

Unity Shader_4-----三类着色器的对比

主要参考：《Unity shader 入门精要 著冯乐乐》

作者博客 <http://blog.csdn.net/candycat1992/article/details/50167285>

浅墨博客 http://blog.csdn.net/poem_qianmo/article/details/49556461

风宇冲博客 http://blog.sina.com.cn/s/blog_471132920101d5kh.html

潜水的小懒猫博客 http://blog.sina.com.cn/s/articlelist_2312702844_6_1.html

猫都能学会的Unity3D Shader <https://onevc.com/2013/07/shader-tutorial-1/>
<https://onevc.com/2013/08/shader-tutorial-2/>

蛮牛教育Shader编程教程 <http://edu.manew.com/course/96>

Unity官方; <http://docs.unity3d.com/Manual/SL-ShaderPrograms.html>

一、三类着色器的对比

- 固定功能着色器（FixedFunction Shader） -----固定功能管线
- 表面着色器（SurfaceShader） -----可编程渲染管线
- 顶点着色器 & 片段着色器（Vertex Shader & Fragment Shader） -----可编程渲染管线

就FixedFunction Shader而言：

只需要使用一些关键字按照一定编写规范，就能实现很多效果；其在Unity底层进行了大量工作。其主要目的是为了兼容一些老式显卡，主要实现方式为**顶点光照**，其**核心模块**是位于SubShader{}下的**Material{}**块中，注意一些效果实现（如玻璃，植被）

1. Shader "浅墨Shader编程/Volume5/固定功能的Shader示例"

```
2. {
3.     //----- 【属性】 -----
4.     Properties
5.     {
6.         _Color ("主颜色", Color) = (1,1,1,0)
7.         _SpecColor ("高光颜色", Color) = (1,1,1,1)
8.         _Emission ("自发光颜色", Color) = (0,0,0,0)
9.         _Shininess ("光泽度", Range (0.01, 1)) = 0.7
10.        _MainTex ("基本纹理", 2D) = "white" {}
11.    }
12.
13.    //----- 【子着色器】 -----
14.    SubShader
15.    {
16.        //----- 通道 -----
17.        Pass
18.        {
19.            //----- 材质 -----
20.            Material
21.            {
22.                //可调节的漫反射光和环境光反射颜色
23.                Diffuse [_Color]
24.                Ambient [_Color]
25.                //光泽度
26.                Shininess [_Shininess]
27.                //高光颜色
28.                Specular [_SpecColor]
```

```

29.         //自发光颜色
30.         Emission [_Emission]
31.     }
32.     //开启光照
33.     Lighting On
34.     //开启独立镜面反射
35. //*****让顶点光照和纹理颜色可以结合*****
36.     SeparateSpecular On
37.     //设置纹理并进行纹理混合
38.     SetTexture [_MainTex]
39.     {
40. //*****2 倍 RGB 和 Alpha *****
41.         Combine texture * primary DOUBLE, texture * primary
42.     }
43. }
44. }
45. }

```

植被效果：

使用两个Pass通道的原因：

渲染树和植物时，透明度测试会出现尖锐的边缘，解决方法之一：渲染对象两次（首次通道中只渲染超过50%透明度的像素，第二次通道中将透明度和上次剔除部分混合，且不记录像素深度（因为可能需要使一些树枝类的覆盖近的其他树枝））

Shader "摘自浅墨博客"

```

{
    //----- 【属性】 -----

    Properties
    {
        _Color ("主颜色", Color) = (.5, .5, .5, .5)
        _MainTex ("基础纹理 (RGB)-透明度(A)", 2D) = "white" {}
        _Cutoff ("Alpha透明度阈值", Range (0,.9)) = .5
    }

    //----- 【子着色器】 -----

    SubShader
    {
        // 【1】 定义材质
        Material
        {
            Diffuse [_Color]
            Ambient [_Color]
        }

        // 【2】 开启光照
        Lighting On

        // 【3】 关闭裁剪，渲染所有面，用于接下来渲染几何体的两面
    }
}

