# 着色器（Shader）

## Unity着色器简介

在Unity中，着色器（Shader）有三种实现方式：

1. 表面着色器（Surface shader）
2. 顶点着色器和片段着色器（Vetex shader & Fragment shader）
3. 固定着色器（Fixed function shader）

其中Unity推荐开发者重点学习和使用表面着色器，简化特殊效果的实现。在本章后面的章节里，会介绍它们的区别，现在先来介绍着色器的整体概念。

无论选择哪一种着色器的实现方式，在Unity中，shader代码最终都以一种特定的语法来编写，称为ShaderLab，它被用来组织shader的结构，看起来像这样：

|  |
| --- |
| Shader "MyShader" {  Properties {  \_MyTexture ("My Texture", 2D) = "white" { }  // 其它属性定义  }  SubShader {  // 具体的代码写在这里  // 可以是surface shader、vertex shader或者fixed function shader  }  SubShader {  // here goes a simpler version of the SubShader  // above than can run on older graphics cards  // 也是写代码的位置  // 这里通常用来编写上面的shader代码的简化版本，  // 用以兼容较老旧的设备  }  } |

本章会介绍以下内容：

1. 编写表面着色器的基础知识。
2. ShaderLab的基本语法和关键字。
3. 材质（Material）、着色器（Shader）以及纹理（Texture）的概念。
4. 展示多个表面着色器的实例。

## 编写表面着色器

编写传统的着色器代码非常复杂，特别是要考虑到光照的时候。3D游戏中有着不同的光照算法、不同的阴影算法一级不同的渲染路径（前向渲染和延迟渲染），在编写着色器时，要考虑以上这些全部问题。

Unity的“表面着色器”实际上是一种代码生成的技术，Unity会将我们所编写的表面着色器代码重新转换为底层的顶点/片段着色器代码，这样就比直接编写常规的着色器代码要简单很多。但是要注意，这个转换过程并没有特别高深的智能处理，而只是一种基于规则的代码转换，所以我们仍然要学习和使用标准的HLSL。

这种转换方法虽然效果很好，但是在实际开发中也有一些问题，比如说很多shader代码的错误是基于转换后的代码的错误，而用户编写的是转换之前的代码，二者的不一致性可能给定位BUG带来难度。

### 简介

首先，我们要定义一个“surface函数”，输入模型UV或其它数据，然后用代码填写SurfaceOutput结构体作为输出结果。SurfaceOutput结构体描述了表面的基本信息，比如固有色、法线、发光、镜面反射等等。这里我们用到的语言是专门用于编写着色器代码的，叫做HLSL。

表面着色器会接收数据输入、填充输出信息，然后这段代码会被转换成实际的顶点/片段着色器代码，同时也会被划分到多个渲染过程（render passes）中去，它会考虑到前向渲染、延迟渲染等等实际情况。

标准输出结构体如下所示：

|  |
| --- |
| struct SurfaceOutput  {  fixed3 Albedo; // 固有色  fixed3 Normal; // 法线，以切线空间（tangent space）表示  fixed3 Emission;  half Specular; // 镜面反射指数，取值0~1  fixed Gloss; // 镜面反射强度  fixed Alpha; // 透明度  }; |

Unity也支持基于物理的光照模型（PBR），内置的两种标准着色器Standard和StandardSpecular使用以下这两种输出结构：

|  |
| --- |
| struct SurfaceOutputStandard  {  fixed3 Albedo; // 固有色  fixed3 Normal; // 法线，以切线空间（tangent space）表示  half3 Emission;  half Metallic; // 金属化程度0~1。0=非金属， 1=金属  half Smoothness; // 光滑度0~1。  half Occlusion; // 灰度，默认为1  fixed Alpha; // 透明度  };  struct SurfaceOutputStandardSpecular  {  fixed3 Albedo; // 固有色  fixed3 Specular; // 镜面反射颜色  fixed3 Normal; // 法线，以切线空间（tangent space）表示  half3 Emission; // 发光  half Smoothness; // 光滑度0~1  half Occlusion; // 灰度，默认为1  fixed Alpha; // 透明度  }; |

