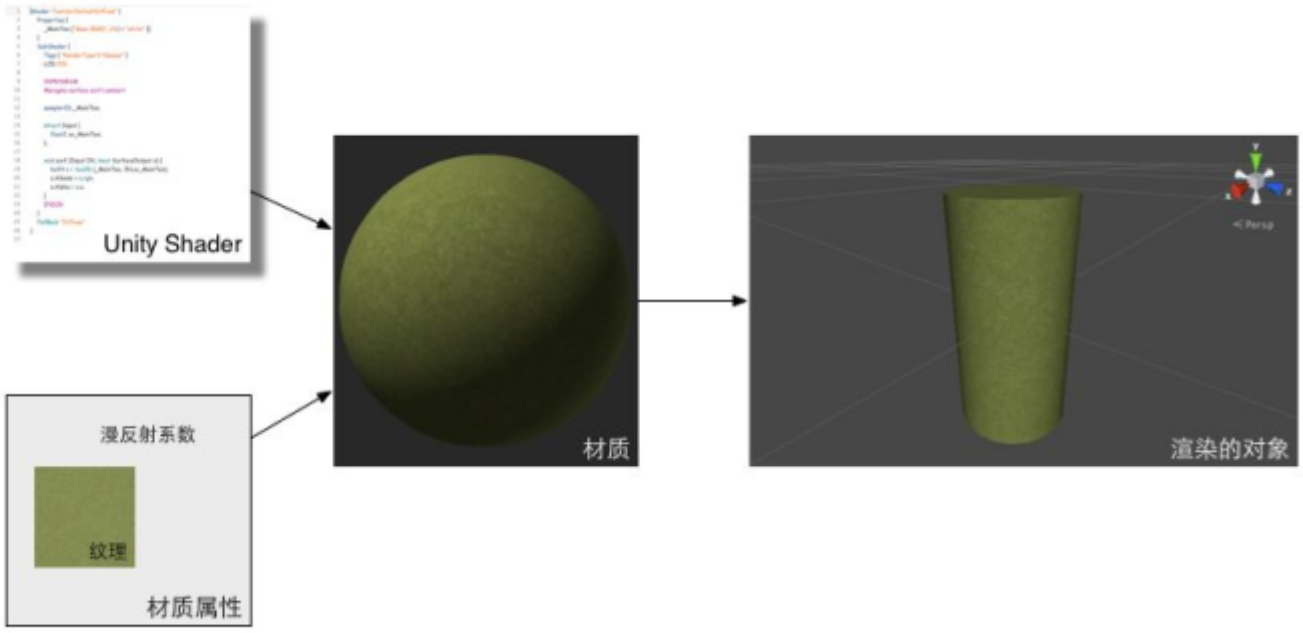
Lecture3 Unity Shader 基础

- Unity Shader 官方文档
- Unity Shader 着色器教程
- NVIDIA CG 文档
- NVIDIA CG 教程

1. 概述

材质和 Unity Shader

Unity 中我们需要配合使用**材质（Material）**和Unity Shader才能达到需要的效果，常见的流程是



1. 创建一个材质
2. 创建一个 Unity Shader，并把它赋给上一步中的材质
3. 把材质赋给要渲染的对象
4. 在材质面板中调整 Unity Shader 的属性，以达到满意效果

Unity 中的材质

Unity 中的材质需要结合一个 GameObject 的 Mesh 或者 Particle Systems 组件来工作，它决定了我们的游戏对象看起来是什么样子的，材质一般会赋给渲染对象的 Mesh Renderer 组件

Unity 中的 Shader

Unity 提供几种默认的 Unity Shader 模板

- Standard Surface Shader：标准光照模型
- Unlit Shader：不包含光照（但包含雾效）
- Image Effect Shader：屏幕后处理效果
- Compute Shader：特殊的 Shader 文件，进行一些与常规流水线无关的计算
    - http://docs.unity3d.com/Manual/ComputeShaders.html

Unity Shader 本质上就是一个文本文件，它也有导入设置（Import Settings）面板，在这个面板上，我们可以在 Default Maps 中指定该 Shader 使用的默认纹理

Shader Lab

Unity Shader是 Unity 为开发者提供的高层级的渲染抽象层，希望通过这种方式来让开发者更加轻松地控制渲染

- 在 Unity 中，所有的 Unity Shader 都是使用 **ShaderLab** 来编写的
- ShaderLab 是 Unity 提供的编写 Unity Shader 的一种**说明性语言**
- 它使用了一些嵌套在**花括号**内部的语义（syntax）

一个 Unity Shader 的基础结构如下所示

```
Shader "ShaderName"{
    Properties{
        // 属性
    }
    Subhader{
        // 显卡A使用的子着色器
    }
    SubShader{
        // 显卡B使用的子着色器
    }
    Fallback "VertexLit"
}
```

