# [2023.11.23]UnityShader概述

## 前言

前面我们了解过了，Shader就是渲染流水线中的某些特定阶段，比如顶点着色器、片元着色器等；

如果使用C#代码的形式来展现设置一个物体的渲染状态，那它可能会被定义成如下的样式：

```
void Initialize()
{
    //加载顶点着色器着色器
    var vertexShader = Load(VertexShader.shader);
    //加载片元着色器着色器
    var fragmentShader = Load(FragmentShader.shader);
    //加载顶点着色器到GPU
    LoadVertexShader(vertexShader);
    //加载片元着色器到GPU
    LoadFragmentShader(fragmentShader);

    //设置属性名为"_VertexPos"的顶点坐标
    SetVertexShaderProperty(_VertexPos, vetrices);
    //设置属性名为"_MainTex"的纹理
    SetVertexShaderProperty(_MainTex, texture);
    //设置属性名为"_MVP"的变换矩阵
    SetVertexShaderProperty(_MVP, MVP);

    //关闭混合
    ENABLE_BLEND = false;
    //设置深度测试
    ENABLE_ZTEST = true;
    //设置测试函数
    SetZTestFunction(LessOrEqual);
    //其他
    ...
}
//每帧渲染
void OnRendering()
{
    //调用渲染命令
    DrawCall();
    //当涉及多种渲染设置时，我们还需要在这里改变各种渲染设置；
}
```

此外还需要Shader的具体实现；

```
Shader VertexShader.shader{
    //输入：顶点位置、纹理、MVP变换矩阵
    in float3 _VertexPos;
    in sampler2D _MainTex;
    in Matrix4x4 _MVP;

    //输出：顶点经过MVP变换后的位置
    out float4 position;

    void main(){
        position = _MVP * _VertexPos;
    }
}

Shader FragmentShader.shader{
    //输入：VertexShader输出的position、经过光栅化程序插值后的该片元对应的position
    in float4 position;

    //输出：该片元的颜色值
    out float4 fragColor;

    void main(){
        //将片元颜色设为白色
        fragColor = float4(1.0, 1.0, 1.0, 1.0);
    }
}
```

经过上述过程，一个简化版的Shader流程就通过伪代码的形式展示出来了；

# Unity Shader概述

你应该知道，Unity的Shader需要配合材质球一起使用；

Unity Shader定义了渲染所需的各种代码(如顶点着色器和片元着色器)、属性(哪些纹理)和指令(渲染和标签设置等)，而材质允许我们调节这些属性，并赋予相应的模型；而材质一般需要配合Mesh或者粒子来工作；(方便描述，后续Unity Shader统称为Shader)

# Unity Shader的结构

## 名字

每个Shader文件的第一行都需要通过Shader语义来指定该Shader的名字，你可以通过在名字前添加'/'来指定该Shader的归属；

```
Shader "Example/Example Shader"{
}
```

## 属性

属性(Properties)是Shader和材质的桥梁，Properties语义块中包含了一系列属性(property)，这些属性将出现在材质面板中；

声明属性可以更方便地在材质面板中去做调整，我们在Shader中访问属性时，需要使用它的名字(**Name**)，这些名字通常以'_'为起始；显示的名称(**display name**)则是出现在材质面板上的名字；之后我们还要为每个属性指定它的类型(PropertyType)；

所以通常一个属性的声明是：`Name ("display name", PropertyType) = DefaultValue`；

表 3.1 Properties 语义块支持的属性类型

| 属性类型 | 默认值的定义语法 | 例子 |
|---|---|---|
| Int | number | _Int ("Int", Int) = 2 |
| Float | number | _Float ("Float", Float) = 1.5 |
| Range(min, max) | number | _Range("Range", Range(0.0, 5.0)) = 3.0 |
| Color | (number,number,number,number) | _Color ("Color", Color) = (1,1,1,1) |
| Vector | (number,number,number,number) | _Vector ("Vector", Vector) = (2, 3, 6, 1) |
| 2D | "defaulttexture" {} | _2D ("2D", 2D) = "" {} |
| Cube | "defaulttexture" {} | _Cube ("Cube", Cube) = "white" {} |
| 3D | "defaulttexture" {} | _3D ("3D", 3D) = "black" {} |

1. 对于**Int、Float、Range**这些数字类型的属性，其默认值就是一个单独的数字；
2. 对于**Color、Vector**这类属性，默认值是用圆括号包围的一个四维向量`(x,y,z,w)`；
3. 对于2D、Cube、3D这种纹理类型，默认值的定义则是通过字符串后跟花括号来指定的`"string" {}`；字符串要么是空的，要么是内置的纹理名称，如"white"、"black"、"gray"或"bump"等；

```
Shader "Example/Example Shader"{
    Properties{
        // Numbers and Sliders
        _Int ("Int", Int) = 2
        _Float ("Float", Float) = 1.5
        _Range ("Range", Range(0.0, 5.0)) = 3.0
```

```
            // Colors and Vectors
            _Color ("Color", Color) = (1,1,1,1)
            _Vector ("Vector", Vector) = (2,3,6,1)
            // Textures
            _2D ("2D", 2D) = ""{}
            _Cube ("Cube", Cube) = "white"{}
            _3D ("3D", 3D) = "black"{}
        }

        Fallback "Diffuse"
    }
```