### 预处理指令

表面着色器的代码放在从CGPROGRAM开始到ENDCG结束的代码块中，和其它类型的着色器代码是一样的，不同之处在于：

1. 代码必须放在SubShader代码块中，而不是Pass代码块。表面着色器会在转换时，自动分解为多个Pass。
2. 可以使用#pragma surface .... 这样的预处理指令，指定shader相关信息。

预处理指令格式如下：

|  |
| --- |
| #pragma surface 方法名称 光照模型 [其它参数] |

#### 必要参数

* 方法名称：执行哪个Cg函数。该函数定义应当为void surf(Input IN, inout SurfaceOutput o)，其中Input是一个自定义的输入结构体类型，它应当包含所有必要的材质坐标和其它参数。
* 光照模型：指定光照模型，默认的光照模型有基于物理的Standard和StandardSpecular，还有其它的非物理简易模型Lambert、BlinnPhong。
* Standard光照模型使用SurfaceOutputStandard 输出结构体，与Standard着色器相对应。
* StandardSpecular 光照模型使用SurfaceOutputStandardSpecular 输出结构体，与StandardSpecular 着色器相对应。
* Lambert和BlinnPhong光照模型不是基于物理的光照模型，它们的像真度相对更低，执行效率更高，适合于一些特殊风格的游戏。

#### 可选参数

**透明度和alpha测试。**透明度与alpha测试受“alpha”和“alphatest”这两个参数影响。典型的透明效果有两种类型：传统的alpha混合（将透明物体与它背后的物体混合）；以及另一种更符合物理的方式，能够表现出半透明物体在透射的同时还有反射的效果，被称为“premultiplied混合”。依据是否开启半透明效果，表面着色器转换成的最终代码会有所不同。

* alpha或alpha:auto 自动选择alpha:fade或者alpha:premul，判定依据是使用简单光照还是PBR光照。
* alpha:blend 开启alpha混合。
* alpha:fade 开启传统的透明混合方式。
* alpha:premul 使用premultiplied混合方式。
* alphatest:变量名 开启alpha裁剪，裁剪参数为指定的float变量的值。可能需要配合使用addshadow参数来生成合适的阴影。
* keepalpha 默认表面着色器对不透明的点会在alpha通道内写入1.0，而不管返回值具体是多少。使用这个参数用于保留alpha通道的值。
* decal:add 附加的贴花着色器。通常是在物体表面上附加一层效果，例如油漆泼洒在箱子上。
* decal:blend 半透明贴花。类似decal:add，但是还要开启alpha混合。

**顶点和颜色调整。**还有一些参数用于调整顶点的计算，以及改变最终的像素颜色。

* vertex:函数名称 自定义附加的顶点调整函数，用于调整最终顶点的位置。
* finalcolor:函数名称 自定义最终颜色调整函数。
* finalgbuffer:函数名称 自定义deferred path，用于调整gbuffer。
* finalprepass:函数名称 自定义prepass base path。

**阴影与细分曲面（Tessellation）。**这一类参数用于控制阴影的计算以及细分曲面的处理。（细分曲面是一种自动优化3D模型顶点的技术）

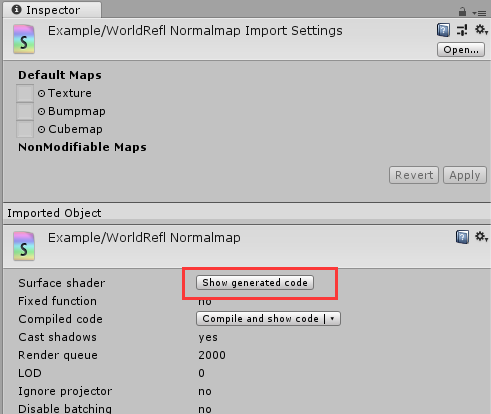
* addshadow 生成阴影pass。通常用于在修改了顶点之后，对阴影做出相应调整。
* fullforwardshadows 在前向渲染（forward rendering）时支持所有的光照阴影类型。默认的着色器在前向渲染时仅支持一个方向光源，当确实需要点光源和探照灯等光源时，使用这个参数。
* tesselate:TessFunction 指定细分曲面处理函数。（DX11支持GPU细分曲面技术）。

