如反射率、法线等），光照函数（定义表面使用的光照模型），顶点修改函数（修改或传递顶点属性），最后的颜色修改函数（对最后的颜色进行修改）。那么，这些函数之间的信息传递是怎么实现的呢？一个表面主搜而起需要使用两个结构体：表面函数的输入结构体Input，以及存储了表面属性的结构体SurfaceOutput（Unity 5新引入了另外两个同种的结构体SurfaceOutputStandard和SurfaceOutputStandardSpecular）

1. 数据来源：Input结构体

Input结构体包含了许多表面属性的数据来源，因此，它会作为表面函数的输入结构体（如果自定义了顶点修改函数，它还会是

顶点修改函数的输出结构体）。Input支持很多内置的变量名，通过这些变量名，我们告诉Unity需要使用的数据信息。下表列出了Input结构体中内置的其他变量：

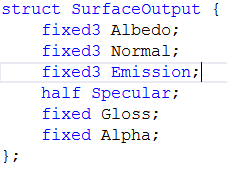


上面这些数据都是unity背后为我们准备好的，一个例外的情况是，我们自定义了顶点修改函数，并需要向表面函数中传递一些自定义的数据。例如，为了自定义雾效，我们可能需要在顶点修改函数中根据顶点在视角空间下的位置信息计算雾效混合系数，这样我们就可以在Input结构体中定义一个名为half fog的变量，把机选结果存储在该变量后进行输出。

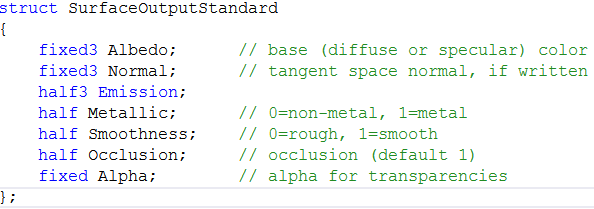
1. 表面属性：SurfaceOutput结构体

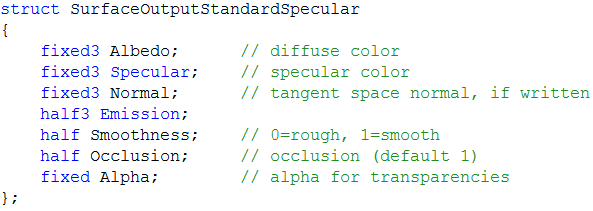
有了Input结构体来提供这些所需要的数据后，我们就可以据此计算各种表面属性。这个结构体就是用于存储这些表面属性的

结构体，即SurfaceOutput、SurfaceOutputStandard、SurfaceOutputStandardSpecular，它会作为表面函数的输出，随后会作为光照函数的输入来进行各种光照计算。这个结构体里面的变量是提前就声明好的，不可改变（如果没有对某些变量赋值，就会使用默认值），比如SurfaceOutput可以在Lighting.cginc文件中找到：



而SurfaceOutputStandard、SurfaceOutputStandardSpecular得声明可以在UnityPBSLighting.cginc中找到：





在一个表面着色器中，只需要选择上述三者中的其一即可，这取决于我们选择使用的光照模型。Unity内置的光照模型有两种，一种是unity 5之前的简单的、非基于物理的光照模型，包括了Lambert和BlinnPhong；另一种是Unity 5添加的基于物理的光照模型，包括Standard和StandardSpecular，这种模型会更加符合物理规律，但计算也会复杂很多。

如果使用了非基于物理的光照模型，我们通常会使用SurfaceOutput结构体，而如果使用了基于物理的光照模型Standard或StandardSpecular，我们会分别使用SurfaceOutputStandard或SurfaceOutputStandardSpecular结构体。其中，SurfaceOutputStandard结构体用于默认的金属工作流程（Metallic Workflow），对应Standard光照函数；而SurfaceOUtputStandardSpecular结构体用于高光工作流程（Specular Workflow），对应了StandardSpecular光照函数，这里我们重点看看SurfaceOUtput结构体中的变量和含义。在表面函数中，我们需要根据Input结构体传递的各个变量计算表面属性，在SurfaceOutput结构体中，这些表面属性包括了：