```

Cull Off

```
//----- 【通道一】 -----  
//      说明：渲染所有超过[_Cutoff] 不透明的像素  
//-----  
Pass  
{  
AlphaTest Greater [_Cutoff]  
SetTexture [_MainTex] {  
    combine texture * primary, texture  
}  
}  
  
//----- 【通道二】 -----  
//      说明：渲染半透明的细节  
//-----  
Pass  
{  
    // 不写到深度缓冲中  
ZWrite off  
  
    // 不写已经写过的像素  
ZTest Less  
  
    // 深度测试中，只渲染小于或等于的像素值  
AlphaTest LEqual [_Cutoff]  
  
    // 设置透明度混合  
Blend SrcAlpha OneMinusSrcAlpha  
  
    // 进行纹理混合  
SetTexture [_MainTex]  
    {  
        combine texture * primary, texture  
    }  
}  
}  
}
```

Surface Shader：可以看做是一个表面光照Shader的语法块、一个光照顶点着色器和片元着色器的生成器；其中不包括Pass通道，其主要代码均包括在CGPROGRAM和ENDCG块中，（根据代码会自动会编译成多个

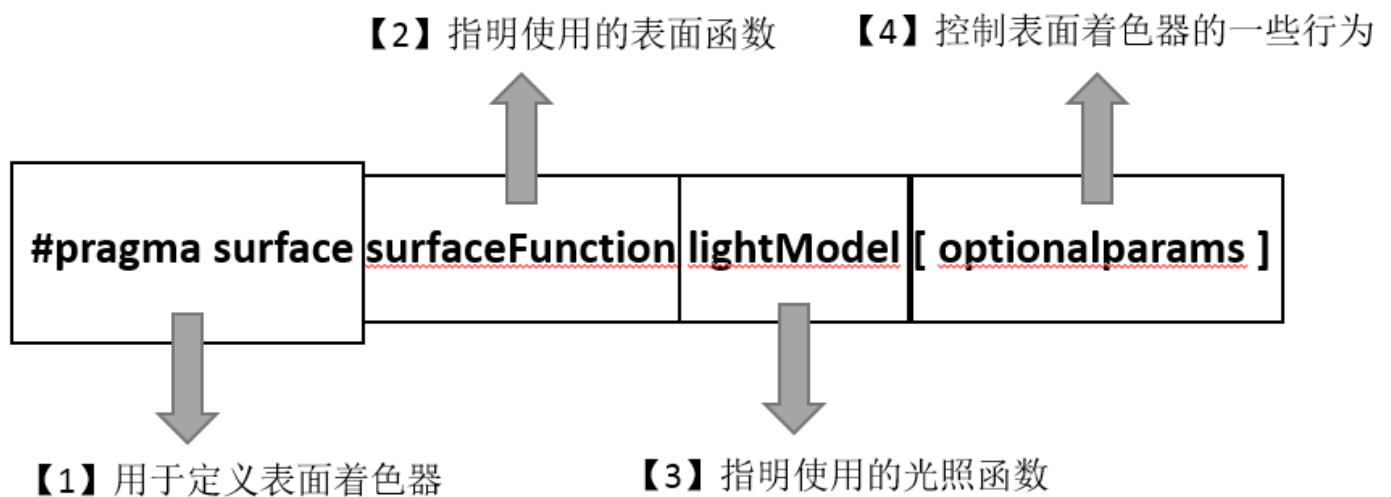
Pass{}))