**代码生成。**默认表面着色器会尝试为所有可能的情景生成对应的代码。某些情况下不需要生成一部分代码（较小的shader加载较快），可以使用以下参数进行调整。

* exclude\_path:deferred, exclude\_path:forward, exclude\_path:prepass 三种参数的写法分别对应跳过deferred、forward或prepass代码的生成。
* noshadow 不生成阴影代码。
* noambient 不生成环境光与光照探针的代码。
* novertexlights 在前向渲染路径中不应用任何光照探针或逐顶点光照。
* nolightmap 在本着色器中禁用lightmapping。
* nodynlightmap  禁用实时全局光照。
* nodirlightmap 禁用directional lightmaps。
* nofog 禁用内置的雾效果。
* nometa 不生成meta pass。
* noforwardadd 禁用前向渲染的附加pass。
* nolppv 禁用Light Probe Proxy Volume。
* noshadowmask 禁用阴影遮罩（包含shadowmask和Distance Shadowmask）。

**其它选项。**包括softvegetation，interpolateview，halfasview，approxview，dualforward，dithercrossfade等。

以上选项会影响最终着色器代码的生成，若想查看具体生成的代码的变化，在Shader Inspector窗口中点击“Show Generated Code”就可以看到由表面着色器生成的最终着色器代码。



### 表面着色器的输入结构体

表面着色器的输入，可以选择任何必要的信息（贴图、坐标等等）。贴图坐标信息必须命名为uv+贴图名称。

以下参数可以添加到输入结构体中：

* float3 viewDir 摄像机视线向量。
* float4 with COLOR semantic 插值后的逐顶点颜色。
* float4 screenPos 屏幕坐标。
* float3 worldPos 世界坐标位置。
* float3 worldRefl 世界坐标系中的反射向量，在表面着色器未填写o.Normal时使用。
* float3 worldNormal 世界坐标系中的法线向量，在表面着色器未填写o.Normal时使用。
* float3 worldRefl; INTERNAL\_DATA 世界坐标系中的反射向量，在表面着色器已经填写o.Normal时使用。
* float3 worldNormal; INTERNAL\_DATA 世界坐标系中的法线向量，在表面着色器已经填写o.Normal时使用。

## ShaderLab简介

Unity中所有的Shader文件都是以一种描述性语言表示的，称为“ShaderLab”。这些文件用一些嵌套的大括号来声明各种各样的信息，例如哪些属性在检视窗口中显示，哪些内容用于兼容旧的设备，使用哪种混合模式等等。而实际的“shader代码”是包含在CGPROGRAM代码段中。

本节将介绍ShaderLab基本的语法和关键字。CGPROGRAM代码段以标准的HLSL/Cg语言编写，篇幅原因本节不做详细介绍。

关键字“Shader”在一个文件中只有一个，它定义了一个着色器。它还有一些参数用于指定该着色器的使用方式。

### 语法

|  |
| --- |
| Shader "name" { [Properties] Subshaders [Fallback] [CustomEditor] } |

上面的一行代码定义了一个着色器。该着色器的名称会显示到Unity材质的检视窗口中。Properties是一个属性的列表，该列表可以显示到Unity编辑器中。Properties之后有一个子着色器（Subshaders）列表，后面是可选的回滚（fallback）定义以及自定义的编辑器声明。

### 属性

着色器有一个属性列表properties，其中定义的所有属性都会在材质（Material）的检视窗口中显示出来。典型的属性包括物体颜色、贴图或者简单的运算用的数值。

### 子着色器与回滚

每个着色器都是由一串“子着色器”组成的，至少有一个子着色器。当加载shader时，unity会按顺序遍历所有的子着色器，选择第一个能够和用户硬件匹配的子着色器。如果所有子着色器都无法匹配，Unity会尝试使用后备的回滚着色器。

不同的显卡有着不同的特性，这对游戏开发者们来说是一个长久的问题。我们总是希望能够利用到最新的硬件技术，但是可能只有3%的设备支持这种技术。因此子着色器的必要性就显而易见了：先为最新的硬件实现一个梦幻级别的效果，然后再添加几个子着色器专门支持其它较老旧的显卡。支持旧设备的着色器可以通过简化一些细节的实现，来增强兼容性。

LOD（level of detail，在不同的距离下显示不同的细节，近处看细节更多，远处看细节更少），以及着色器替换（shader replacement）也会用到子着色器的技术。

