一、什么是shader？

- Shader君简介
  中文名：着色器
  本质：只是一段程序而已
  功能：用来处理3D图形的渲染过程

- 这么介绍大家一定还是不懂，还是让Shader的搭档们也介绍一下自己吧~

shader的小伙伴们

- 模型君Model：
大家好，我是一个3D模型，来自3D Max、Maya等建模软件。我有九百多个顶点，而且我知道每个顶点的位置、法线，还有他们对应的贴图的坐标。

- 材质君Material：
Model说的没错，作为一个胖纸，它拥有的信息非常多。然而，即使有这些信息，它仍然只是个没穿衣服的胖子。作为材质君Material，我可以描述他表面的颜色(主颜色)，贴图等等，当然，我可以提供多少信息还得Shader君说了算，最终显示效果也是它说了算……

- 着色器君Shader：
顶楼上！以上两位M君虽然包含一个3D模型的渲染的很多信息，但是最终的渲染效果还是我说了算。我可以使用Model告诉我的顶点位置，也可以修改后在使用。我可以用Material提供的信息决定最终的渲染颜色，也可以完全不理它。嘿嘿，总之，看心情~

关系总结

- 1、直接和渲染打交道的是Shader，模型和材质只是给Shader提供信息。shader好比是一个裁缝，模型就是一个没穿衣服的人，而材质就是布料。

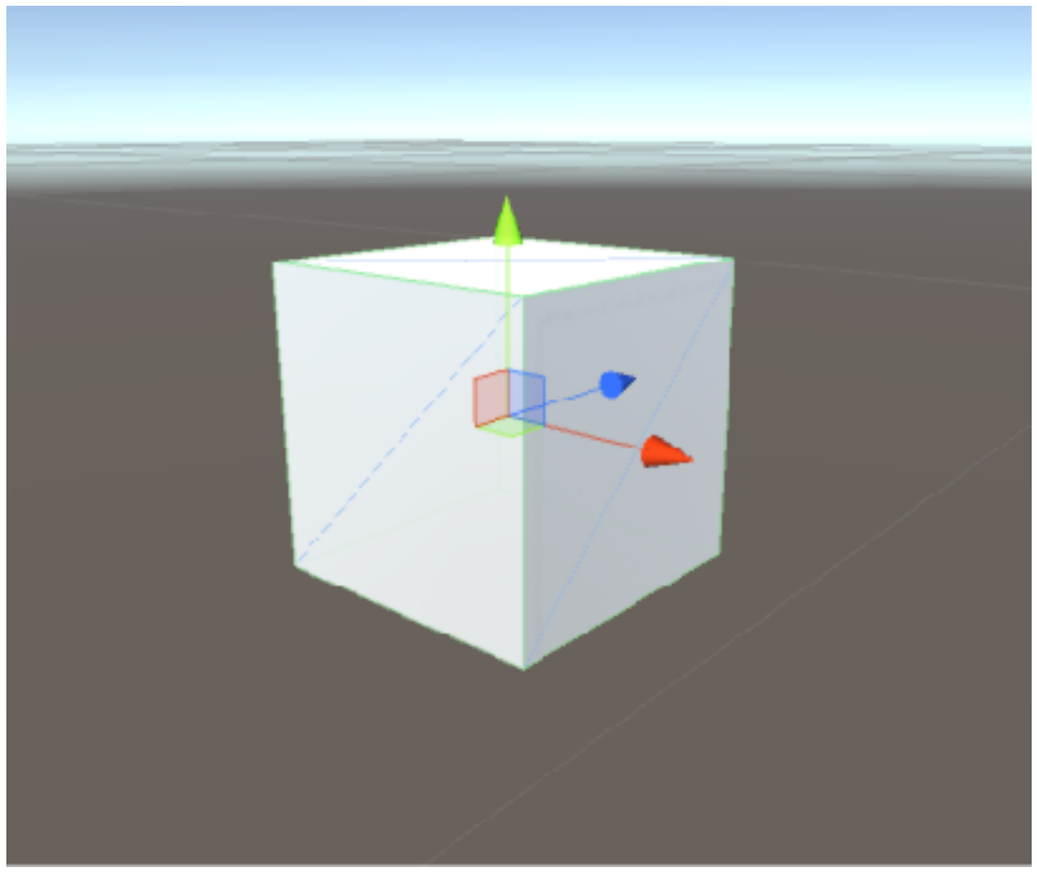
- 2、Shader君拥有很高的自主权，它可以选择使用模型和材质提供的信息，也可以不使用(当然一般不这么做)，也可以用这些信息经过复杂运算后使用。就像你给裁缝提供了你的身体数据，裁缝可以给你在冬天做一件短袖，也可以在夏天给你做一件棉袄，还能秀几朵小花....虽然他很任性，不过一般都会好好干活的。。。