严格上说，表面着色器本质是顶点着色器和片元着色器，所以本质来说，Unity Shader只有两种形式，顶点/片元着色器和固定功能（函数）着色器

在表面着色器和顶点&片元着色器中都存在编译指令

编译指令：

一般在Surface Shader中的格式：



【1】：相类似的，顶点&片元着色器也有相应的函数定义，一般格式为：

`#pragma vertex vert`（vert即为定义的顶点着色器函数名）

`#pragma fragment frag`（frag即为定义的片元着色器函数名）

【2】：表面着色器相较于顶点&片元着色器，将“表面”这一概念抽离出来，“表面”包含了对象的反射率、光滑度、透明度等等值；

类似 **【1】** 中所说 通常将其命名成“surf”（当然也可以是其他的），在函数体内可以定义这些表面属性以实现不同效果，函数格式是固定的，如下：

```
void surf (Input IN, inout SurfaceOutput o) { }
```

```
void surf (Input IN, inout SurfaceOutputStandard o) { }
```

```
void surf (Input IN, inout SurfaceOutputStandardSpecular o) { }
```

两个结构体：

• 输出：Input

结构内容：

- float3 viewDir - 视图方向(view direction)值。为了计算视差效果(Parallax effects)，边缘光照(rim lighting)等，需要包含视图方向(view direction)值。
- float4 with COLOR semantic -每个顶点(per-vertex)颜色的插值。
- float4 screenPos - 屏幕空间中的位置。为了反射效果，需要包含屏幕空间中的位置信息。比如在Dark Unity中所使用的 WetStreet着色器。
- float3 worldPos - 世界空间中的位置。
- float3 worldRefl - 世界空间中的反射向量。如果表面着色器(surface shader)不写入法线(o.Normal)参数，将包含这个参数。请参考这个例子：Reflect-Diffuse 着色器。

- float3 worldNormal - 世界空间中的法线向量(normal vector)。如果表面着色器(surface shader)不写入法线(o.Normal)参数, 将包含这个参数。
- float3 worldRefl; INTERNAL_DATA - 世界空间中的反射向量。如果表面着色器(surface shader)不写入法线(o.Normal)参数, 将包含这个参数。为了获得基于每个顶点法线贴图(per-pixel normal map)的反射向量(reflection vector)需要使用世界反射向量(WorldReflectionVector (IN, o.Normal))。请参考这个例子: Reflect-Bumped着色器。
- float3 worldNormal; INTERNAL_DATA -世界空间中的法线向量(normal vector)。如果表面着色器(surface shader)不写入法线(o.Normal)参数, 将包含这个参数。为了获得基于每个顶点法线贴图(per-pixel normal map)的法线向量(normal vector)需要使用世界法线向量(WorldNormalVector (IN, o.Normal))。

其中的Input结构体, 如上面三个函数中的Input结构体

常用结构内容如下:

```
struct
```

```
{
```

形式必须为 uv (小写) 加上 面板属性名 (根据命名写全名, 有"_"不要忘写"_")

```
uv_MainTex,
```

//也可设置为uv2表示次级纹理, 其中_MainTex为属性 (properties中定义的名称))

```
uv_BumpTex
```

```
uv_Detail
```

```
...
```

```
float3 viewDir, //视角方向, 常用于计算边缘光照
```

```
...还有一些Input的内置结构变量
```

语义:

float4 color : COLOR (在顶点&片元着色器中的结构体定义中 使用较多)

//使用COLOR语义, 作为从Vertex阶段到Fragment阶段的输入输出的约定

其他语义还有

SV_POSITION

POSITION

关于上面两者的区别: http://blog.csdn.net/zhao_92221/article/details/46797969

http://forum.unity3d.com/threads/what-is-the-difference-between-float4-pos-sv_position-and-float4-pos.165351/

NORMAL

TEXCOORD[n] : 如TEXCOORD1

具体可见HLSL语义库 (不仅仅只是Unity支持的语义) :

<https://msdn.microsoft.com/en-us/library/windows/desktop/bb509647.aspx>

```
};
```

