Unity ShaderLab 学习总结

黄和燕

September 28, 2018

Contents

1 Unity 渲染路径（Rendering Path）种类

1.1 概述

开发者可以在 Unity 工程的 PlayerSettings 设置对渲染路径进行 3 选 1：

- Deferred Lighting，延迟光照路径。3 者中最高质量地还原光照阴影。光照性能只与最终像素数目有关，光源数量再多都不会影响性能。
  - 不关心有多少个光源会影响它，每个物体一般都会绘制两次；类似地，Vertex Lit 只绘制一次。所以对于这两种类型来说，shader 对表现效果的改变大多在于多重纹理方面。
- Forward Rendering，顺序渲染路径。能发挥出 Shader 全部特性的渲染路径，当然也就支持像素级光照。最常用、功能最自由，性能与光源数目 * 受光照物体数目有关，具体性能视乎其具体使用到的 Shader 的复杂度。Forward 通道类型的表现效果要取决于 shader 和场景中的光源。它有两种基本的计算方式 Vertex-Lit 和 Pixel-Lit。可以翻译为逐顶点渲染法和逐像素渲染法吧应该，对应着 D3D 中的顶点着色和像素着色过程。
  - Vertex-Lit 用于对网格模型表面顶点进行光照计算，一次性将所有光源的影响都计算在内，所以无论场景中有多少个光源，这种方式绘制的物体只绘制一次。
  - Pixel-Lit 会计算每个像素上面最终的光照，因此一个物体必须先呗绘制一次来获得环境光和主方向光的光照信息，再绘制一次来获得其他每个额外的光源信息。应用 Pixel-Lit 的物体的大小也会影响绘制的效率，越大的物体，绘制越慢。
  - Vertex-Lit 的开销大于 Pixel-Lit，但是 Pixel-Lit 可以提供很多非常好的效果。
- Vertex Lit，顶点光照路径。顶点级光照。性能最高、兼容性最强、支持特性最少、品质最差。

1.2 渲染路径的内部阶段和 Pass 的 LightMode 标签

- 每个渲染路径的内部会再分为几个阶段。
- 然后，Shader 里的每个 Pass，都可以指定为不同的 LightMode。而 LightMode 实际就是说：“我希望这个 Pass 在这个 XXX 渲染路径的这个 YYY 子阶段被执行”。

1.2.1 Deferred Ligting

渲染路径内部子阶段	对应的 LightMode	描述
Base Pass	”PrepassBase”	渲染物体信息。即把法向量、高光度到一张 ARGB32 的物体信息纹理上，把深度信息保存在 Z-Buff 上。
Lighting Pass	无对应可编程 Pass	根据 Base Pass 得出的物体信息，在屏幕坐标系下，使用 BlinnPhong 光照模式，把光照信息渲染到 ARGB32 的光照信息纹理上（RGB 表示 diffuse 颜色值、A 表示高光度）
Final Pass	”PrepassFinal”	根据光照信息纹理，物体再渲染一次，将光照信息、纹理信息和自发光信息最终混合。LightMap 也在这个 Pass 进行。

1.2.2 Forward Rendering

渲染路径内部子阶段	对应的 LightMode	描述
Base Pass	”ForwardBase”	渲染：最亮一个的方向光光源（像素级）和对应的阴影、所有顶点级光源、LightMap、所有 LightProbe 的 SH 光源（Sphere Harmonic，球谐函数，效率超高的低频光）、环境光、自发光。
Additional Passes	”ForwardAdd”	其他需要像素级渲染的的光源

- 注意到的是，在 Forward Rendering 中，光源可能是像素级光源、顶点级光源或 SH 光源。其判断标准是：
  - 配制成“Not Important”的光源都是顶点级光源和 SH 光源
  - 最亮的方向光永远都是像素级光源
  - 配置成“Important”的都是像素级光源
  - 上面 2 种情况加起来的像素级光源数目小于“Quality Settings”里面的“Pixel Light Count”的话，会把第 1 种情况的光源补为额外的像素级光源。
- 另外，配置成“Auto”的光源有更复杂的判断标注，截图如下：
- 具体可参考 Forward Rendering Path Details。

1.2.3 Vertex Lit

渲染路径内部子阶段	对应的 LightMode	描述
Vertex	”Vertex”	渲染无 LightMap 物体
VertexLMRGBM	”VertexLMRGBM”	渲染有 RGBM 编码的 LightMap 物体
VertexLM	”VertexLM”	渲染有双 LDR 编码的 LightMap 物体

1.2.4 不同 LightMode 的 Pass 的被选择

- 一个工程的渲染路径是唯一的，但一个工程里的 Shader 是允许配有不同 LightMode 的 Pass 的。
- 在 Unity，策略是“从工程配置的渲染路径模式开始，按 Deferred、Forward、VertxLit 的顺序，搜索最匹配的 LightMode 的一个 Pass”。
- 比如，在配置成 Deferred 路径时，优先选有 Deferred 相关 LightMode 的 Pass；找不到才会选 Forward 相关的 Pass；还找不到，才会选 VertexLit 相关的 Pass。
- 再比如，在配置成 Forward 路径时，优先选 Forward 相关的 Pass；找不到才会选 VertexLit 相关的 Pass。