### 例子

这是一个简单表面着色器的例子：

|  |
| --- |
| // 带颜色的顶点光照  Shader "Simple colored lighting"  {  // 只有一个颜色属性  Properties {  \_Color ("Main Color", Color) = (1,.5,.5,1)  }  // 定义一个子着色器  SubShader  {  // 定义一个pass  Pass  {  // 指定顶点的颜色  Material  {  Diffuse [\_Color]  }  Lighting On  }  }  } |

这个着色器定义了一个颜色属性\_Color，默认值为(1, 0.5, 0.5, 1)。还定义了一个子着色器，子着色器包含了pass，最终是使用固定的方法指定了顶点颜色。

本章的最后会介绍一系列更实际的例子，让读者对shader有一个全面的认识。

## 材质，着色器与贴图的关系

Unity中和渲染有关的基本概念包括材质（Material）、着色器（Shader）和贴图（Texture），这三者有着非常紧密的联系。

材质定义了物体的表面如何被渲染，材质引用了要使用的贴图、拼贴信息、色调信息等等。材质的选项和参数，取决于定义材质的着色器需要哪些参数。

材质球文件示例：



与上面的材质球对应的贴图文件：



材质球还包含了着色器：



前文介绍过，着色器可以看成是小的GPU脚本，通过数学计算和算法来计算每个像素的颜色，这种算法是基于光照系统与材质参数设置的。

贴图就是由像素组成的图片，材质一般都包含了对贴图的引用，而着色器在计算像素颜色就会用到这个贴图的信息，注意，贴图不仅仅是直觉上的物体外表图案和颜色，贴图还可以用来表示任何其它表面信息，例如物体的粗糙程度、表面凹凸、甚至顶点位置也可以用贴图来改变，一切都可以由着色器代码控制。

材质关联了一个着色器，着色器定义了该材质的表现，以及该材质可以定义哪些参数。着色器定义了一个或多个需要用到的贴图，在Unity编辑器中，选中材质后可以在检视窗口中看到所有的参数，并为需要贴图的参数指定贴图。

Unity默认的Standard Shader足够用于大部分常规游戏的渲染，Standard Shader是基于物理的（PBR），它有很多参数可以调节，用来实现各种逼真的物体外观。

除Standard Shader之外，还有各种不同的内置着色器，可以在编辑器中随意改变这些着色器，看看截然不同的效果。不同的着色器适用于不同的场景，例如液体、树叶、半透明玻璃、粒子特效、卡通效果或者其它各种独特的美术效果；甚至还可以实现例如X射线、红外线、夜视仪效果。

## 表面着色器实例

本节将通过一系列的表面着色器实例，来演示说明表面着色器的使用方法。相信通过实验，读者可以很快对表面着色器的使用有一个全面的认识，且可以灵活运动到实际游戏开发中去。

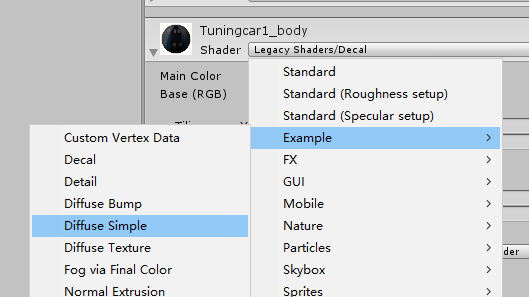
以下实例都使用了内置的光照模型，实际上还可以自定义其它的光照模型，也是用表面着色器实现的。

### 从最简单的开始

我们从一个非常简单的例子开始，以下着色器代码只是将颜色设置为白色（white），且它使用了内置的Lambert（漫反射）光照模型。

|  |
| --- |
| Shader "Example/Diffuse Simple" {  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float4 color : COLOR;  };  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = 1;  }  ENDCG  }  Fallback "Diffuse"  } |

将以上代码写入一个扩展名为shader的文本文件，并放在工程中之后，Unity就会自动编译这个shader脚本。如果没有错误，那么在材质属性那里，就可以选中该shader：