二、搭建shader学习环境

- 首先打开Unity，创建一个空的工程，创建几个文件夹便于我们存放资源，首先是三位君的屋子：

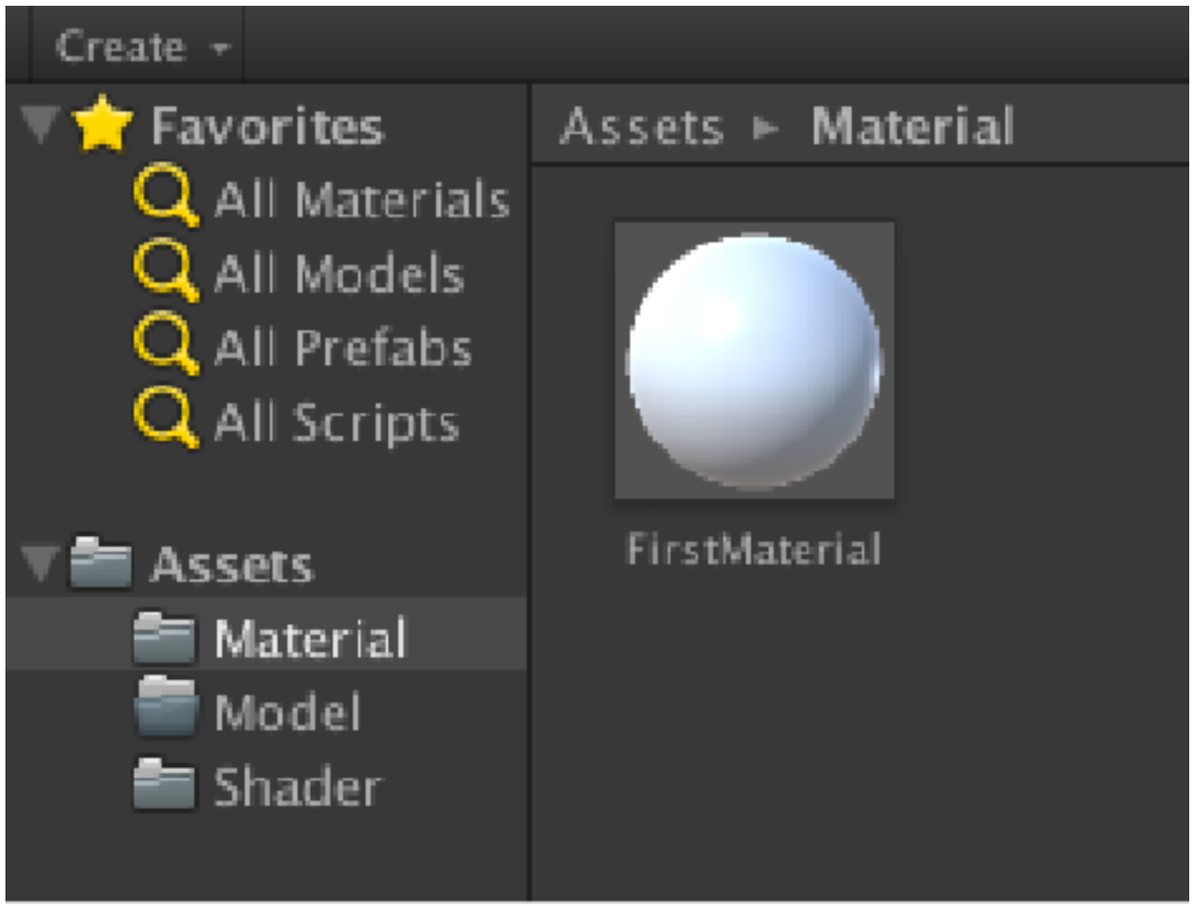
搭建shader学习环境

- 首先创建一个Cube，从今往后他就是我们的模型君了