• **输入: SurfaceOutput SurfaceOutputStandard SurfaceOutputStandardSpecular**

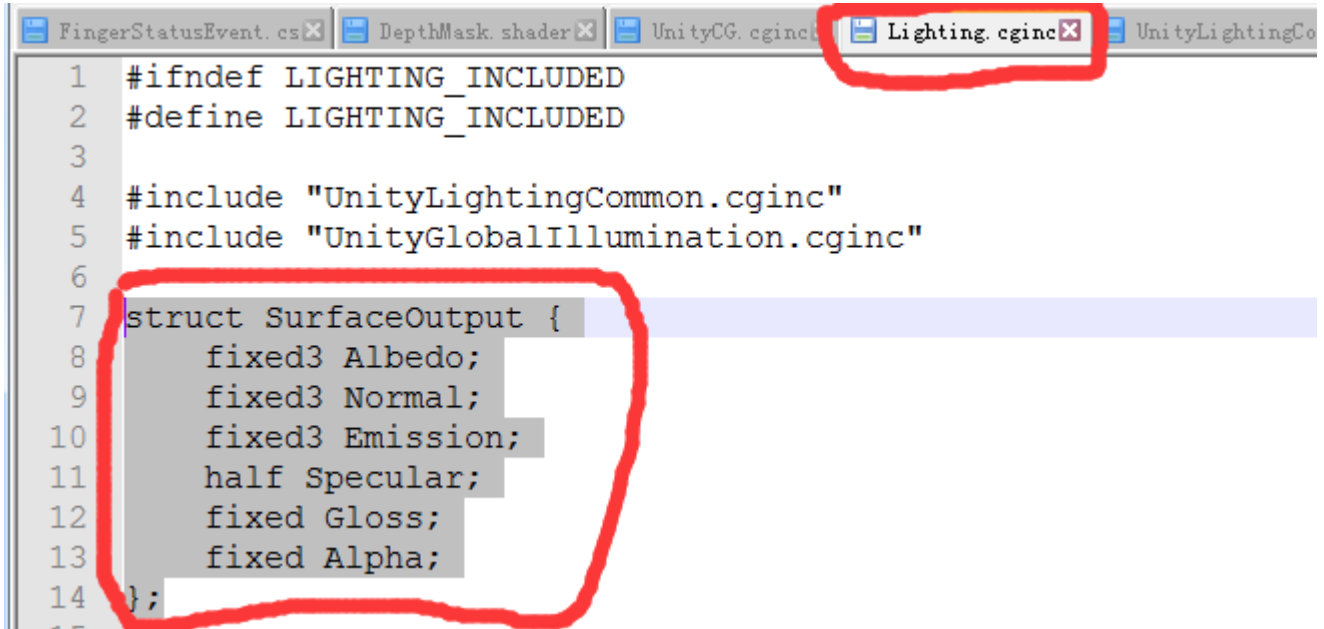
其中SurfaceOutput SurfaceOutputStandard SurfaceOutputStandardSpecular 为Unity内置的结构体 (inout表示可作为输入输出), 可以在光照模型扩展文件找到:

SurfaceOutput

1. struct SurfaceOutput

```
2. {
3.     half3 Albedo; // 纹理颜色值 (r, g, b), 反射率
4.     half3 Normal; // 法向量(x, y, z) , 法线
5.     half3 Emission; // 自发光颜色值(r, g, b)
6.     half Specular; // 镜面反射度
7.     half Gloss; // 光泽度
8.     half Alpha; // Alpha不透明度
9. };
```

查找路径:



```
1 #ifndef LIGHTING_INCLUDED
2 #define LIGHTING_INCLUDED
3
4 #include "UnityLightingCommon.cginc"
5 #include "UnityGlobalIllumination.cginc"
6
7 struct SurfaceOutput {
8     fixed3 Albedo;
9     fixed3 Normal;
10    fixed3 Emission;
11    half Specular;
12    fixed Gloss;
13    fixed Alpha;
14 };
15
```

像反射率的赋值:

1. //表面着色函数的编写

2. void surf (Input IN, inout SurfaceOutput o)