2. Unity Shader 的结构

名字

Unity Shader 文件的第一行可以通过 Shader 语义定义该 Unity Shader 的名字

```
Shader "Custom/MyShader" {
    //...
}
```

- 通过在字符串中添加斜杠，可以控制 Unity Shader 在材质面板出现的位置

属性 Properties：材质和 Unity Shader 的桥梁

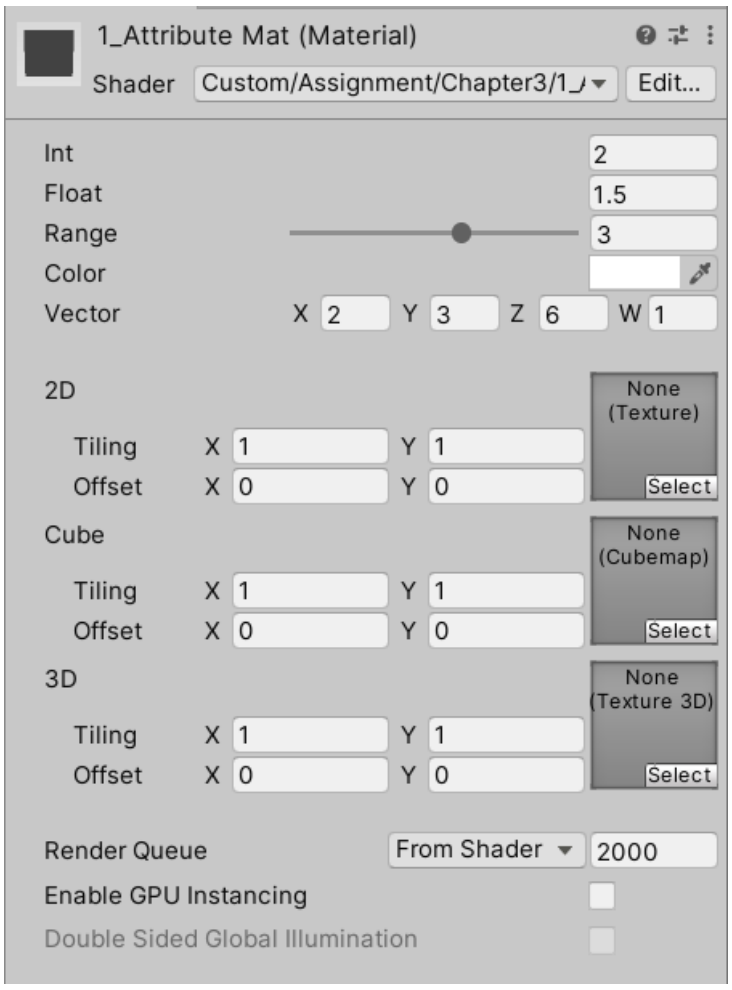
Properties 语义块中包含了一系列属性，这些属性会出现在材质面板中

```
Properties{
    Name {"display name", PropertyType} = DefaultValue;
    Name {"display name", PropertyType} = DefaultValue;
    // 更多属性
}
```

- Name：每个属性的名字（通常由一个**下划线**开始）
- Display Name：出现在材质面板上的名字
- PropertyType：属性的类型

属性常见类型

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" {}



为了在 Shader 中可以访问到这些属性，我们需要在 **CG 代码片段**中**定义**和这些属性类型相匹配的变量（即使我们不在 Properties 语义块中声明这些属性，也可以直接在 CG 代码片中定义比哪里）

- Properties 语义块的作用仅仅是为了让这些属性可以出现在材质面板中

SubShader

每一个 Unity Shader 文件可以包含多个 SubShader 语义块，但**最少要有一个**

当 Unity 需要加载这个 Unity Shader 时，Unity 会

- 扫描所有的 SubShader 语义块
- 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就会使用 Fallback 语义指定的 Unity Shader

SubShader 结构

```
SubShader{
    // 可选的
    {Tags}

    // 可选的
    {RenderSetup}

    Pass{

    }

    // Other Passes
}
```

SubShader 中定义了一系列 Pass 以及可选状态（RenderSetup）和标签（Tags），每个 Pass 定义了一次完整的渲染流程。如果 Pass 数量过多，则会造成渲染性能的下降。

Tags

Tags 是一个**键值对**，它告诉 Unity 的渲染引擎希望怎样以及合适渲染这个对象

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

标签类型	说明	例子
Queue	控制渲染顺序，**指定物体属于哪个渲染队列**，通过中各种方式可以保证所有透明物体在不透明物体后面被渲染，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" }