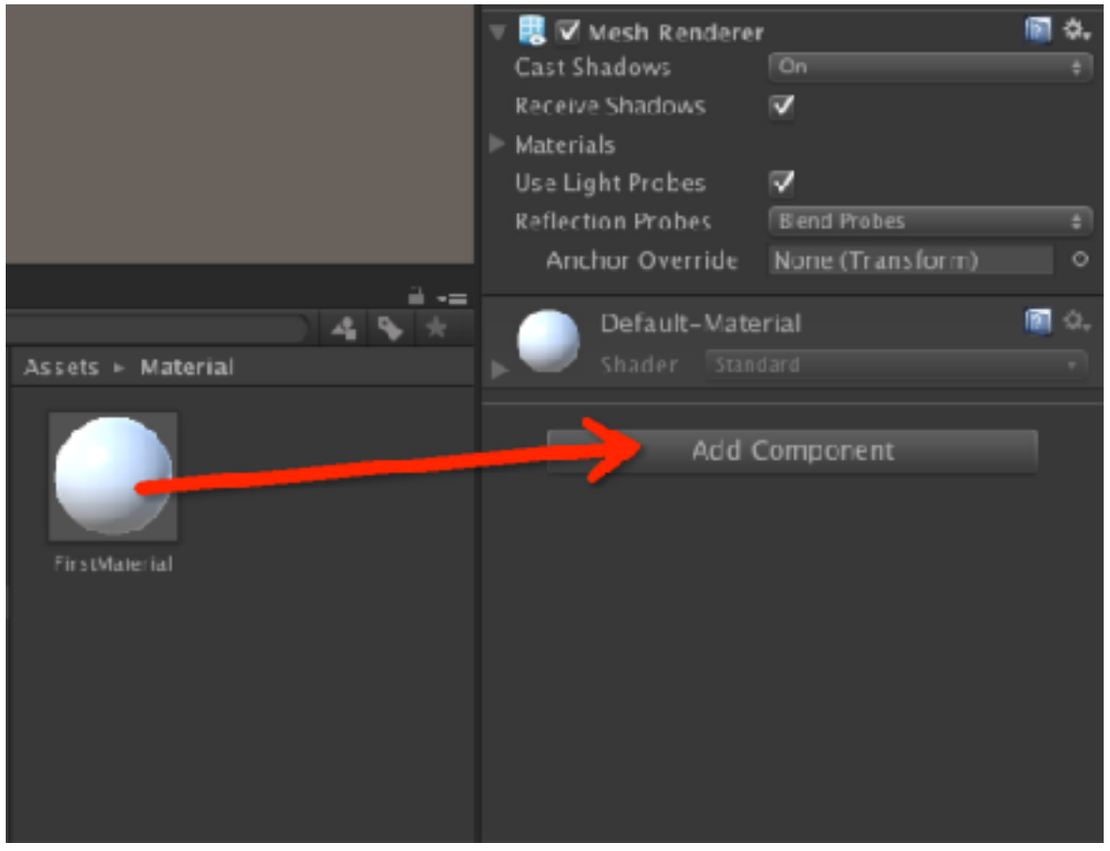
搭建shader学习环境

- 在Material文件夹中右键，创建一个Material命名为FirstMaterial

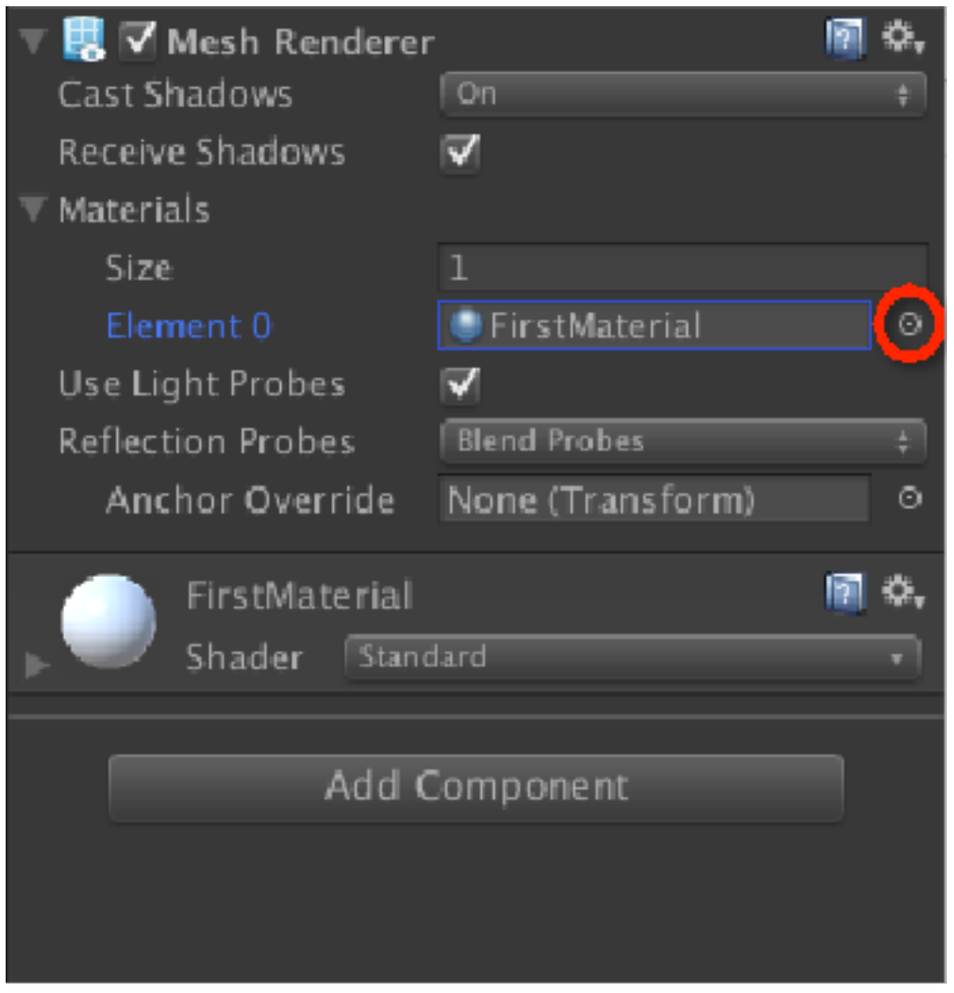
- 

搭建shader学习环境

- 好了，现在我们的模特（模型君）有了，衣服也有了，先给他穿衣服吧