2 Shader 基础猫都能学会的 Unity3D Shader 入门指南（一）

- https://onevcat.com/2013/07/shader-tutorial-1/

2.1 属性

- 在 Properties{} 中定义着色器属性，在这里定义的属性将被作为输入提供给所有的子着色器。每一条属性的定义的语法是这样的：

```
_Name("Display Name", type) = defaultValue[{options}]
```

- _Name - 属性的名字，简单说就是变量名，在之后整个 Shader 代码中将使用这个名字来获取该属性的内容
- Display Name - 这个字符串将显示在 Unity 的材质编辑器中作为 Shader 的使用者可读的内容
- type - 这个属性的类型，可能的 type 所表示的内容有以下几种：
  - Color - 一种颜色，由 RGBA（红绿蓝和透明度）四个量来定义；
  - 2D - 一张 2 的阶数大小（256，512 之类）的贴图。这张贴图将在采样后被转为对应基于模型 UV 的每个像素的颜色，最终被显示出来；
  - Rect - 一个非 2 阶数大小的贴图；
  - Cube - 即 Cube map texture（立方体纹理），简单说就是 6 张有联系的 2D 贴图的组合，主要用来做反射效果（比如天空盒和动态反射），也会被转换为对应点的采样；
  - Range(min, max) - 一个介于最小值和最大值之间的浮点数，一般用来当作调整 Shader 某些特性的参数（比如透明度渲染的截止值可以是从 0 至 1 的值等）；
  - Float - 任意一个浮点数；
  - Vector - 一个四维数；
- defaultValue 定义了这个属性的默认值，通过输入一个符合格式的默认值来指定对应属性的初始值（某些效果可能需要某些特定的参数值来达到需要的效果，虽然这些值可以在之后在进行调整，但是如果默认就指定为想要的值的话就省去了一个个调整的时间，方便很多）。
  - Color - 以 0 ～ 1 定义的 rgba 颜色，比如 (1,1,1,1)；
  - 2D/Rect/Cube - 对于贴图来说，默认值可以为一个代表默认 tint 颜色的字符串，可以是空字符串或者”white”,”black”,”gray”,”bump”中的一个
  - Float，Range - 某个指定的浮点数
  - Vector - 一个 4 维数，写为 (x,y,z,w)
- 另外还有一个 {option}，它只对 2D，Rect 或者 Cube 贴图有关，在写输入时我们最少要在贴图之后写一对什么都不含的空白的 {}，当我们需要打开特定选项时可以把其写在这对花括号内。如果需要同时打开多个选项，可以使用空白分隔。可能的选择有 ObjectLinear, EyeLinear, SphereMap, CubeReflect, CubeNormal 中的一个，这些都是 OpenGL 中 TexGen 的模式，具体的留到后面有机会再说。
- 所以，一组属性的申明看起来也许会是这个样子的

```
//Define a color with a default value of semi-transparent blue
_MainColor ("Main Color", Color) = (0,0,1,0.5)
//Define a texture with a default of white
_Texture ("Texture", 2D) = "white" {}
```