Fixed3 Albedo：对光源的反射率，通常由纹理采样和颜色属性的乘积计算而得

Fixed3 Normal：表面法线方向

Fixed3 Emission：自发光。Unity通常会在片元着色器最后输出前（并在最后的顶点函数被调用前，如果定义了的话），使用类似下面的语句进行简单的颜色叠加：c.rgb += o.Emission

Half Specular：高光反射中的指数部分的系数，影响高光发射的计算。例如，如果使用了内置的BlinnPhong光照函数，它会使用如下语句计算高光反射的强度：float spec = pow(nh, s.Specular \*128.0) \* s.Gloss

Fixed Gloss：高光发射中的强度系数

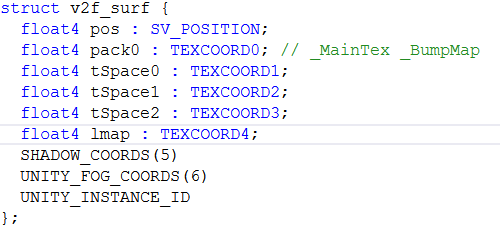
Fixed Alpha：透明通道，如果开启了透明度的话，会使用该值进行颜色混合。

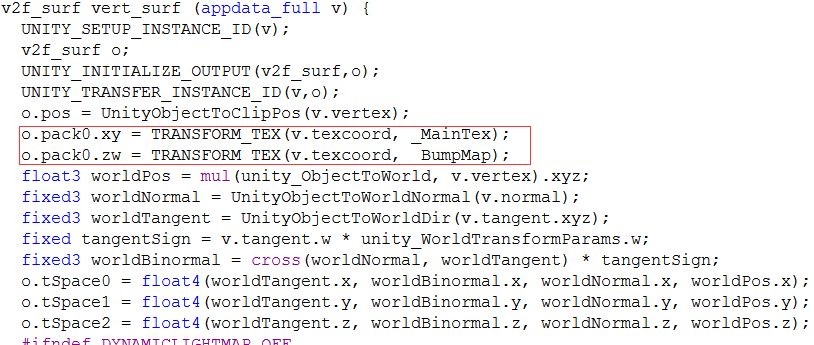
**（时刻记住，表面着色器本质上就是包含了很多Pass顶点/片元着色器）**

**背后的原理**

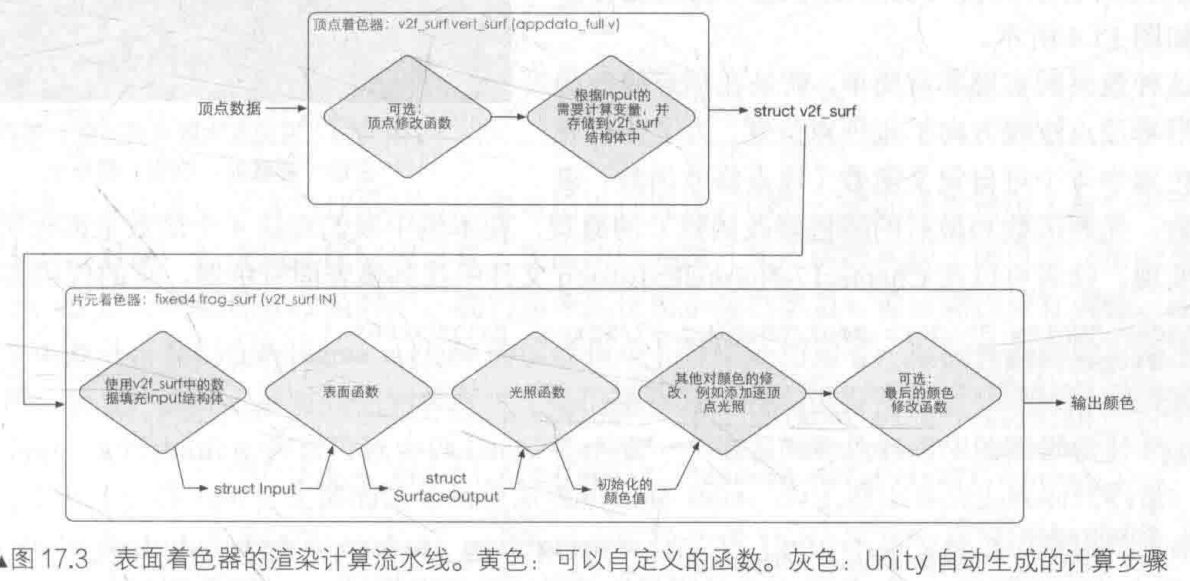
Unity在背后会根据表面着色器生成一个包含多个Pass的顶点/片元着色器。这些Pass有些是为了针对不同的渲染路径，例如，默认情况下，unity会为前向渲染路径生成LightMode为ForwardBase和ForwardAdd的Pass，为unity 5之前的延迟渲染路径生成LightMode为PrePassBase和PrePassFinal的Pass，为unity 5之后的延迟渲染路径生成LightMode为Deffered的Pass，还有一些Pass是用于产生额外的信息。例如，为了给光照映射和动态全局光照提取表面信息，unity会生成一个LightMode为Meta的Pass。有些表面着色器由于修改了顶点位置，因此，我们可以利用addshadow编译指令为它生成相应的LightMode为ShadowCaster的阴影投射Pass，这些Pass的生成都是基于我们在表面着色器中的编译指令和自定义函数。