- 方式一：直接将材质球拖到Add Component的位置

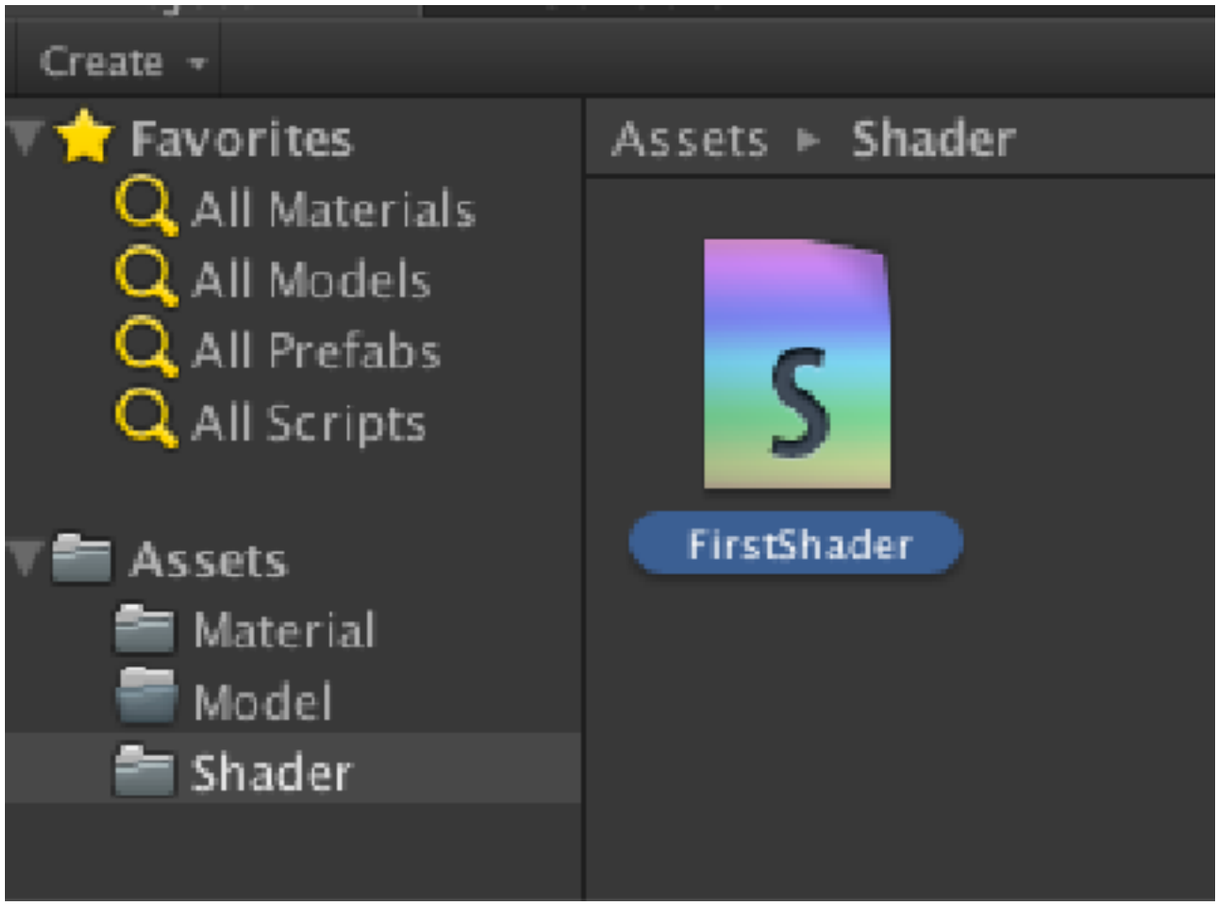


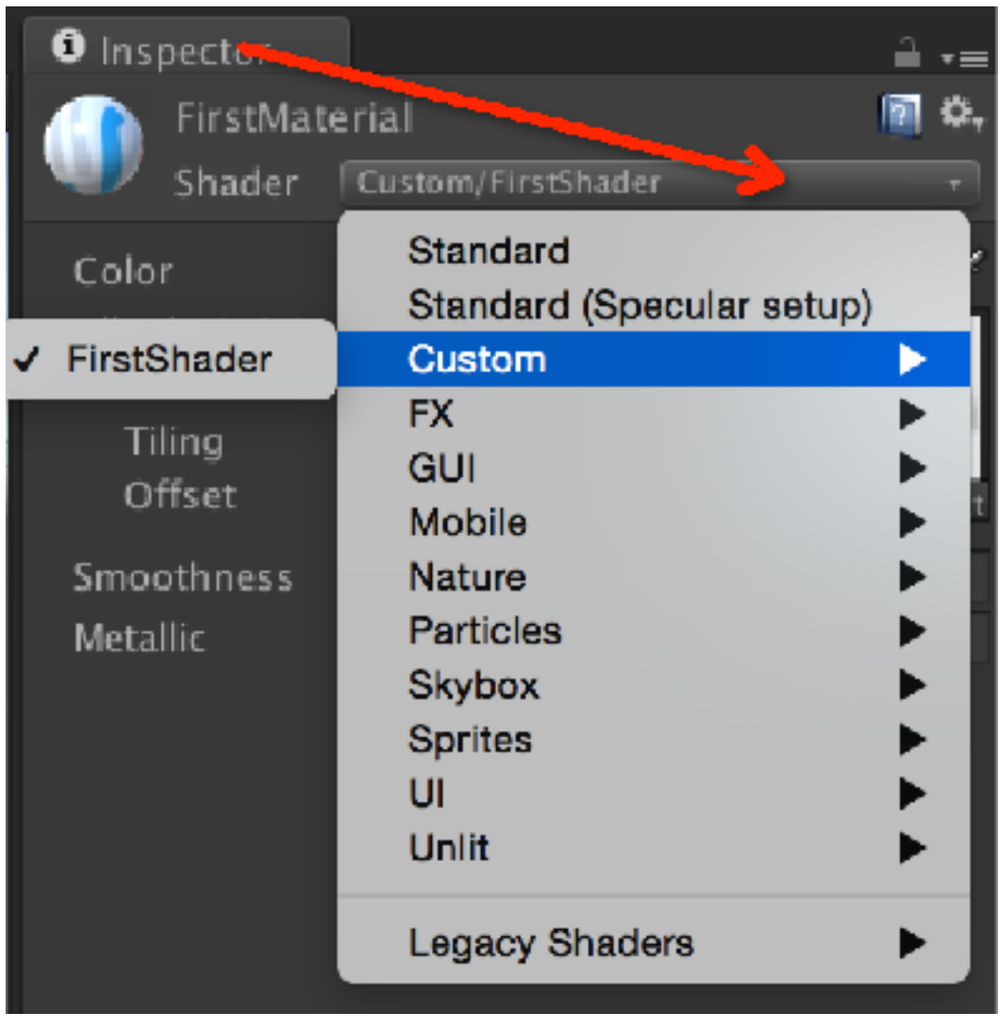
方式二：点击Cube身上Mesh Renderer里圈红的选项找到我们新建的材质球