3. {

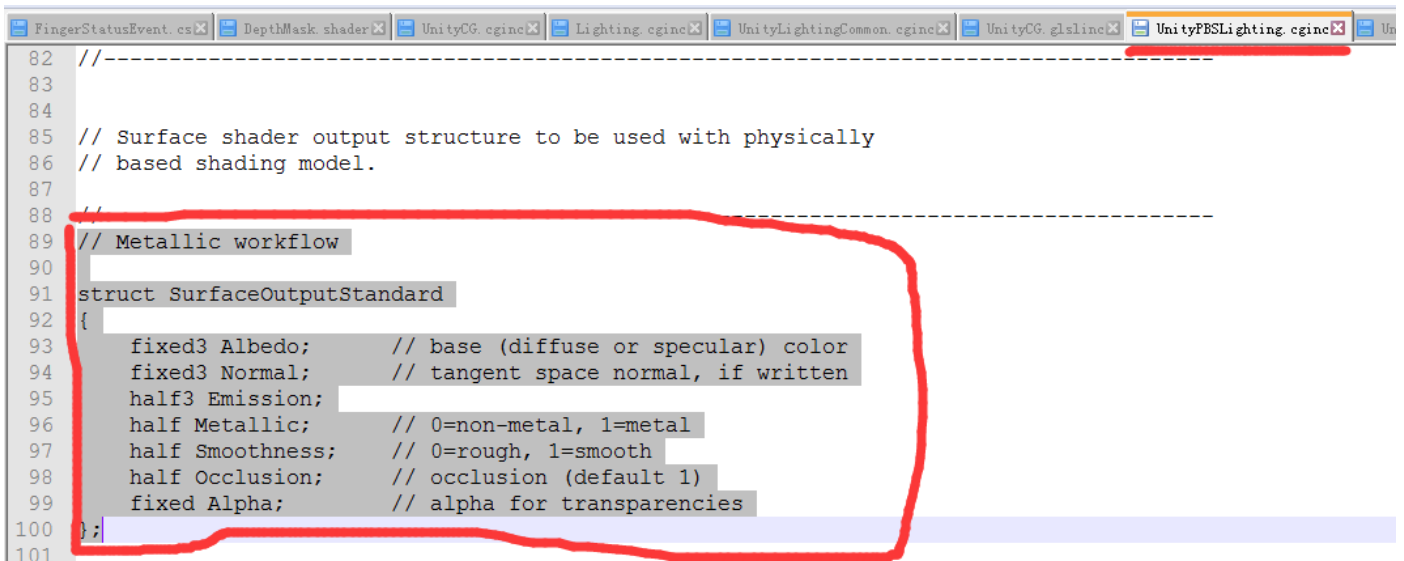
4. //反射率, 也就是纹理颜色值赋为(0.6, 0.6, 0.6)

5. o.Albedo= 0.6;

6. }

SurfaceOutputStandard || SurfaceOutputStandardSpecular

查找路径: Unity 5.x新添加的



```
82 //-----
83
84
85 // Surface shader output structure to be used with physically
86 // based shading model.
87
88 //-----
89 // Metallic workflow
90
91 struct SurfaceOutputStandard
92 {
93     fixed3 Albedo; // base (diffuse or specular) color
94     fixed3 Normal; // tangent space normal, if written
95     half3 Emission;
96     half Metallic; // 0=non-metal, 1=metal
97     half Smoothness; // 0=rough, 1=smooth
98     half Occlusion; // occlusion (default 1)
99     fixed Alpha; // alpha for transparencies
100 };
101
```

```
FingerStatusEvent.cs DepthMask.shader UnityCG.cginc Lighting.cginc UnityLightingCommon.cginc UnityCG.glslinc UnityPBSLighting.cginc
139 UnityGIInput data,
140 inout UnityGI gi)
141 {
142     UNITY_GI(gi, s, data);
143 }
144
145 //-----
146 // Specular workflow
147
148 struct SurfaceOutputStandardSpecular
149 {
150     fixed3 Albedo;        // diffuse color
151     fixed3 Specular;      // specular color
152     fixed3 Normal;        // tangent space normal, if written
153     half3 Emission;
154     half Smoothness;      // 0=rough, 1=smooth
155     half Occlusion;        // occlusion (default 1)
156     fixed Alpha;          // alpha for transparencies
157 };
158
```