以LightMode=“ForwardBase”为例，unity对该Pass的自动生成过程大致如下：





1）直接将表面着色器中的CGPROGRAM和ENDCG之间的代码复制过来，这些代码包括了我们对Input结构体、表面函数、光照函数（如果自定义的话）等变量和函数的定义，这些函数和变量会在之后的处理过程中被当成正常的结构体和函数进行调用



2）unity会分析上述代码，并据此生成顶点着色器的输出---v2f\_surf结构体，用于在顶点着色器和片元着色器之间进行数据传递。Unity会分析我们在自定义函数中所使用的变量，例如，纹理坐标、视角方向、反射方向等。如果需要，它就会在v2f\_surf中生成相应的变量，而且，即便我们在Input中定义了某些变量（如某些纹理坐标），但unity在分析后续代码时发现我们并没有使用这些变量，那么这些变量实际上是不会再v2f\_surf中生成的，也就是说，unity做了一些优化，v2f\_surf中还包含了一些其他需要的变量，例如阴影纹理坐标、光照纹理坐标、逐顶点光照等

1. 接着，生成顶点着色器
2. 如果我们自定义了顶点修改函数，unity会首先调用顶点修改函数来修改顶点数据，或填充自定义的Input结构体中的变量，然后，Unity会分析顶点修改函数中修改的数据，在需要时通过Input结构体将修改结果存储到v2f\_surf相应的变量中
3. 计算v2f\_surf中其他生成的变量。这主要包括了顶点位置、纹理坐标、法线方向、逐顶点光照、光照纹理的采样坐标等，当然我们可以通过编译指令来控制某些变量是否需要计算
4. 最后，将v2f\_surf传递给接下来的片元着色器
5. 生成片元着色器
6. 使用v2f\_surf中的对应变量填充Input结构体，例如，纹理坐标、视角方向等
7. 调用我们自定义的表面函数填充SurfaceOutput结构体
8. 调用光照函数得到初始的颜色值，如果使用的是内置的Lambert或BlinnPhong光照函数，Unity还会计算动态全局光照，并添加到光照模型的计算中
9. 进行其他的颜色叠加。例如，如果没有使用光照烘焙，还会添加逐顶点光照的影响
10. 最后，如果自定义了最后的颜色修改函数，unity就会调用它进行最后的颜色修改

下面的例子全部采用自定义实现：

Shader "Custom/NormalExtrusion" {

Properties

{

\_ColorTint ("Color Tint", Color) = (1.0, 1.0, 1.0, 1.0)