搭建shader学习环境

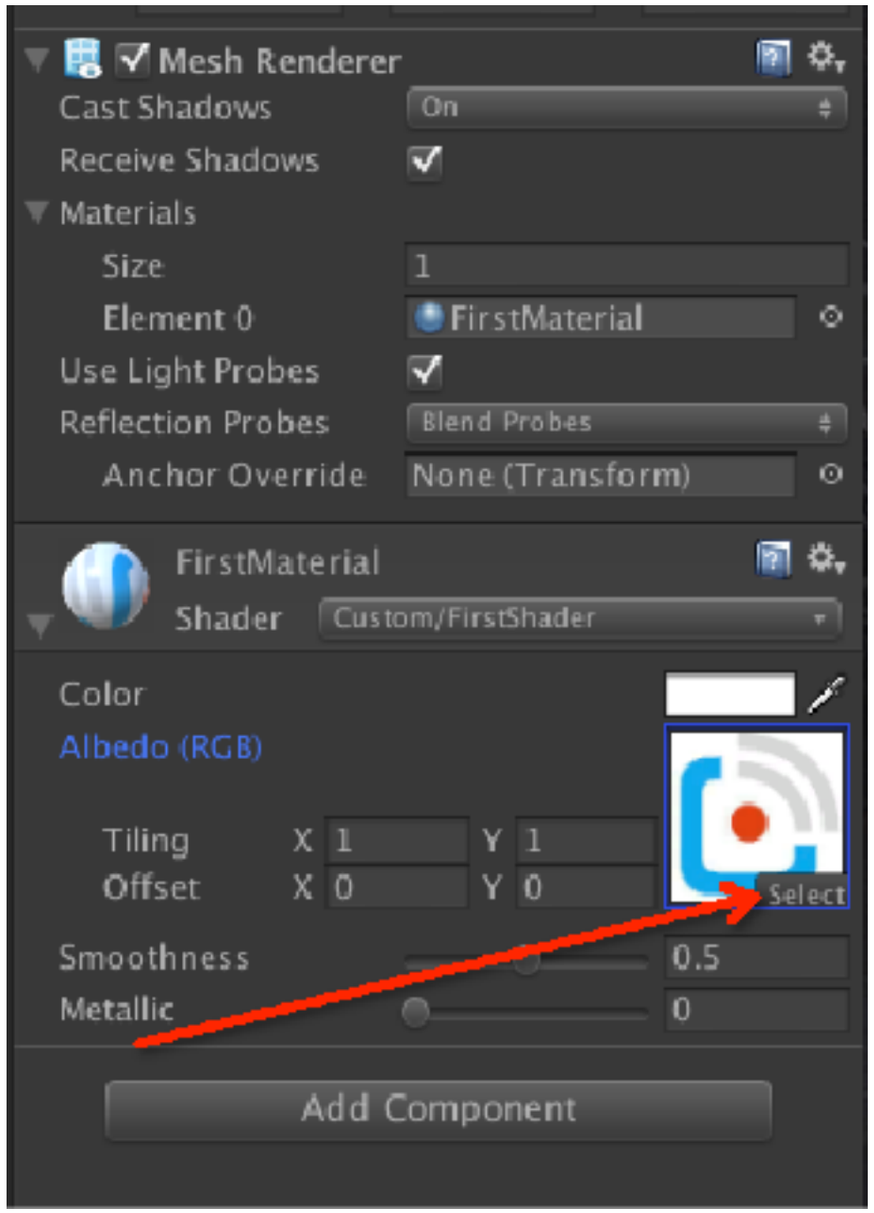
- 在Shader文件夹下右键创建一个Shader脚本，然后选择Material文件夹中的材质球，在菜单中选择Custom/FirstShader(此处应该是自己的Shader名)：





搭建shader学习环境

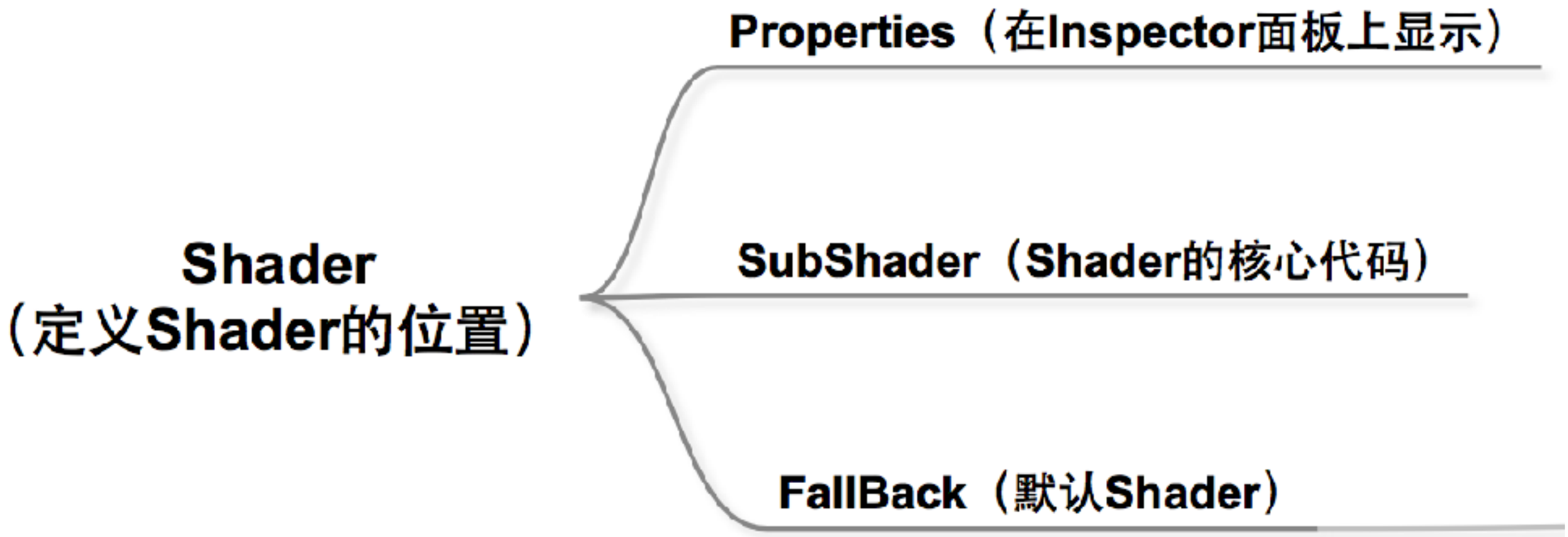
- 现在我们的模型君身上的衣服还是纯白色的，我们来给他换个花纹的



现在点击Cube，可以看到它的Material不仅可以改变颜色，还能选择图片！

这个是Shader代码里提供的一个可以选择贴图的接口

三、shader结构介绍



关键字Shader

Unity的Shader文件是通过Shader这个关键字开始的，用户可以像目录一样组织Shader的命名。Shader的文件名和引用名不必一样,如下：

```
// shader的代码，都是以Shader开头
// "Custom/FirstShader"就是我们从材质列表中选择Shader的位置默认在Custom
Shader "Custom/FirstShader" {
```