## 子着色器

一个Shader文件中可以包含多个SubShader语义块(翻译过来应该是子着色器吧..)，但至少要有一个；当Unity需要加载这个Shader时，它会扫描所有的SubShader语义块，然后选择第一个能够在目标平台上运行的SubShader；如果都不支持的话，Unity就会用Fallback指定的Shader；SubShader在兼容低端机上能起到非常大的作用；

SubShader中定义了一系列Pass以及**可选的状态(RenderSetup)**和**标签[Tags]**设置，每个Pass定义了一次完整的渲染流程，但如果Pass的数量过多，往往会造成渲染性能的下降；

状态和标签可以在Pass中声明，对于状态来说，如果在SubShader中进行了设置，那么将会应用于所有的Pass，但SubShader中的一些标签是特定的，和在Pass中的不一样；

```
    SubShader{
        //可选
        [Tags]
        //可选
        [RenderSetup]

        Pass{
        }
    }
```

## 状态设置

ShaderLab提供了一系列渲染状态的设置指令，这些指令可以设置显卡的各种状态；

表 3.2　　常见的渲染状态设置选项

| 状态名称 | 设置指令 | 解　释 |
| --- | --- | --- |
| Cull | Cull Back \| Front \| Off | 设置剔除模式：剔除背面/正面/关闭剔除 |
| ZTest | ZTest Less Greater \| LEqual \| GEqual \| Equal \| NotEqual \| Always | 设置深度测试时使用的函数 |
| ZWrite | ZWrite On \| Off | 开启/关闭深度写入 |
| Blend | Blend SrcFactor DstFactor | 开启并设置混合模式 |

## SubShader的标签

SubShader的标签(**Tags**)是一个键值对，它们都是字符串类型，它用来告诉Unity引擎，我要怎么渲染这个对象；

```
Tags {"TagName1" = "Value1" "TagName2" = "Value2"}
```

表 3.3　　SubShader 的标签类型

| 标签类型 | 说　明 | 例　子 |
| --- | --- | --- |
| Queue | 控制渲染顺序，指定该物体属于哪一个渲染队列，通过这种方式可以保证所有的透明物体可以在所有不透明物体后面被渲染（详见第 8 章），我们也可以自定义使用的渲染队列来控制物体的渲染顺序 | Tags { "Queue" = "Transparent" } |
| RenderType | 对着色器进行分类，例如这是一个不透明的着色器，或是一个透明的着色器等。这可以被用于着色器替换（Shader Replacement）功能 | Tags { "RenderType" = "Opaque" } |
| DisableBatching | 一些 *SubShader* 在使用 Unity 的批处理功能时会出现问题，例如使用了模型空间下的坐标进行顶点动画（详见 11.3 节）。这时可以通过该标签来直接指明是否对该 *SubShader* 使用批处理 | Tags { "DisableBatching" = "True" } |
| ForceNoShadowCasting | 控制使用该 *SubShader* 的物体是否会投射阴影（详见 8.4 节） | Tags { "ForceNoShadowCasting" = "True" } |
| IgnoreProjector | 如果该标签值为“True”，那么使用该 *SubShader* 的物体将不会受 Projector 的影响。通常用于半透明物体 | Tags { "IgnoreProjector" = "True" } |
| CanUseSpriteAtlas | 当该 *SubShader* 是用于精灵（sprites）时，将该标签设为“False” | Tags { "CanUseSpriteAtlas" = "False" } |
| PreviewType | 指明材质面板将如何预览该材质。默认情况下，材质将显示为一个球形，我们可以通过把该标签的值设为“Plane”“SkyBox”来改变预览类型 | Tags { "PreviewType" = "Plane" } |

上述标签尽可以在SubShader中声明，而不可以在Pass块中声明，这些是属于SubShader的标签类型；

## Pass语义块

Pass语义块包含的语义如下：

```
    Pass{
        [Name]
        [Tags]
```

```
        [RenderSteup]
        // Other Code
    }
```