- 现在看懂上面那段 Shader（以及其他所有 Shader）的 Properties 部分应该不会有任何问题了。接下来就是 SubShader 部分了。

2.2 SubShader 的 Tags

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader {
        Tags { "Queue"="Geometry+10" "RenderType"="Opaque" }
        //...
    }
}
```

- SubShader 内部可以有标签（Tags）的定义。Tag 指定了这个 SubShader 的渲染顺序（时机），以及其他的一些设置。

2.2.1 "RenderType" 标签

- 表面着色器可以被若干的标签（tags）所修饰，而硬件将通过判定这些标签来决定什么时候调用该着色器。比如我们的例子中 SubShader 的第一句

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

- 告诉了系统应该在渲染非透明物体时调用我们。Unity 定义了一些列这样的渲染过程，与 RenderType 是 Opaque 相对应的显而易见的是"RenderType" = "Transparent"，表示渲染含有透明效果的物体时调用。在这里 Tags 其实暗示了你的 Shader 输出的是什么，如果输出中都是非透明物体，那写在 Opaque 里；如果想渲染透明或者半透明的像素，那应该写在 Transparent 中。

- 另外比较有用的标签还有"IgnoreProjector"="True"（不被 Projectors 影响），"ForceNoShadowCasting"="True（从不产生阴影）以及"Queue"="xxx"（指定渲染顺序队列）。这里想要着重说一下的是 Queue 这个标签，如果你使用 Unity 做过一些透明和不透明物体的混合的话，很可能已经遇到过不透明物体无法呈现在透明物体之后的情况。这种情况很可能是由于 Shader 的渲染顺序不正确导致的。Queue 指定了物体的渲染顺序，预定义的 Queue 有：

- Unity 可以运行时替换符合特定 RenderType 的所有 Shader。Camera.RenderWithShader 或 Camera.SetRepla 配合使用。Unity 内置的 RenderType 包括：

  - **Background** - 最早被调用的渲染，用来渲染天空盒或者背景
  - **Geometry** - 这是默认值，用来渲染非透明物体（普通情况下，场景中的绝大多数物体应该是非透明的）
  - **AlphaTest** - 用来渲染经过 Alpha Test 的像素，单独为 AlphaTest 设定一个 Queue 是出于对效率的考虑
  - **Opaque** - 绝大部分不透明的物体都使用这个；
  - **Transparent** - 以从后往前的顺序渲染透明物体，绝大部分透明的物体、包括粒子特效都使用这个；
  - **Overlay** - 用来渲染叠加的效果，是渲染的最后阶段（比如 GUI?、镜头光晕等特效）

- 这些预定义的值本质上是一组定义整数，

  - Background = 1000，
  - Geometry = 2000，
  - AlphaTest = 2450，
  - Transparent = 3000，最后
  - Overlay = 4000。

- 在我们实际设置 Queue 值时，不仅能使用上面的几个预定义值，我们也可以指定自己的 Queue 值，写成类似这样："Queue"="Transparent+100"，表示一个在 Transparent 之后 100 的 Queue 上进行调用。通过调整 Queue 值，我们可以确保某些物体一定在另一些物体之前或者之后渲染，这个技巧有时候很有用处。u

- 用户也可以定义任意自己的 RenderType 这个标签所取的值。

- 应注意，Camera.RenderWithShader 或 Camera.SetReplacementShader 不要求标签只能是 RenderType，RenderType 只是 Unity 内部用于 Replace 的一个标签而已，你也可以自定义自己全新的标签用于 Replace。

- 比如，你为自己的 ShaderA.SubShaderA1（会被 Unity 选取到的 SubShader，常为 Shader 文件中的第一个 SubShader）增加 Tag 为"Distort"="On"，然后将"Distort" 作为参数 replacementTag 传给函数。此时，作为 replacementShader 实参的 ShaderB.SubShaderB1 中若有也有一模一样的"Distort"="On"，则此 SubShaderB1 将代替 SubShaderA1 用于本次渲染。

- 具体可参考 Rendering with Replaced Shaders https://docs.unity3d.com/Manual/SL-ShaderReplacement html

2.2.2 "Queue" 标签

- 定义渲染顺序。预制的值为
  - "Background"。值为 1000。比如用于天空盒。
  - "Geometry"。值为 2000。大部分物体在这个队列。不透明的物体也在这里。这个队列内部的物体的渲染顺序会有进一步的优化（应该是从近到远，early-z test 可以剔除不需经过 FS 处理的片元）。其他队列的物体都是按空间位置的从远到近进行渲染。
  - "AlphaTest"。值为 2450。已进行 AlphaTest 的物体在这个队列。
  - "Transparent"。值为 3000。透明物体。
  - "Overlay"。值为 4000。比如镜头光晕。
- 用户可以定义任意值，比如"Queue"="Geometry+10"
- "ForceNoShadowCasting"，值为"true" 时，表示不接受阴影。
- "IgnoreProjector"，值为"true" 时，表示不接受 Projector 组件的投影。
- 另，关于渲染队列和 Batch 的非官方经验总结是，一帧的渲染队列的生成，依次决定于每个渲染物体的：
  - Shader 的 RenderType tag,
  - Renderer.SortingLayerID,
  - Renderer.SortingOrder,
  - Material.renderQueue（默认值为 Shader 里的"Queue"），
  - Transform.z(ViewSpace)（默认为按 z 值从前到后，但当 Queue 是"Transparent"的时候，按 z 值从后到前）。
  - 这个渲染队列决定了之后（可能有 dirty flag 的机制?）渲染器再依次遍历这个渲染队列，"同一种"材质的渲染物体合到一个 Batch 里。
- 另，UGUI 物体的渲染顺序 (同一个 canvas 下的?)
  - 不同 Camera 的 Depth。（大在前，小在后）
  - 同 Camera 的 SortingLayer。（下在前，上在后）
  - 同 SortingLayer 下的 Order in Layer。（大在前，小在后）
  - 同 Order in Layer 下的 Z 轴。（小在前，大在后）
- 另,如果是多个 Canvas 的渲染先后顺序 http://blog.csdn.net/huutu/article/details/43636241 Unity3
中的渲染顺序如下：
  - 不同 Camera 的 Depth
  - 同 Camera 的 soringLayer
  - 同 sortingLayer 下面的 Z 轴
  - 不同 Canvas 的 Sort Order 值，调 Canvas 下面有一个 Sort Order 值，默认为 0，越大越在后面。（顺序应该在最前还是最后???）
  - 创建任意 UGUI 元素时自动生成一个 Canvas 物体，Canvas 下的所有物体从上往下渲染，即排在下面的会遮盖排上面的。同理，子元素会覆盖父元素。
- 另，在游戏运行中如何修改 UGUI 的显示层级? 在代码中调整该元素的层级位：使用 RectTransform 类的函数。
  - SetAsFirstSibling：移动到所有兄弟节点的第一个位置（Hierarchy 同级最上面，先渲染，显示在最下面）
  - SetAsLastSibling：移动到所有兄弟节点的最后一个位置（Hierarchy 同级最下面，后渲染，显示在最上面）
  - GetSiblingIndex：获得该元素在当前兄弟节点层级的位置
  - SetSiblingIndex：设置该元素在当前兄弟节点层级的位置

2.3 LOD

- LOD 很简单，它是 Level of Detail 的缩写，在这里例子里我们指定了其为 200（其实这是 Unity 的内建 Diffuse 着色器的设定值）。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

2.4 Pass

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader {
        Pass {
            //...
        }
    }
}
```

- 一个 SubShader（渲染方案）是由一个个 Pass 块来执行的。每个 Pass 都会消耗对应的一个 DrawCall。在满足渲染效果的情况下尽可能地减少 Pass 的数量。