关键字Properties

```
1 // shader的代码，都是以Shader开头
2 // "Custom/FirstShader"就是我们从材质列表中选择Shader的位置默认在Custom
3 Shader "Custom/FirstShader" {
4     // 是shader的属性，就是在Inspector面板中显示的
5     Properties {
6         // _Color     是在SubShader中使用的时候用的变量名
7         // Color      是在Inspector面板中显示的名字
8         // Color      是变量类型名称
9         // (1,1,1,1) 颜色的RGBA值
10        _Color ("Color", Color) = (1,1,1,1)
11        _MainTex ("Albedo (RGB)", 2D) = "white" {}
12        _Glossiness ("Smoothness", Range(0,1)) = 0.5
13        _Metallic ("Metallic", Range(0,1)) = 0.0
14    }
```

- Shader里的面板属性值与Unity和Cg语言变量名的对应关系。

硬件平台	CPU	（Inspector面板设置）	GPU
变量类型	C#脚本语言	ShaderLab 属性	Cg语言
纹理贴图	Texture	2D	sampler2D
颜色	Color	Color	fixed4 / half4
3D贴图	Cubemap	Cube	sampleCUBE
4元素向量	Vector4	Vector	float4
浮点数	float	Float	float
浮点范围	float	Range	float
矩形纹理	Texture	Rect	sampleRect

关键字SubShader

真正的呈现渲染物体的内容是在SubShader中实现的，之所以用SubShader是为了能让开发者针对不同性能的显卡编号编写不同的Shader，可以同时存在多个SubShader，Unity会针对实际的运行环境在代码中从上到下选择一个最适合的SubShader执行，理论上SubShader数量没有限制，实际操作中为了减少文件的大小，一般写两三个就行了，针对目前最流行的显卡写一个，针对老旧显卡写一个。

```
// 是Shader的主要部分，一个Shader中可以有一个或者多个SubShader，Unity会选择一个最适合的执行
// 如果有多个合适的，会按照从上到下的顺序选择执行，主要的目的是为了兼容新旧显卡，我们可以把只有在
// 新显卡才能执行的代码写在最上面，这样程序会选择显示效果最好的代码来执行，在旧显卡上也可以运行。
SubShader {
    Tags { "RenderType"="Opaque" }
    LOD 200

    CGPROGRAM
    // Physically based Standard lighting model, and enable shadows on all light types
    // 告知 Unity3D 该表面着色器将使用 物理渲染 (PBR)光照模型
    #pragma surface surf Standard fullforwardshadows

    // Use shader model 3.0 target, to get nicer looking lighting
    // 意味着该着色器将使用高级特性，因而其将不同在落后的硬件上使用。
    // 同样的，SurfaceOutput 也不能同 PBR 一起使用；而是必须使用  SurfaceOutputStandard。
    #pragma target 3.0

    sampler2D _MainTex;

    struct Input {
        float2 uv_MainTex;
    };

    half _Glossiness;
    half _Metallic;
    fixed4 _Color;

    void surf (Input IN, inout SurfaceOutputStandard o) {
        // Albedo comes from a texture tinted by color
        fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
        o.Albedo = c.rgb;
        // Metallic and smoothness come from slider variables
        // 金属含量
        o.Metallic = _Metallic;
        // 平滑值
        o.Smoothness = _Glossiness;
        o.Alpha = c.a;
    }
    ENDCG
}
```

关键字SubShader-tags{}

```
Tags {"Queue" = "Geometry" "RenderType"="Opaque" }
```

Queue就是队列的意思，在这里指的是渲染队列，表示希望Unity渲染引擎的在什么时候渲染自己，Queue有5个可选值，分别是：BackGround、Geometry、AlphaTest、Transparent、Overlay

分别对应数字1000、2000、2450、3000、4000，值越小越先渲染，既然可以对应成数字，当然也可以把这些单词当做整形变量来看待，例如：

```
Tags {"Queue" = "Geometry" + 1000}
```

关键字SubShader-tags{}

```
Tags {"Queue" = "Geometry" "RenderType"="Opaque" }
```

RenderType标签常用的内置值有，Opaque、Transparent、TransparentCutout、Background、Overlay。一个正确的渲染方式在Shader中是必不可少的。

关键字FallBack

- 如果用户所有的SubShader都失败了，为了在用户的计算机上能呈现设定的机制，一般会使用FallBack。FallBack是Unity自己预制的Shader实现，一般能够在所有显卡上运行。我们在开发Shader的时候一般不使用FallBack，只有在实际发布的时候才会为了追求平台的最大适用性而追加上。

```
// Shader的“备胎”,类似于default语句，当所有的SubShader都不能执行的时候
// Unity就会选择执行FallBack来执行。
FallBack "Diffuse"
```

ShaderLab所支持的语言

- 在Unity的ShaderLab所提供的结构中，我们可以使用GLSL和Cg/HLSL来写Shader的逻辑代码，需要注意的是：

如果使用GLSL语言：

```
#GLSLPROGRAM
// GLSL的代码要写在这两行关键字之间
#ENDGLSL
```

如果使用Cg/HLSL语言：

```
CGPROGRAM
// Cg的代码要写在这两行关键字之间
ENDCG
```

四、Surface Shader

- 如果你想写一个能处理不同的照明、点光源、平行光又能处理不同的阴影选项，还能处理两个渲染路径（Forward和Deferrd）下正常工作，是一件很复杂的事情。Unity通过Surface Shader（表面着色器）把上面一切复杂性包装了起来，看一个简单的例子：

Surface Shader

```
Shader "Custom/mySurfaceShader" {

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

    SubShader
    {
        Tags { "RenderType" = "Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf Lambert
        sampler2D _MainTex;
        struct Input
        {
            float2 uv_MainTex;
        };

        void surf (Input IN,inout SurfaceOutput o)
        {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }

        ENDCG
    }
    FallBack "Diffuse"
}
```