可以在Pass中定义该Pass的名称，例如：

`Name "PassName"`

通过这个名称，我们可以使用ShaderLab的UsePass命令来直接使用其他Unity Shader中的Pass，例如：

`UsePass "Example/EXAMPLE SHADER"`

Unity内部会把所有Pass转换成大写字母，所以使用UsePass时必须使用全大写的名字；

表 3.4 *Pass* 的标签类型

| 标签类型 | 说明 | 例子 |
| --- | --- | --- |
| LightMode | 定义该 *Pass* 在 Unity 的渲染流水线中的角色 | Tags { "LightMode" = "ForwardBase" } |
| RequireOptions | 用于指定当满足某些条件时才渲染该 Pass，它的值是一个由空格分隔的字符串。目前，Unity 支持的选项有：SoftVegetation。在后面的版本中，可能会增加更多的选项 | Tags { "RequireOptions" = "SoftVegetation" } |

除此之外，还有**UsePass**和**GrabPass**，UsePass上面已经说过了，GrabPass负责抓取屏幕并将结果存储在一张纹理中，用于后续Pass处理；

### Fallback

在SubShader语义块后面跟着Fallback指令，表示如果上面所有的Pass都不能运行，就用Fallback后的Shader；它的语义如下：

```
    Fallback "name"
    Fallback Off
```

比如可以使用 `Fallback "VertexLit"` ；此外Fallback还会影响阴影的投射，在渲染阴影纹理时，Unity会在每个Shader中寻找一个阴影投射的Pass，通常情况下，Fallback的内置Shader中就包含了这样一个通用Pass；

## Unity Shader的形式

官方的介绍中，Unity的Shader种类可以分为3种：

1. 表面着色器(Surface Shader)，它使用Unity预制的光照模型来进行光照和阴影运算；
2. 顶点和片元着色器(Vertex and Fragment Shader)，这就是前面我们提到的着色器，最强大的Shader类型，属于可编程渲染管线；

3. 固定渲染管线(Fixed function shader)，用于高级Shader在老显卡无法显示时的Fallback；个人猜测Unity渲染材质丢失后的物体，呈现通体的粉色的原因，就是它的功劳；

## 表面着色器

表面着色器是Unity自己创造的着色器代码类型，它代码很少但Unity在背后做了很多东西，渲染代价比较大，它在本质上和顶点/片元着色器没有区别，相当于是Unity自己封装了额外一层抽象；

```
Shader "Sample/Sample Surface Shader" {
    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"
}
```

上面一个示例中可以看到，表面着色器被定义在SubShader语义块(非Pass块)中的 `CGPROGRAM` 和 `ENDCG` 中，它不需要开发者关心背后用多少Pass来实现，Unity会自己处理，开发者只需要声明用哪些纹理，哪些光照模型即可；

## 顶点/片元着色器

在Unity里也可以使用CG/HLSL语言来编写顶点/片元着色器，它们更加复杂但灵活性也更高；

```
Shader "Sample/Sample VertexFragment Shader" {
    SubShader {
        Pass {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            float4 vert(float4 v : POSITION) : SV_POSITION {
                return mul (UNITY_MATRIX_MVP, v);
            }
```

```
            fixed4 frag() :  SV_Target {
                return fixed4(1.0,0.0,0.0,1.0);
            }
            ENDCG
        }
    }
}
```

和表年着色器类似，顶点/片元着色器的代码也需要定义在 `CGPROGRAM` 和 `ENDCG` 之间，但不同的是，顶点/片元着色器是写在Pass语义块内，而非SubShader内；因为我们需要自己定义每个Pass需要的Shader代码，这样使它的灵活性很高，我们也可以控制更多渲染细节；

## 固定函数着色器

这种着色器应用于不支持可编程管线着色器的设备，目前看起来是已经被时代抛弃的产物了(其实仔细想一想，很多年以后的未来，也许可编程管线着色器也会被更高维度的产物给击败)，他们往往只可以实现一些非常简单的效果；

```
Shader "Tutorial/Basic" {
    Properties {
        _Color ("Main Color", Color) = (1,0.5,0.5,1)
    }
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

固定函数着色器的代码被定义在Pass块中，相当于Pass中的一些渲染设置；现在绝大多数GPU都支持可编程渲染管线，实际上，现在的Unity版本中，所有固定函数着色器在背后都会被Unity编译成对应的顶点/片元着色器，真正意义上的固定函数着色器已经消亡了；

所以你看，本质上来说，无论是表面着色器还是固定函数着色器，在背后都会被Unity二次编译为顶点/片元着色器，所以实际上，Unity只存在顶点/片元着色器；

## 如何选择着色器

事实上，着色器性能好坏只是一方面的标准，游戏开发中还是尽可能保证美术效果，所以TA嗦了蒜，但如果你想要有个较为清晰的认知，从着色器本身的属性来看，你应该能理解如下几点：

1. 默认使用顶点/片元着色器；
2. 如果需要各种光源的信息，表面着色器会更加方便，但如你所知，它的性能比较差；
3. 如果有自定义各种渲染效果，用顶点/片元着色器；