2.4.1 Pass 的 Tag

```
Shader "ShaderLab Tutorials/TestShader" {
    SubShader {
        Pass {
            Tags {"LightMode"="ForwardBase"}
            //...
        }
    }
}
```

- 和 SubShader 有自己专属的 Tag 类似，Pass 也有 Pass 专属的 Tag。
- 其中最重要 Tag 是"LightMode"，指定 Pass 和 Unity 的哪一种渲染路径（"Rendering Path"）搭配使用。除最重要的 ForwardBase、ForwardAdd 外，这里需额外提醒的 Tag 取值可包括：
  - Always，永远都渲染，但不处理光照
  - ShadowCaster，用于渲染产生阴影的物体
  - ShadowCollector，用于收集物体阴影到屏幕坐标 Buff 里。
- 其他渲染路径相关的 Tag 详见下面章节"Unity 渲染路径种类"。
- 具体所有 Tag 取值，可参考 ShaderLab syntax: Pass Tags。

2.4.2 FallBack

```
Shader "ShaderLab Tutorials/TestShader"{
    SubShader { Pass {} }
    FallBack "Diffuse" // "Diffuse" 即 Unity 预制的固有 Shader
    // FallBack Off //将关闭 FallBack
}
```

- 当本 Shader 的所有 SubShader 都不支持当前显卡，就会使用 FallBack 语句指定的另一个 Shader。FallBack 最好指定 Unity 自己预制的 Shader 实现，因其一般能够在当前所有显卡运行。

2.4.3 Properties

```
Shader "ShaderLab Tutorials/TestShader" {
    Properties {
        _Range ("My Range", Range (0.02,0.15)) = 0.07 // sliders
        _Color ("My Color", Color) = (.34, .85, .92, 1) // color
        _2D ("My Texture 2D", 2D) = "" {} // textures
        _Rect("My Rectangle", Rect) = "name" { }
        _Cube ("My Cubemap", Cube) = "name" { }
        _Float ("My Float", Float) = 1
        _Vector ("My Vector", Vector) = (1,2,3,4)

        // Display as a toggle.
        [Toggle] _Invert ("Invert color?", Float) = 0
        // Blend mode values
        [Enum(UnityEngine.Rendering.BlendMode)] _Blend ("Blend mode", Float) = 1
        //setup corresponding shader keywords.
        [KeywordEnum(Off, On)] _UseSpecular ("Use Specular",  Float) = 0
    }
    // SubShader
    SubShader {
        Pass {
            uniform float4 _Color;
            float4 frag() : COLOR {
                return fixed4(_Color);
            }
            #pragma multi_compile __ _USESPECULAR_ON
        }
    }
    // fixed pipeline
    SubShader {
        Pass {
            Color[_Color]
        }
    }
}
```

- Shader 在 Unity 编辑器暴露给美术的参数，通过 Properties 来实现。
- 所有可能的参数如上所示。主要也就 Float、Vector 和 Texture 这 3 类。
- 除了通过编辑器编辑 Properties，脚本也可以通过 Material 的接口（比如 SetFloat、SetTexture 编辑）
- 之后在 Shader 程序通过 [name]（固定管线）或直接 name（可编程 Shader）访问这些属性。
- 在每一个 Property 前面也能类似 C# 那样添加 Attribute，以达到额外 UI 面板功能。详见 MaterialPropertyDrawer.html。

2.5 Shader 中的数据类型

- 有 3 种基本数值类型：float、half 和 fixed。
- 这 3 种基本数值类型可以再组成 vector 和 matrix，比如 half3 是由 3 个 half 组成、float4x4 是由 16 个 float 组成。
  - float：32 位高精度浮点数。
  - half：16 位中精度浮点数。范围是 [-6 万, +6 万]，能精确到十进制的小数点后 3.3 位。
  - fixed：11 位低精度浮点数。范围是 [-2, 2]，精度是 1/256。
- 数据类型影响性能
  - 精度够用就好。
  - 颜色和单位向量，使用 fixed
  - 其他情况，尽量使用 half（即范围在 [-6 万, +6 万] 内、精确到小数点后 3.3 位）；否则才使用 float。

2.6 ShaderLab 中的 Matrix

- 当提到“Row-Major”、“Column-Major”，根据不同的场合，它们可能指不同的意思：
  - 数学上的，主要是指矢量 V 是 Row Vector、还是 Column Vector。引用自 [Game Engine Architecture 2nd Edition, 183]。留意到 V 和 M 的乘法，当是 Row Vector 的时候，数学上写作 VM，Matrix 在右边，Matrix 的最下面一行表示 Translate；当是 Column Vector 的时候，数学上写作 MtVt，Matrix 在左边并且需要转置，Matrix 最右面一列表示 Translate。
  - 访问接口上的：Row-Major 即 MyMatrix[Row][Column]、Column-Major 即 MyMatrix[Column][Row]。HLSL/CG 的访问接口都是 Row-Major，比如 MyMatrix[3] 返回的是第 3 行；GLSL 的访问接口是 Column-Major，比如 MyMatrix[3] 返回的是第 3 列。
  - 寄存器存储上的：每个元素是按行存储在寄存器中、还是按列存储在寄存器中。需要关注它的一般情况举例是，float2x3 的 MyMatrix，到底是占用 2 个寄存器（Row-Major）、还是 3 个寄存器（Column-Major）。在 HLSL 里，可以通过 #pragma pack$_{matrixrowmajorcolumnmajor}$
- 上述情况，互不相干。
- 然后，ShaderLab 中，数学上是 Column Vector、访问接口上是 Row-Major、存储上是（尚未查明）。