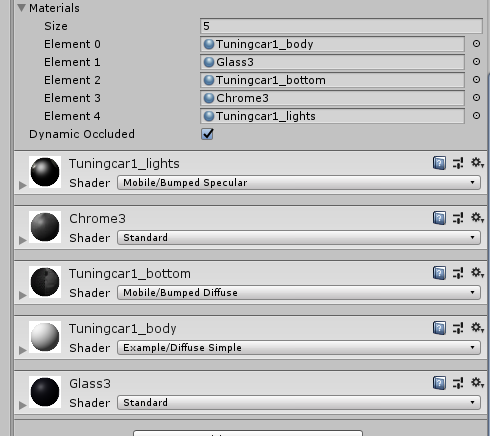


注意，shader的名称就是上面代码中第一行指定的Shader "Example/Diffuse Simple" 。

下图是渲染效果：



读者会发现，我们所用的模型，车轮、窗户、车灯等部分都没有变成白色，确实如此。这个模型可以配置5个不同部分的材质，每个材质都可以有独立的Shader。



### 贴图

纯白的模型没有什么用处，接下来我们添加上贴图。关键是在Properties块中添加一个属性“\_MainTex”，这个属性可以在Unity的检视窗口中看到，可以任意指定贴图。完整的着色器代码如下：

|  |
| --- |
| Shader "Example/Diffuse Texture" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  };  sampler2D \_MainTex;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  }  ENDCG  }  Fallback "Diffuse"  } |

这个例子使用了内置的tex2D函数在贴图上进行采样。采样出来的是一个像素颜色值，而且采样不仅需要贴图，还需要模型本身的uv信息（uv决定了模型顶点和贴图之间的对应关系）。效果如图：



### 法线贴图

法线贴图是一种为模型增加细节的实用方法，例如毛巾表面的凹凸、木材表面小的凹陷都可以用法线贴图表现。且法线贴图不会真的增加模型的顶点数和面熟，只是实现类似的效果，性价比非常高。

表示法线的“贴图”本身是一种特殊的贴图，通常用贴图上面明暗的变化来表示哪一块是凸起、哪一块是凹陷。法线贴图可以在制作3D模型时一并制作好，也可以用自动化工具生成。一个好的法线贴图可以为模型增色不少。

|  |
| --- |
| Shader "Example/Diffuse Bump" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_BumpMap ("Bumpmap", 2D) = "bump" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  float2 uv\_BumpMap;  };  sampler2D \_MainTex;  sampler2D \_BumpMap;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  o.Normal = UnpackNormal (tex2D (\_BumpMap, IN.uv\_BumpMap));  }  ENDCG  }  Fallback "Diffuse"  } |

以上代码中的\_BumpMap即为法线贴图，在surf函数中，使用内置的UnpackNormal方法将贴图像素转换为法线数据，即可实现很好的法线效果。

下图是增加法线的效果，这个法线贴图的效果并不明显，但是在细节处侧面的凹陷感还是非常明显。在模型没有任何改变的情况下添加了一些细节，这就是法线贴图的作用。但是要记得事先准备好一张法线贴图。

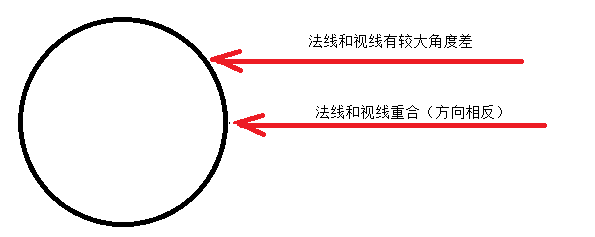


### 边缘发光

现在我们再添加边缘发光效果。

边缘发光是现实中明显存在的一种现象，特别是在阳光充足的户外，可以观察到物体的边缘更明亮，在游戏中我们不需要通过真实的光线计算达到这个效果，而只需要用一个巧妙的算法找出哪些地方是“边缘”，哪些地方不是，然后只对更加边缘的位置增加更多亮度。

这个原理非常简单，就是考虑摄像机的视线，与物体表面法线的夹角。



|  |
| --- |
| Shader "Example/Rim" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_BumpMap ("Bumpmap", 2D) = "bump" {}  \_RimColor ("Rim Color", Color) = (0.26,0.19,0.16,0.0)  \_RimPower ("Rim Power", Range(0.5,8.0)) = 3.0  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  float2 uv\_BumpMap;  float3 viewDir;  };  sampler2D \_MainTex;  sampler2D \_BumpMap;  float4 \_RimColor;  float \_RimPower;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  o.Normal = UnpackNormal (tex2D (\_BumpMap, IN.uv\_BumpMap));  half rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));  o.Emission = \_RimColor.rgb \* pow (rim, \_RimPower);  }  ENDCG  }  Fallback "Diffuse"  } |