Surface Shader

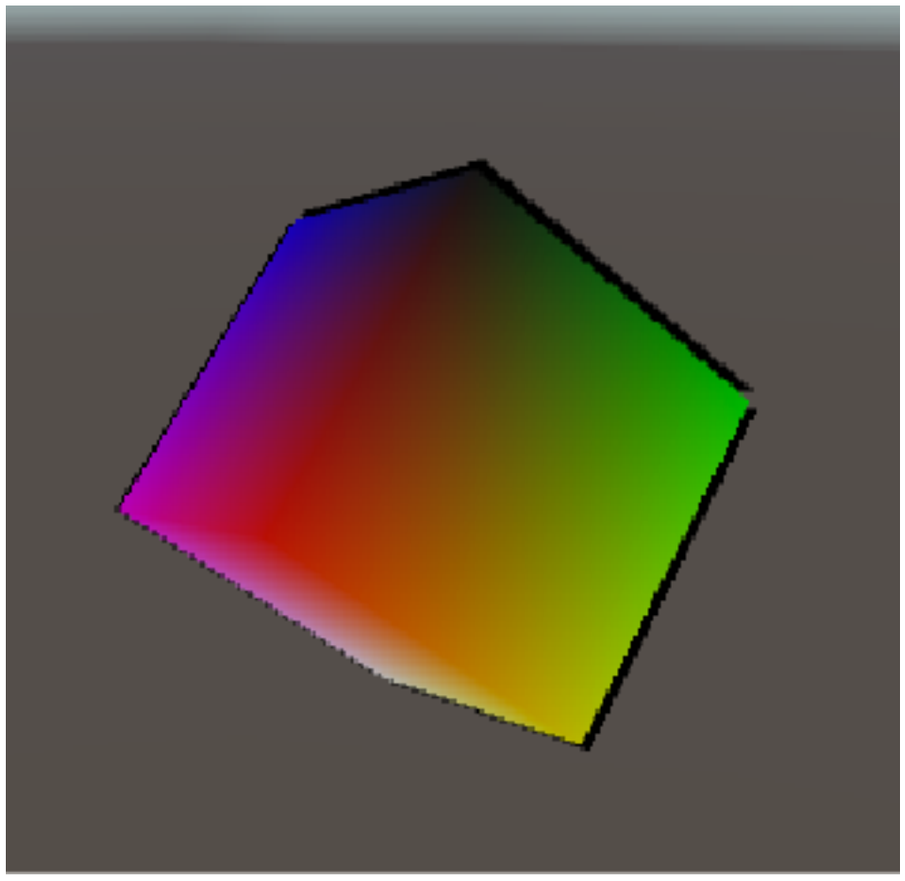
在Surface函数的surf中，SurfaceOutput是一个包含大多数描述一个物体表面渲染特征的系统内置的结构体：

```
struct SurfaceOutput
{
    half3 Albedo;   // 颜色
    half3 Normal;   // 法线
    half3 Emission; // 自发光，不受光照影响
    half  Specular; // 高光指数
    half  Gloss;    // 光泽度
    half  Alpha;    // 透明度
}
```

SurfaceOutput		
half3	Abedo	反射光
half3	Normal	法线
half3	Emission	自发光
half	Specular	高光
half	Alpha	透明度

Vertex And Fragment Shader

顶点和片段着色器和其他着色器一样都由4部分组成
以下的代码是右边效果图的代码截图
第一部分：Shader关键字部分
作用：确定shader代码的选择路径

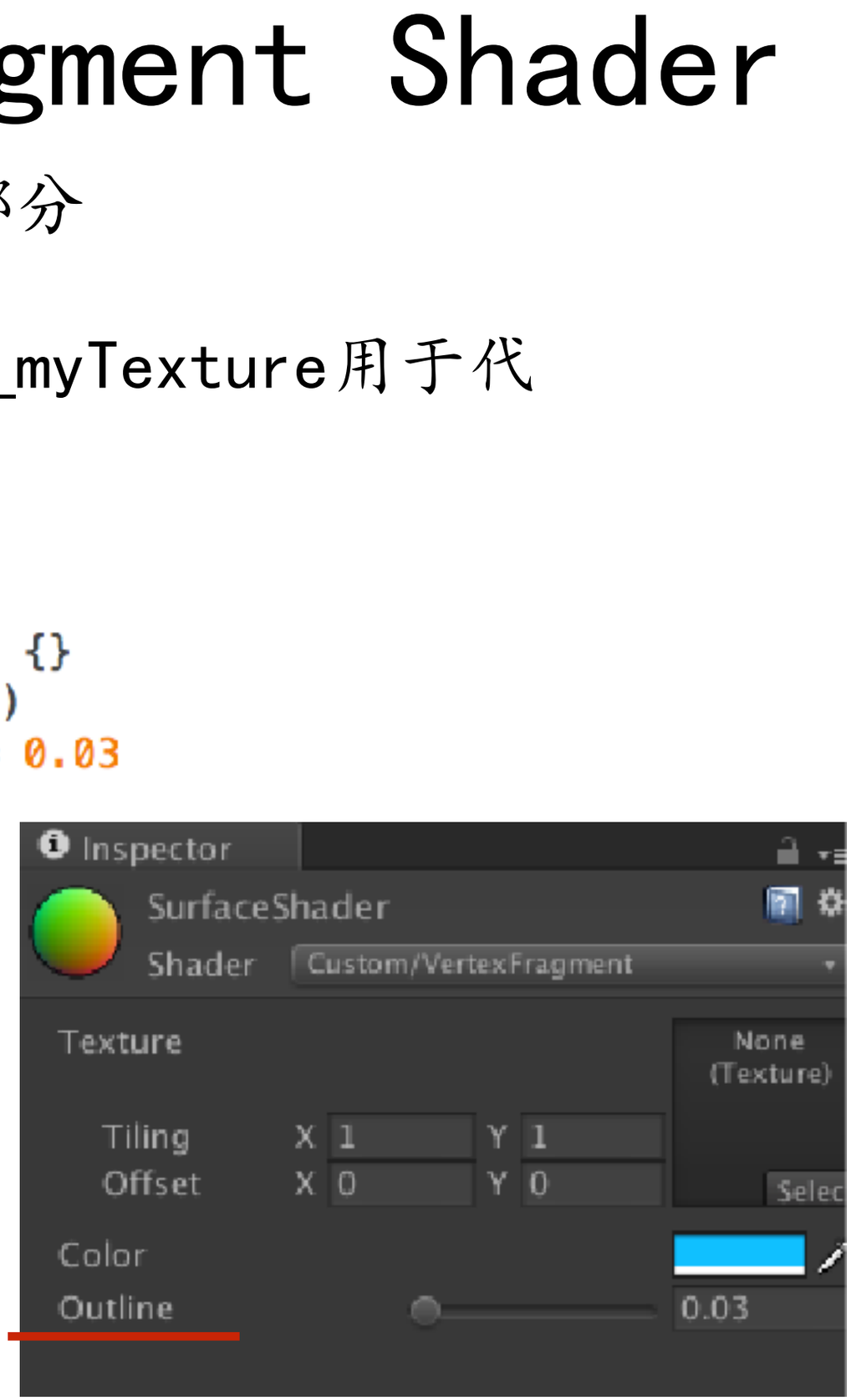


```
Shader "Custom/VertexFragment" {
```