【3】由【2】中的几种SurfaceOutput结构体可知，Unity存在多种光照模型：

- 1、Unity5.x之前的简单的、不基于物理的光照模型，包括Lambert（兰伯特光照模型，Diffuse）和BlinnPhong（Specular）
- 2、Unity5.x之后的基于物理的光照模型，包括Standard（默认的金属工作流程，对应光照函数为Standard）和StandardSpecular（基于高光工作流程，对应的光照函数为StandardSpecular）
- 3、当然，也可以自定义光照函数，如MyCalc，则在新建后缀为cginc或者在unity安装目录下的CGInclude文件夹中原有的文件中添加新定义的光照函数
声明类似于：

half4 **Lighting**MyCalc(SurfaceOutputs,参数省略){ } 注意：需要在自定义的函数名前加上“Lighting”

【4】其他可选参数，待补充了解

- alpha -透明(Alpha)混合模式。使用它可以写出半透明的着色器。
- alphatest:VariableName -透明(Alpha)测试模式。使用它可以写出 镂空效果的着色器。镂空大小的变量(VariableName)是一个float型的变量。
- vertex:VertexFunction - 自定义的顶点函数(vertex function)。相关写法可参考Unity内建的Shader：树皮着色器(Tree Bark shader)，如Tree Creator Bark、Tree Soft Occlusion Bark这两个Shader。
- **finalcolor:ColorFunction - 自定义的最终颜色函数(final color function)**

注意：此处区分大小写 finalColor X

常见书写格式：

```
#pragma surface surf Lambert finalcolor:setcolor
void setcolor(Input IN, SurfaceOutput o, inout fixed4 color)
//注意：***需要三个参数***
{
    color *= _ColorTint;
}
```

- 其他：
- exclude_path:prepass 或者 exclude_path:forward - 使用指定的渲染路径，不需要生成通道。

- addshadow - 添加阴影投射 & 收集通道(collector passes)。通常用自定义顶点修改，使阴影也能投射在任何程序的顶点动画上。
- dualforward - 在正向(forward)渲染路径中使用 双重光照贴图(dual lightmaps)。
- fullforwardshadows - 在正向(forward)渲染路径中支持所有阴影类型。
- decal:add - 添加贴图着色器(decal shader) (例如： terrain AddPass)。
- decal:blend - 混合半透明的贴图着色器(Semitransparent decal shader)。
- softvegetation - 使表面着色器(surface shader)仅能在Soft Vegetation打开时渲染。
- noambient - 不适用于任何环境光照(ambient lighting)或者球面调和光照(spherical harmonics lights)。
- novertexlights - 在正向渲染(Forward rendering)中不适用于球面调和光照(spherical harmonics lights)或者每个顶点光照(per-vertex lights)。
- nolightmap - 在这个着色器上禁用光照贴图(lightmap)。(适合写一些小着色器)
- nodirlightmap - 在这个着色器上禁用方向光照贴图(directional lightmaps)。(适合写一些小着色器)。
- noforwardadd - 禁用正向渲染添加通道(Forward rendering additive pass)。这会使这个着色器支持一个完整的方向光和所有光照的per-vertex/SH计算。(也是适合写一些小着色器)。
- approxview - 着色器需要计算标准视图的每个顶点(per-vertex)方向而不是每个像素(per-pixel)方向。这样更快，但是视图方向不完全是当前摄像机(camera)所接近的表面。
- halfasview - 在光照函数(lightning function)中传递进来的是half-direction向量，而不是视图方向(view-direction)向量。Half-direction会计算且会把每个顶点(per vertex)标准化。这样做会提高执行效率，但是准确率会打折扣。
- 此外，还可以在 CGPROGRAM块内编写 `#pragma debug`，然后表面编译器(surface compiler)会进行解释生成代码。