2.7 ShaderLab 中各个 Space 的坐标系

一般情况下，从 Vertex Buff 输入顶点到 Vertex Shader，

- 该顶点为左手坐标系 Model Space 中的顶点 vInModel，
- 其用 w=1 的 Homogenous Cooridniates（故等效于 Cartesian Coordinates）表达 vInModel = float4(xm, ym, zm, 1)；
- vInWrold = mul($_{Object2World}$, vInModel) 后，得出左手坐标系 World Space 中的 vInWorld，其为 w=1 的 Homogenous Cooridniates（故等效于 Cartesian Coordinates）vInWorld = float4(xw, yw, zw, 1)；
- vInView = mul(UNITY$_{MATRIXV}$, vInWrold) 后，得出右手坐标系 View Space 中的 vInView，其为 w=1 的 Homogenous Cooridniates（故等效于 Cartesian Coordinates）vInWorld = float4(xv, yv, zv, 1)；
- vInClip = mul(UNITY$_{MATRIXP}$, vInView) 后，得出左手坐标系 Clip Space 中的 vInClip，其为 w 往往不等于 1 的 Homogenous Cooridniates（故往往不等效于 Cartesian Coordinates）vInClip = float4(xc, yc, zc, wc)；

设 r、l、t、b、n、f 的长度绝对值如下图：

注意 View Space 中摄像机前方的 z 值为负数、-z 为正数。则 GL/DX/Metal 的 Clip Space 坐标为：

- GL:
  - xc=(2nx+rz+lz)/(r-l);
  - yc=(2ny+tz+bz)/(t-b);

  - zc=(-fz-nz-2nf)/(f-n);
  - wc=-z;
- DX/Metal:
  - xc=(2nx+rz+lz)/(r-l);
  - yc=(2ny+tz+bz)/(t-b);
  - zc=(-fz-nf)/(f-n);
  - wc=-z;
- vInNDC = vInClip / vInClip.w 后，得出左手坐标系 Normalized Device Coordinates 中的 vInNDC，其为 w=1 的 Homogenous Cooridniates（故等效于 Cartesian Coordinates）vInNDC = float4(xn, yn, zn, 1)。xn 和 yn 的取值范围为 [-1,1]。
  - GL: zn=zc/wc=(fz+nz+2nf)/((f-n)z);
  - DX/Metal: zn=zc/wc=(fz+nf)/((f-n)z);
  - 在 Unity 中，zn 的取值范围可以这样决定：
    * 如果 $\text{UNITY}_{REVERSEDZ}$ $zn[UNITY\,NEARCLIPVALUE, 0]$，即 [1,0]
    * 如果 $\text{UNITY}_{REVERSEDZ}$ $zn[UNITY\,NEARCLIPVALUE, 1]$
      · 如果 $\text{SHADER}_{API\,D3D9}$/$\text{SHADER}_{API\,D3D11\,9X}$ [0,1]
      · 否则，即 OpenGL 情况，即 [-1,1]

```
v2f vert (appdata v) {
    v2f o;
    o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
    // 1 、2、3 是等价的，和 4 是不等价的
    // 因为是 M 在左、V 在右，所以是 Column Vector
    // 因为是 HLSL/CG 语言，所以是访问方式是 Row-Major
    o.rootInView = mul(UNITY_MATRIX_MV, float4(0, 0, 0, 1)); // 1
    o.rootInView = float4(UNITY_MATRIX_MV[0].w, UNITY_MATRIX_MV[1].w, UNITY_MATRIX_MV[
    o.rootInView = UNITY_MATRIX_MV._m03_m13_m23_m33;  // 3
    //o.rootInView = UNITY_MATRIX_MV[3]; // 4
    return o;
}

fixed4 frag (v2f i) : SV_Target {
    // 因为是 ViewSpace 是右手坐标系，所以当 root 在 view 前面的时候，z 是负数，所以需要-
    fixed4 col = fixed4(i.rootInView.x, i.rootInView.y, -i.rootInView.z, 1);
    return col;
}
struct appdata {
    float4 vertex : POSITION;
};
struct v2f {
    float4 rootInView : TEXCOORD0;
    float4 vertex : SV_POSITION;
};
```

3 Shader 形态

3.1 Shader 形态之 1：固定管线

- 固定管线是为了兼容老式显卡。都是顶点光照。之后固定管线可能是被 Unity 抛弃的功能，所以最好不学它、当它不存在。特征是里面出现了形如下面 Material 块、没有 CGPROGRAM 和 ENDCG 块。

```
Shader "ShaderLab Tutorials/TestShader" {
    Properties {
        _Color ("My Color", Color) = (.34, .85, .92, 1) // color
```

```
    }
    // Fixed Pipeline
    SubShader {
        Pass {
            Material {
                Diffuse [_Color]
                Ambient [_Color]
            }
            Lighting On
        }
    }
}
```