\_MainTex ("Base (RGB)", 2D) = "white" {}

\_BumpMap ("Normal Map", 2D) = "bump" {}

\_Amount ("Extrusion Amount", Range(-0.5, 0.5)) = 0.1

}

SubShader

{

Tags { "RenderType" = "Opaque" }

LOD 200

CGPROGRAM

/\* surf : which surface function

CustomLambert : which lighting model to use

vertex:myvertx : use custom vertex modification function

finalcolor:mycolor : use custom final color modification function

addshadow : generate a shadow caster pass. Because we modify the vertex position, the shadow needs special shadows handling

exclude\_path:defferred : do not generate passes for deferred rendering path

nometa : do not generate a "meta" pass (that's used by lightingmapping & dynamix global illymination to extract surface information)

\*/

#pragma surface surf CustomLambert vertex:myvertex finalcolor:mycolor addshadow exclude\_path:deferred exclude\_path:prepass nometa

#pragma target 3.0

fixed4 \_ColorTint;

sampler2D \_MainTex;

sampler2D \_BumpMap;

half \_Amount;

struct Input

{

float2 uv\_MainTex;

float2 uv\_BumpMap;

};

void myvertex(inout appdata\_full v)

{

v.vertex.xyz += v.normal \* \_Amount;

}

void mycolor(Input IN, SurfaceOutput o, inout fixed4 color)

{

color \*= \_ColorTint;

}

half4 LightingCustomLambert(SurfaceOutput s, half3 lightDir, half atten)

{

half NdotL = dot(s.Normal, lightDir);

half4 c;

c.rgb = s.Albedo \* \_LightColor0.rgb \* (NdotL \* atten);

c.a = s.Alpha;

return c;

}

void surf(Input IN, inout SurfaceOutput o)

{

fixed4 tex = tex2D(\_MainTex, IN.uv\_MainTex);

o.Albedo = tex.rgb;

o.Alpha = tex.a;

o.Normal = UnpackNormal(tex2D(\_BumpMap, IN.uv\_BumpMap));

}

ENDCG

}

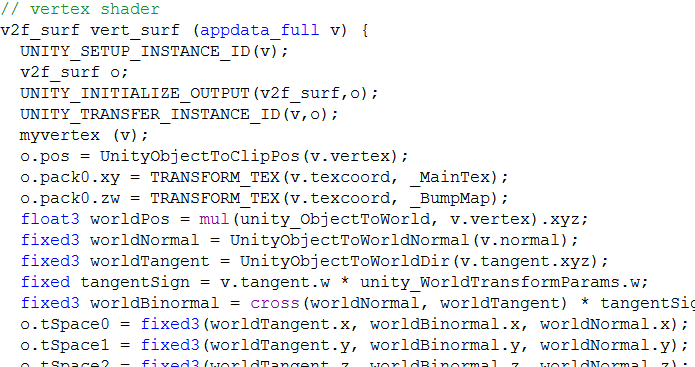
FallBack "Diffuse"

}



在上面的例子中，我们只是对顶点坐标进行了修改，而不需要向Input结构体添加并存储新的变量，也可以使用另一个版本的函数声明来把顶点修改函数中的某些计算结果保存到Input结构体中：

Void vert(input appdata\_full v, out Input o)



注意，定义的光照函数名字在前面添加一个Lighting字符串

**Surface Shader的缺点**

表面着色器只是unity在顶点/片元着色器上面提供的一种封装，是一种更高层的抽象。但任何在表面着色器中完成的事情，我们都可以在顶点/片元着色器中重现，但不幸的是，这句话反过来并不成立。表面着色器虽然可以快速实现各种光照效果，但我们失去了对各种优化和各种特效实现的控制。因此，使用表面着色器往往会对性能造成一定的影响，而内置的Shader，例如Diffuse、BumpedSpecular等都是使用表面着色器编写的。尽管Unity提供了移动平台的相应版本，例如Mobile/Diffse和Modiel/Bumped Specular等，但这些版本的Shader往往只是去掉了额外的逐像素Pass、不计算全局光照和其他一些光照计算上的优化。但要想进行更多深层的优化，表面着色器就不能满足我们的需求了。