结合上面四点，编译指令可以如下形式：

```
#pragma surface surf Lambert finalcolor:mycolor。
```

表面着色器整体结构：

表面着色器放在CGPROGRAM...ENDCG块中，但必须放在SubShader块中，不是Pass{}里，区别于顶点/片元着色器的结构：**SubShader{ CGPROGRAM{ Pass{ .. } }ENDCG }**

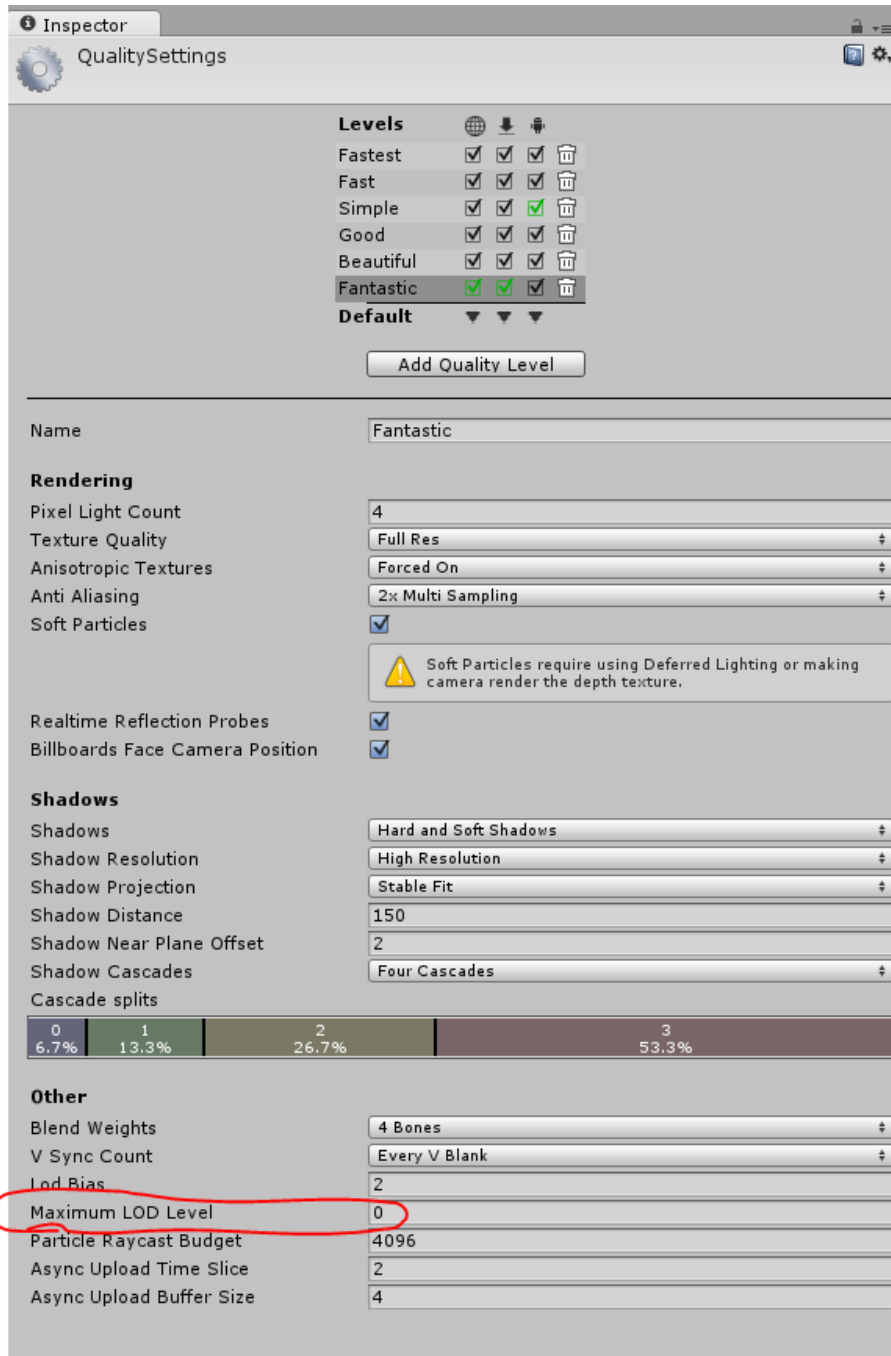
LOD:

Level of Detail

这个数值决定了我们能用什么样的Shader。在Unity的Quality Settings中我们可以设定允许的最大LOD，当设定的LOD小于SubShader所指定的LOD时，这个SubShader将不可用。Unity内建Shader定义了一组LOD的数值，我们在实现自己的Shader的时候可以将其作为参考来设定自己的LOD数值，这样在之后调整根据设备图形性能来调整画质时可以进行比较精确的控制。

- VertexLit及其系列 = 100
- Decal, Reflective VertexLit = 150
- Diffuse = 200

- Diffuse Detail, Reflective Bumped Unlit, Reflective Bumped VertexLit = 250
- Bumped, Specular = 300
- Bumped Specular = 400
- Parallax = 500
- Parallax Specular = 600



Vertex Shader & Fragment Shader :

与DirectX的HLSL和CG紧密相连，Unity中就是用ShaderLab进行了包装。

就渲染管线中的概念总结：

顶点着色器：产生纹理坐标，颜色，点大小，雾坐标，然后把它们传递给裁剪阶段。

片段着色器：进行纹理查找，决定什么时候执行纹理查找，是否进行纹理查找，及把什么作为纹理坐标。

注意：由于顶点着色器一般是渲染管线的最开始，所以Unity预定义了一些常用的输入数据结构（下面列出的是针对顶点着色器的输入结构：注意区分表面着色器的输入结构input）

数据结构	含义
appdata_base	顶点着色器输入位置、法线以及一个纹理坐标。
appdata_tan	顶点着色器输入位置、法线、切线以及一个纹理坐标。
appdata_full	顶点着色器输入位置、法线、切线、顶点颜色以及两个纹理坐标。
appdata_img	顶点着色器输入位置以及一个纹理坐标。

具体查找路径：...\Editor\Data\CGIncludes\unityCG.cginc

如果想让顶点着色器和片元着色器进行通信，需要保证顶点着色器传递了一个语义为（POSITION或者SV_POSITION）的变量，其中包含了顶点坐标

语义：

特殊情况

SV_POSITION 替换 POSITION

SV_Target 替换 COLOR

Vertex shader & fragment shader 大致形式

```
Shader "Inspect面板显示名称/子菜单名称"{
    Properties{
        _Color("Inspect中的Shader面板显示名称", Color) = (1,1,1,1)
        .....
    }
    SubShader{
        pass{
            CGPROGRAM
            //定义顶点着色器和片元着色器的函数名
            #pragma vertex vert
            #pragma fragment frag
            //引用后缀为 .cginc 的文件 包含了一些常用的函数和宏
            #include "unitycg.cginc"

            //声明结构体
            struct v2f{
                float4 pos : POSITIVE;
                fixed4 color : COLOR;
```

```

};
//声明面板属性 (uniform)
float4 _Color;

v2f vert(appdata_base v)    //顶点着色器
{
    v2f o;
    o.pos = mul(UNITY_MATRIX_MVP,v.vertex);
    o.color = fixed4(1,1,1,1);    //float4 fixed4 half4仅仅表示精度，精度有高到低
    return o;                    //输出顶点颜色
}
fixed4 frag(v2f IN) : COLOR
{
    ..... //片元着色器
    return IN.color;            //输出像素颜色
}
ENDCG
}
}
}

```