上面的代码中，RimColor是发光颜色，RimPower是发光强度。通过将视线向量IN.ViewDir和法线向量进行一些运算后，转换为一个半浮点数值rim，然后通过rim和参数\_Rimpower的幂运算得到发光的量，赋值给o.Emission，即可得到正确的效果。



### 细节贴图

接下来演示如何在原有贴图的基础上覆盖一层表现细节的贴图。细节贴图用到的uv和模型贴图uv基本是一致的，但是Tiling略有不同，所以还是要单独指定一个uv\_Detail专门用于细节贴图。

|  |
| --- |
| Shader "Example/Detail" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_BumpMap ("Bumpmap", 2D) = "bump" {}  \_Detail ("Detail", 2D) = "gray" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  float2 uv\_BumpMap;  float2 uv\_Detail;  };  sampler2D \_MainTex;  sampler2D \_BumpMap;  sampler2D \_Detail;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  o.Albedo \*= tex2D (\_Detail, IN.uv\_Detail).rgb \* 2;  o.Normal = UnpackNormal (tex2D (\_BumpMap, IN.uv\_BumpMap));  }  ENDCG  }  Fallback "Diffuse"  } |

可以看到，细节贴图的采样以乘法叠加到了o.Albedo上，也就是说乘法可以用于颜色的混合，且可以通过系数控制混合的比例。我将一个较杂乱的贴图直接贴在车身上，效果如下：



这种做法在实际游戏中不多见，但是用于学习和解释多层贴图的方法还是很有意义的。

### 屏幕空间中的细节贴图

接下来是一个特殊的细节贴图演示，主要用来展示屏幕空间和模型uv空间不同的计算方法：

|  |
| --- |
| Shader "Example/ScreenPos" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_Detail ("Detail", 2D) = "gray" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  float4 screenPos;  };  sampler2D \_MainTex;  sampler2D \_Detail;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  float2 screenUV = IN.screenPos.xy / IN.screenPos.w;  screenUV \*= float2(8,6);  o.Albedo \*= tex2D (\_Detail, screenUV).rgb \* 2;  }  ENDCG  }  Fallback "Diffuse"  } |

以上代码移除了法线贴图，以便突出重点。可以看出，上面的代码用screenPos.xy和screenPos.w属性构造出了一个屏幕空间的UV，然后用这个UV进行贴图采样，得到了一个和模型无关，只和屏幕坐标有关的效果：



### 立方体（Cubemap）反射

这个例子采用了内置的worldRefl世界空间反射输入参数，来让环境中的光线和颜色改变模型的发光值，这样就实现了一种金属反射的效果：

|  |
| --- |
| Shader "Example/WorldRefl" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_Cube ("Cubemap", CUBE) = "" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  float3 worldRefl;  };  sampler2D \_MainTex;  samplerCUBE \_Cube;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb \* 0.5;  o.Emission = texCUBE (\_Cube, IN.worldRefl).rgb;  }  ENDCG  }  Fallback "Diffuse"  } |

代码中，\_Cube属性的使用是一个重点，简单说来，就是假设将模型放在了一个立方体中，然后让立方体内表面的颜色映照在模型表面上，这样就好像模型反射了周围环境的光线，实现了金属反射效果。立方体贴图可以在编辑器中指定，它也可以用来实现天空盒。



上图中，模型的凹凸并不明显，也可以将法线贴图和反光一并考虑，实现细节处的反射。

|  |
| --- |
| Shader "Example/WorldRefl Normalmap" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_BumpMap ("Bumpmap", 2D) = "bump" {}  \_Cube ("Cubemap", CUBE) = "" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  float2 uv\_BumpMap;  float3 worldRefl;  INTERNAL\_DATA  };  sampler2D \_MainTex;  sampler2D \_BumpMap;  samplerCUBE \_Cube;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb \* 0.5;  o.Normal = UnpackNormal (tex2D (\_BumpMap, IN.uv\_BumpMap));  o.Emission = texCUBE (\_Cube, WorldReflectionVector (IN, o.Normal)).rgb;  }  ENDCG  }  Fallback "Diffuse"  } |