3.2 Shader 形态之 2：可编程 Shader

```
Shader "ShaderLab Tutorials/TestShader" {
    Properties { }
    SubShader {
        Pass {
            // ... the usual pass state setup ...
            CGPROGRAM
            // compilation directives for this snippet, e.g.:
            #pragma vertex vert
            #pragma fragment frag

            // the Cg/HLSL code itself
            float4 vert(float4 v:POSITION) : SV_POSITION {
                return mul(UNITY_MATRIX_MVP, v);
            }
            float4 frag() : COLOR {
                return fixed4(1.0, 0.0, 0.0, 1.0);
            }
            ENDCG
            // ... the rest of pass setup ...
        }
    }
}
```

- 功能最强大、最自由的形态。
- 特征是在 Pass 里出现 CGPROGRAM 和 ENDCG 块
- 编译指令 #pragma。详见官网 Cg snippets。其中重要的包括：

编译指令	示例/含义
#pragma vertex name #pragma fragment name	替换 name，来指定 Vertex Shader 函数、Fragment Shader 函数。
#pragma target name	替换 name（为 2.0、3.0 等）。设置编译目标 shader model 的版本。
#pragma only$_{renderers}$ name name ... #pragma exclude$_{renderers}$ name name...	#pragma only$_{renderers}$ gles gles3， #pragma exclude$_{renderers}$ d3d9 d3d11 opengl， 只为指定渲染平台（render platform）编译

- 引用库。通过形如 #include "UnityCG.cginc" 引入指定的库。常用的就是 UnityCG.cginc 了。其他库详见官网 Built-in shader include files。
- ShaderLab 内置值。Unity 给 Shader 程序提供了便捷的、常用的值，比如下面例子中的 UNITY$_{MATRIX MVPMV}$ built-in values。
- Shader 输入输出参数语义（Semantics）。在管线流程中每个阶段之间（比如 Vertex Shader 阶段和 FragmentShader 阶段之间）的输入输出参数，通过语义字符串，来指定参数的含义。常用的语义包括：COLOR、SV$_{Position\,TEXCOORD[n]\,HLSL}$ Semantic（由于是 HLSL 的连接，所以可能不完全在 Unity 里可以使用）。

- 特别地，因为 Vertex Shader 的的输入往往是管线的最开始，Unity 为此内置了常用的数据结构：

数据结构	含义
appdata_{base}	vertex shader input with position, normal, one texture coordinate.
appdata_{tan}	vertex shader input with position, normal, tangent, one texture coordinate.
appdata_{full}	vertex shader input with position, normal, tangent, vertex color and two texture coordinates.
appdata_{img}	vertex shader input with position and one texture coordinate.

3.3 Shader 形态之 3：SurfaceShader

```
Shader "ShaderLab Tutorials/TestShader" {
    Properties {   }
    // 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"
}
```

- SurfaceShader 可以认为是一个光照 Shader 的语法糖、一个光照 VS/FS 的生成器。减少了开发者写重复代码的需要。
- 在手游，由于对性能要求比较高，所以不建议使用 SurfaceShader。因为 SurfaceShader 是一个比较“通用”的功能，而通用往往导致性能不高。
- 特征是在 SubShader 里出现 CGPROGRAM 和 ENDCG 块。（而不是出现在 Pass 里。因为 SurfaceShader 自己会编译成多个 Pass。）
- 编译指令是：

```
#pragma surface surfaceFunction lightModel [optionalparams]
```

  - surfaceFunction：surfaceShader 函数，形如 void surf (Input IN, inout SurfaceOutput o)
  - lightModel：
    * 使用的光照模式。包括 Lambert（漫反射）和 BlinnPhong（镜面反射）。
    * 也可以自己定义光照函数。比如编译指令为 #pragma surface surf MyCalc
    * 在 Shader 里定义 half4 LightingMyCalc (SurfaceOutput s, 参数略) 函数进行处理 (函数名在签名加上了“Lighting”)。
    * 详见 Custom Lighting models in Surface Shaders
    * 你定义输入数据结构（比如上面的 Input）、编写自己的 Surface 函数处理输入、最终输出修改过后的 SurfaceOutput。
    * SurfaceOutput 的定义为

```
struct SurfaceOutput {
  half3 Albedo;     // 纹理颜色值 (r, g, b)
  half3 Normal;     // 法向量 (x, y, z)
  half3 Emission;   // 自发光颜色值 (r, g, b)
  half Specular;    // 镜面反射度
  half Gloss;       // 光泽度
  half Alpha;       // 不透明度
};
```