标签类型	说明	例子
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**，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**	Tags { "ForceNoShadowCasting" = "True" }
IgnoreProjector	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" }
CanUseSpriteAltas	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" }

- 上述标签仅可以在 SubShader 中声明，而不可以再 Pass 块中声明

RenderSetup

ShaderLab 提供了一系列渲染状态的设置指令，可以设置显卡各种状态

状态名称	指令	解释
Cull	CullBack | 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

当在 SubShader 块中设置了上述渲染状态时，会应用到所有的 Pass，如果我们想在 Pass 中运用不同的渲染设置时，可以在 Pass 语义块中单独设置

Pass

Pass 语义块包含语义如下

```
Pass{
    [Name]
    [Tags]
    [RenderSetup]
    // Other Code
}
```

- 首先，我们可以在 Pass 中定义该 Pass 的**名称**
  - `Name "MyPassName"`
- 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，提高代码的复用性
  - `UsePass "MyShader/MYPASSNAME"`
  - 需要注意的是，由于 Unity 内部会把所有 Pass 的名称转换成大写字母的表示，因此，在使用 UsePass 命令时必须使用**大写形式的名字**
- 其次，我们可以对 Pass 设置渲染状态，SubShader 中的状态设置同样适用于 Pass
  - 除了上面提到的状态设置外，在 Pass 中我们还可以使用**固定管线的着色器命令**
- Pass 同样可以设置**标签**，但它的标签不同于 SubShader 的标签

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"}

标签类型	说明	例子
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" }

- 除了上面普通的 Pass 定义外，Unity Shader 还支持一些特殊的 Pass，以便进行代码复用或实现更复杂的效果

除了上面的 Pass 之外，Unity Shader 还有一些特殊的 Pass

- **UsePasss**：使用该命令来复用 Unity Shader 中的 Pass
- **GrabPass**：负责抓取屏幕并将结果存储在一张纹理中，以用于后续的 Pass 处理

FallBack

- 如果上面的 SubShader 都不能在显卡上运行，那么使用一个低级的 Shader

```
Fallback "name"
// 或者
Fallback Off
```

3. Unity Shader 的形式

在 Unity 中，可以用三种形式来编写 Unity Shader，不论使用哪种，真正的 Shader 代码都包含在 SubShader 语义块中

```
Shader "MyShader"{
    Properties{
        // 所需的各种属性
    }
    SubShader{
        // 真正意义上的 Shader 代码
        // 表面着色器 or 顶点/片元着色器 or 固定函数着色器
    }
    SubShader{
        // 与上一个 SubShader 类似
    }
}
```

表面着色器 Surface Shader

Unity 自己创造的一种着色器代码类型

- 代码少
- 渲染代价较大

Unity 对顶点/片元着色器更高一层的抽象，处理了很多光照细节

```
Shader "Custom/Simple 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"
}
```

- 表面着色器被定义在 **SubShader 语义块**（而不是 Pass 语义块）

- 在 **CGPROGRAM** 和 **ENDCG** 之间，使用 **CG/HLSL** 编写，嵌套在 ShaderLab 中
  - CG/HLSL 经过 Unity 封装，与标准的 CG/HLSL 语法几乎一样，有细微不同

顶点/片元着色器 Vertex/Fragment Shader

- 更复杂
- 更灵活

```
Shader "Custom/Simple 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
        }
    }
}
```

- 顶点/片元着色器被定义在 **Pass 语义块**
- 在 **CGPROGRAM** 和 **ENDCG** 之间，使用 **CG/HLSL** 编写

固定函数着色器 Fixed Function Shader

- 不支持可编程渲染管线
- 只能完成非常简单的效果

```
Shader "Custom Fixed Function Shader"{
    Properties{
        _Color {"Main Color", Color} = {1, 0.5, 0.5, 1}
    }
    SubShader{
        Pass{
            Material{
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

- 完全使用 **ShaderLab 的语法**，而非 CG/HLSL

4. 释疑

Unity Shader ≠ Shader

优点

传统 Shader	Unity Shader (Shader Lab)
仅可以编写特定类型的 Shader，比如顶点着色器或者片元着色器	同一个文件里可以同时包含顶点着色器和片元着色器代码
无法设置一些渲染设置，而是开发者在另外的代码里设置的	通过一行特定的指令即可完成
需要编写冗长的代码来处理着色器的输入输出	在特定的语句块中声明一些属性，就可以靠材质来改变属性，对于模型的自带数据（顶点、位置、纹理坐标、法线等）Unity Shader 也提供直接访问的方法

缺点

- 高度封装性
- 不支持更多精细的 Shader，比如曲面细分着色器或几何着色器
- 不支持更高级的语法

可以使用 GLSL 吗

- 可以，把代码嵌套在 GLSLPROGRAM 和 ENGGLSL 中即可