以上代码的关键点：在输入结构体中添加INTERNAL\_DATA，然后先填写法线信息o.Normal，再利用WorldReflectionVector函数，并传入输入结构体IN，这样就得到了考虑过法线信息的采样结果。效果如下，细节明显变多了，特别是引擎盖的凹凸：



### 世界空间切片

以下实例对模型进行了空间中的“切片”，实际上是通过不渲染某些像素实现的。

|  |
| --- |
| Shader "Example/Slices" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_BumpMap ("Bumpmap", 2D) = "bump" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  Cull Off  CGPROGRAM  #pragma surface surf Lambert  struct Input {  float2 uv\_MainTex;  float2 uv\_BumpMap;  float3 worldPos;  };  sampler2D \_MainTex;  sampler2D \_BumpMap;  void surf (Input IN, inout SurfaceOutput o) {  clip (frac((IN.worldPos.y+IN.worldPos.z\*0.1) \* 5) - 0.5);  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  o.Normal = UnpackNormal (tex2D (\_BumpMap, IN.uv\_BumpMap));  }  ENDCG  }  Fallback "Diffuse"  } |

可以看到，用IN.worldPos属性可以得到一个点在世界空间中的坐标，然后利用内置的clip函数取消掉某些点的渲染。这样就实现了切片效果：



### 修改顶点位置

表面着色器不仅可以用来对贴图进行计算，甚至还可以直接改变顶点的位置。以下的例子很有意思：让每个顶点朝着自己的法线方向向外移动，就可以让模型“变胖”。

|  |
| --- |
| Shader "Example/Normal Extrusion" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_Amount ("Extrusion Amount", Range(-1,1)) = 0.5  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert vertex:vert  struct Input {  float2 uv\_MainTex;  };  float \_Amount;  void vert (inout appdata\_full v) {  v.vertex.xyz += v.normal \* \_Amount;  }  sampler2D \_MainTex;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  }  ENDCG  }  Fallback "Diffuse"  } |

注意到，以上代码的这一句：

#pragma surface surf Lambert vertex:vert

指定了表面渲染的函数surf以及顶点处理函数vert，vert函数作为顶点处理函数，用到的参数和计算方式又和surf函数有很多不同。通过在编辑器中调整Amount的值，可以让模型“变胖”或“变瘦”。

对我们的汽车模型来说，变胖意味着解体：



出现“解体”的原因和这个模型有关，这个模型的引擎盖、侧面是分离的面组成的，所以改变顶点位置以后会出现不连接的情况，一般的角色模型没有这个问题。

### 逐顶点数据处理

顶点处理函数不仅可以修改顶点的位置，甚至可以在处理顶点时附加一些其它信息。这些信息在渲染的流程中，会传递给后续的片段着色阶段，也就是说可以在表面着色函数中获取到。利用顶点信息可以实现更多可能的效果。

|  |
| --- |
| Shader "Example/Custom Vertex Data" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert vertex:vert  struct Input {  float2 uv\_MainTex;  float3 customColor;  };  void vert (inout appdata\_full v, out Input o) {  UNITY\_INITIALIZE\_OUTPUT(Input,o);  o.customColor = abs(v.normal);  }  sampler2D \_MainTex;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  o.Albedo \*= IN.customColor;  }  ENDCG  }  Fallback "Diffuse"  } |

这段代码中，顶点处理函数依然是vert，但是它又添加了一个参数out Input o，其中Input是一直以来用到的输入结构体，在其中我们又添加了一个customColor属性让信息从vert传递到surf。注意这里的customColor命名虽然可以任意，但是规定绝对不能以uv开头，否则会带来问题。

这个例子直接将顶点法线转换为颜色，然后得到了一个特殊的效果：



这个例子不太实际，更切合实际的用法请参考前面“边缘发光”的例子。当时我们计算了每一个像素的法线和视线的夹角关系，计算量很大，实际上完全可以用计算顶点法线代替计算像素法线，这样计算次数就能以级数减小，且效果相差并不大。