3.4 Shader 形态之 4：Compiled Shader

- 点击 a.shader 文件的“Compile and show code”，可以看到该文件的“编译”过后的 ShaderLab shader 文件，文件名形如 Compiled-a.shader。
- 其依然是 ShaderLab 文件，其包含最终提交给 GPU 的 shader 代码字符串。
- 先就其结构进行简述如下，会发现和上述的编译前 ShaderLab 结构很相似。

```
// Compiled shader for iPhone, iPod Touch and iPad, uncompressed size: 36.5KB
// Skipping shader variants that would not be included into build of current scene.
Shader "ShaderLab Tutorials/TestShader" {
    Properties {...}
    SubShader {
        // Stats for Vertex shader:
        //        gles : 14 avg math (11..19), 1 avg texture (1..2)
        //       metal : 14 avg math (11..17)
        // Stats for Fragment shader:
        //       metal : 14 avg math (11..19), 1 avg texture (1..2)
        Pass {
            Program "vp" { // vertex program
                SubProgram "gles" {
                    // Stats: 11 math, 1 textures
                    Keywords {...} // keywords for shader variants ("uber shader")

                    //shader codes in string
                    "
                    #ifdef VERTEX
                    vertex shader codes
                    #endif

                    // Note, on gles, fragment shader stays here inside Program vp
                    #ifdef FRAGMENT
                    fragment shader codes
                    #endif
                    "
                }
                SubProgram "metal"  {
                    some setup
                    Keywords {...}
                    //vertex shader codes in string
                    "..."
                }
            }
            Program "fp" {// fragment program
                SubProgram "gles" {
                    Keywords {...}
                    "// shader disassembly not supported on gles"
                    //(because gles fragment shader codes are in Program "vp")
                }
                SubProgram "metal" {
                    common setup
                    Keywords {...}
                    //fragment shader codes in string
                    "..."
                }
            }
        }
    }
    ...
```

```
52 }
```

4 表面着色器的写法

- http://www.unity.5helpyou.com/2381.html

4.1 表面着色器的编译指令

- 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 (lighting 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。

4.2 表面着色器输入结构（Input Structure）

- 表面着色器书写的第三个要素是指明表面输入结构（Input Structure）。
- Input 这个输入结构通常拥有着色器需要的所有纹理坐标 (texture coordinates)。纹理坐标 (Texturecoordinates) 必须被命名为“uv”后接纹理 (texture) 名字。(或者 uv2 开始，使用第二纹理坐标集)。
- 可以在输入结构中根据自己的需要，可选附加这样的一些候选值：
  - 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) 不为 SurfaceOutput 结构中的 Normal 赋值，也就是说 Normal 不会发生变化，也就不需要重新求取 worldRefl 值了，那么就可以直接通过 Input 结构体传递该参数。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。

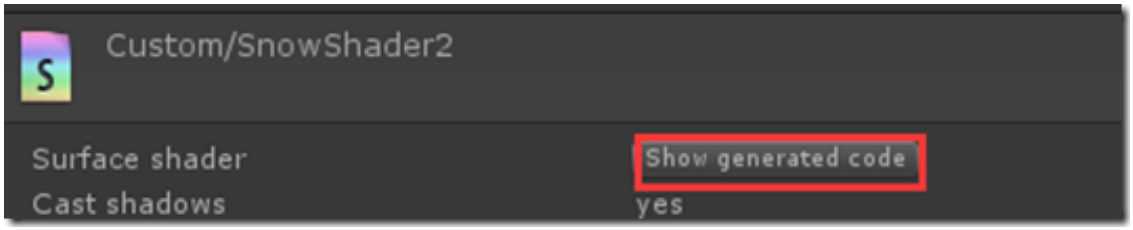
- float3 worldNormal – 世界空间中的法线向量 (normal vector)。如果表面着色器 (surface shader) 不为 SurfaceOutput 结构中的 Normal 赋值，也就是说 Normal 不会发生变化，也就不需要重新求取 worldNormal 值了，那么就可以直接通过 Input 结构体传递该参数。如果表面着色器 (surface shader) 不写入法线 (o.Normal) 参数，将包含这个参数。
- float3 worldRefl; INTERNAL$_{DATA}$ – 世界空间中的反射向量。如果表面着色器 (surface shader) 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))。
- INTERNAL$_{DATA}$ – 相对于上面的 float3 worldRefl 和 float3 worldNormal，如果表面着色器为 SurfaceOutput 结构中的 Normal 赋值了，比如在 surf 函数中使用了 o.Normal = ···; 此时表面着色器的法向值发生了改变，因此我们就只能借助赋值后的 o.Normal 来对世界坐标系下的反射向量进行修改。借助 Input 结构体我们传递 worldRefl 参数，并且我们用内置的 INTERNAL$_{DATA}$ (WorldReflectionVector (IN, o.Normal)) 计算世界坐标系下的反射向量，其中 o.Normal 表示的是切空间的法向量，而非世界坐标系下的法向量。

4.2.1 关于 INTERNAL$_{DATA}$

- 为了更清楚的弄懂 INTERNAL$_{DATA\ shader}$#pragma debug。

```
CGPROGRAM
#program surface surf Lambert vertex: vert
#program debug
```

- 然后点击 Show generated code。



- 我们查找 INTERNAL$_{DATA}$

```
#define INTERNAL_DATA half3 TtoW0; half3 TtoW1; half3 TtoW2;
#define WorldReflectionVector(data,normal) reflect (data.worldRefl, half3(dot(data.TtoW0,n
#define WorldNormalVector(data,normal) fixed3(dot(data.TtoW0,normal), dot(data.TtoW1,norma
```

- 我们发现 INTERNAL$_{DATA3half}$ TtoWi(i=0,1,2) 的变量，这三个变量合并在一起是一个 3x3 的矩阵，表示局部坐标系到世界坐标系的转换（Translate To World）。所以我们看到如果要使用 o.Normal 重新计算 worldRefl 和 worldNormal，就得使用到 INTERNAL$_{DATA}$