Vertex And Fragment Shader

第二部分：Properties关键字部分
作用：在Inspector面板中显示
Texture用于面板显示的名字，_myTexture用于代码中调用

```
Properties
{
    _myTexture("Texture", 2D) = "white" {}
    _myColor("Color", Color) = (1,1,1,1)
    _Outline ("Outline", Range(0,1)) = 0.03
}
```



Vertex And Fragment Shader

第三部分：SubShader
作用：Shader代码的主要部分

```
SubShader
{
    Tags{"Queue" = "Geometry"  "RenderType"="Opaque" "IgnoreProjector" = "True"}
    // 第一个通道
    Pass
    {
        CGPROGRAM
        // 声明顶点shader函数
        #pragma vertex vert
        // 声明片段shader函数
        #pragma fragment frag
        // 使用Vertex and Fragment的CG时
        // 会#include "UnityCG.cginc",用到里面的很多函数
        #include "UnityCG.cginc"
        sampler2D _myTexture;
        float4    _myColor;
        struct v2f{
        float4 pos:SV_POSITION;
        float3 color :COLOR;
        };
        // appdata_full v是"UnityCG.cginc"里的结构体
        v2f vert(appdata_full v)
        {
            v2f o;
            // UNITY_MATRIX_MVP 当前模型视图投影矩阵
            o.pos = mul(UNITY_MATRIX_MVP, v.vertex);
            // 彩虹色
            o.color = v.vertex * 0.8 + 0.5;
            // 纯色
            o.color = v.normal * 0.4 + 0.5;
            return o;
        }

        float4 frag(v2f i):COLOR
        {
            return float4(i.color,1);
        }
        ENDCG
}
```

Vertex And Fragment Shader

第三部分：SubShader

会发现，比之前的代码多出了Pass关键字部分。
Pass：渲染通道
其实是SubShader包装了一个渲染方案，而这个方案是由一个个Pass块来执行的，可以包含多个Pass块，每个Pass都包含了渲染一个几何体的具体代码，我们如果写Shader，大部分费神费力且能体现每位作者劳动价值的地方就在Pass块中。

```
// 第二个通道
Pass {
    Tags { "LightMode"="ForwardBase" }
    Cull Front
    Lighting Off
    ZWrite On
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #pragma multi_compile_fwdbase
    #include "UnityCG.cginc"
    float _Outline;
    struct a2v
    {
        float4 vertex : POSITION;
        float3 normal : NORMAL;
    };

    struct v2f
    {
        float4 pos : POSITION;
    };

    v2f vert (a2v v)
    {
        v2f o;

        float4 pos = mul( UNITY_MATRIX_MV, v.vertex);
        float3 normal = mul( (float3x3)UNITY_MATRIX_IT_MV, v.normal);
        // 轮廓黑线的宽度
        pos = pos + float4(normalize(normal),0) * _Outline;

        o.pos = mul(UNITY_MATRIX_P, pos);

        return o;
    }

    float4 frag(v2f i) : COLOR
    {
        return float4(0, 0, 0, 1);
    }
    ENDCG
}
```

Vertex And Fragment Shader ——— Pass

1. 我们为什么需要多个Pass?

就我们的事例而言，我们简单分析，首先我们第一个Pass渲染了Cube的彩虹颜色，使用到了模型的一些数据进行计算，最后return，ruturn的是计算后的颜色。然后是我们的第二个Pass也使用模型的数据计算，return一个边框的颜色。

像这种需要多次渲染的时候就需要用到多个Pass通道了。

2. Pass块的意义：

```
Shader "Custom/TestShader" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
    }
    SubShader {
        Pass
        {
            Name "MYTEST"
            Material{
                Diffuse(1, 0.2, 0.4, 1)
                Ambient(1, 0.2, 0.4, 1)
            }
            Lighting On
        }
    }
    FallBack "Diffuse"
}
```

```
Shader "Custom/ApplyShader" {
    Properties {
        _Color ("Color", Color) = (1,1,1,1)
    }
    SubShader {
        UsePass "Custom/TestShader/MYTEST"
    }
    FallBack "Diffuse"
}
```

在另一个Shader脚本中可以直接调用其他脚本中定义好的Pass代码块，但需要通过名字调用

（注：Name 后面的名字必须大写）