### 最终颜色调整

可以设计一个“最终颜色调整”函数用来在渲染的最后步骤改变颜色值。表面着色器提供了finalcolor:变量名的参数来添加一个最终执行函数，它的参数是Input IN，SurfaceOutput o和inout fixed4 color。

|  |
| --- |
| Shader "Example/Tint Final Color" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_ColorTint ("Tint", Color) = (1.0, 0.6, 0.6, 1.0)  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert finalcolor:mycolor  struct Input {  float2 uv\_MainTex;  };  fixed4 \_ColorTint;  void mycolor (Input IN, SurfaceOutput o, inout fixed4 color)  {  color \*= \_ColorTint;  }  sampler2D \_MainTex;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  }  ENDCG  }  Fallback "Diffuse"  } |

这个例子添加了一个mycolor函数作为最终执行函数，mycolor里面只有一行代码，让输入的\_ColorTint直接和当前颜色color做乘法。注意：这里得到的color参数，实际上已经被光照贴图、光照探针以及其它因素影响过了，所以说这里的调整在其它步骤之后，是一个“最终调整”。下图颜色会整体偏向红色：



### 雾

上面所讲的最终颜色调整，常见用途是用来在前向渲染中实现自定义的雾效果。雾会影响最终的像素颜色，正好适用最终颜色调整的方法。

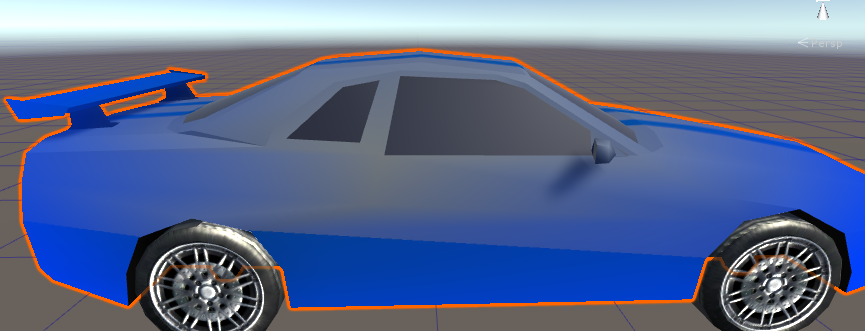
|  |
| --- |
| Shader "Example/Fog via Final Color" {  Properties {  \_MainTex ("Texture", 2D) = "white" {}  \_FogColor ("Fog Color", Color) = (0.3, 0.4, 0.7, 1.0)  }  SubShader {  Tags { "RenderType" = "Opaque" }  CGPROGRAM  #pragma surface surf Lambert finalcolor:mycolor vertex:myvert  struct Input {  float2 uv\_MainTex;  half fog;  };  void myvert (inout appdata\_full v, out Input data)  {  UNITY\_INITIALIZE\_OUTPUT(Input,data);  float4 hpos = UnityObjectToClipPos(v.vertex);  hpos.xy/=hpos.w;  data.fog = min (1, dot (hpos.xy, hpos.xy)\*0.5);  }  fixed4 \_FogColor;  void mycolor (Input IN, SurfaceOutput o, inout fixed4 color)  {  fixed3 fogColor = \_FogColor.rgb;  #ifdef UNITY\_PASS\_FORWARDADD  fogColor = 0;  #endif  color.rgb = lerp (color.rgb, fogColor, IN.fog);  }  sampler2D \_MainTex;  void surf (Input IN, inout SurfaceOutput o) {  o.Albedo = tex2D (\_MainTex, IN.uv\_MainTex).rgb;  }  ENDCG  }  Fallback "Diffuse"  } |

这段代码实现了从屏幕四周到中央衰减的雾。它的算法原理是在计算顶点时，计算该顶点离屏幕中央的距离，处理后记录到fog变量中。在最后的mycolor处理函数中通过IN.fog可以访问到之前保存的fog值，依据fog进行颜色调整。

注意到，在前向渲染中，雾的颜色只能是黑色，所以这段代码里用

#ifdef UNITY\_PASS\_FORWARDADD

对当前的渲染方式进行了检查和处理。



### 示例总结

从以上的多个例子可以看出，着色器核心算法的实现，离不开3D图形学以及数学技巧两大部分。我们既可以选择认真学习3D图形学和数学，从头打造一个独特的着色器；也可以选择在现有的着色器范例上进行微调，实现适合于特定游戏的着色器。着色器的编写方法博大精深，且新的更快、更逼真的算法依然在发展之中，新的硬件与软件为游戏画面表现力的提升带来了更多可能。