- 你可能会问上面的 COLOR semantic 是什么意思？当你写一个正常的片段着色器时，你得告诉别人你的输入结构体每个变量代表什么意思？如果你够疯狂，你可以试试下面这样定义：float2 MyUncleFred : TEXCOORD0; 并告诉别人 MyUncleFred 表示该模型的 uv 坐标。（画外音就是这种变量命名方式很令人费解）在表面着色器中你唯一担心的就是对 COLOR 类型的定义。float4 currentColor : COLOR; 可以看做目前已经经过插值后的像素颜色。当然你也可以不用关心这些，不过建议你命名上最好规范些，方便自己也方便别人。

4.3 一些本次写 Shader 用到的 CG 函数讲解

- 本次 Shader 书写用到了四个 CG 着色器编程语言中的函数——UnpackNormal、saturate、dot、tex2D。下面将分别对其进行讲解。

4.3.1 UnpackNormal( ) 函数

- UnpackNormal 接受一个 fixed4 的输入，并将其转换为所对应的法线值（fixed3），并将其赋给输出的 Normal，就可以参与到光线运算中完成接下来的渲染工作了。
- 一个调用示例：

```
o.Normal = UnpackNormal (tex2D (_BumpMap,IN.uv_BumpMap));
```

4.3.2 saturate( ) 函数

- saturate 的字面解释是浸湿，浸透。其作用其实也就是将取值转化为 [0,1] 之内的一个值。
- 其可选的原型如下：

```
float saturate(float x);
float1 saturate(float1 x);
float2 saturate(float2 x);
float3 saturate(float3 x);
float4 saturate(float4 x);
half saturate(half x);
half1 saturate(half1 x);
half2 saturate(half2 x);
half3 saturate(half3 x);
half4 saturate(half4 x);
fixed saturate(fixed x);
fixed1 saturate(fixed1 x);
fixed2 saturate(fixed2 x);
fixed3 saturate(fixed3 x);
fixed4 saturate(fixed4 x);
```

- 其唯一的一个参数 x 表示矢量或者标量的饱和值（Vector or scalar to saturate.），也就是将这个 x 转化为 [0,1] 之内的值。
- 其返回值：
  - 如果 x 取值小于 0，则返回值为 0.
  - 如果 x 取值大于 1，则返回值为 1.
  - 若 x 在 0 到 1 之间，则直接返回 x 的值。
- 其代码实现大致如下：

```
float saturate(float x) {
    return max(0,min(1, x));
}
```

- 一个调用示例：

```
half rim = 1.0 - saturate(dot(normalize(IN.viewDir), o.Normal));
```

4.3.3 dot( ) 函数

- dot 函数顾名思义，是高等数学中的点积操作，用于返回两个向量的标量积。
- 可选原型如下：

```
float dot(float a, float b);
float1 dot(float1 a, float1 b);
float2 dot(float2 a, float2 b);
float3 dot(float3 a, float3 b);
float4 dot(float4 a, float4 b);
half dot(half a, half b);
half1 dot(half1 a, half1 b);
half2 dot(half2 a, half2 b);
```

```
half3 dot(half3 a, half3 b);
half4 dot(half4 a, half4 b);
fixed dot(fixed a, fixed b);
fixed1 dot(fixed1 a, fixed1 b);
fixed2 dot(fixed2 a, fixed2 b);
fixed3 dot(fixed3 a, fixed3 b);
fixed4 dot(fixed4 a, fixed4 b);
```

- 其代码实现大致是这样的：

```
float dot(float4 a, float4 b) {
    return a.x * b.x + a.y * b.y + a.z * b.z + a.w * b.w;
}
```

- 一个调用示例：

```
float answer = dot (normalize(IN.viewDir),o.Normal);
```

4.3.4 tex2D( ) 函数

- 让我们看一看 CG 中用得比较多的用于 2D 纹理采样的 tex2D 函数的用法 (tex2d 函数，这是 CG 程序中用来在一张贴图中对一个点进行采样的方法)。其备选的原型也是非常之多：

```
float4 tex2D(sampler2D samp, float2 s)
float4 tex2D(sampler2D samp, float2 s, inttexelOff)
float4 tex2D(sampler2D samp, float3 s)
float4 tex2D(sampler2D samp, float3 s, inttexelOff)
float4 tex2D(sampler2D samp, float2 s,float2 dx, float2 dy)
float4 tex2D(sampler2D samp, float2 s,float2 dx, float2 dy, int texelOff)
float4 tex2D(sampler2D samp, float3 s,float2 dx, float2 dy)
float4 tex2D(sampler2D samp, float3 s,float2 dx, float2 dy, int texelOff)
int4 tex2D(isampler2D samp, float2 s)
int4 tex2D(isampler2D samp, float2 s, inttexelOff)
int4 tex2D(isampler2D samp, float2 s,float2 dx, float2 dy)
int4 tex2D(isampler2D samp, float2 s,float2 dx, float2 dy, int texelOff)
unsigned int4 tex2D(usampler2D samp, float2s)
unsigned int4 tex2D(usampler2D samp, float2s, int texelOff)
unsigned int4 tex2D(usampler2D samp, float2s, float2 dx, float2 dy)
unsigned int4 tex2D(usampler2D samp, float2s, float2 dx, float2 dy,int texelOff)
```

- 参数简介：
  - samp-需要查找的采样对象，也就是填个纹理对象在这里。
  - s-需进行查找的纹理坐标。
  - dx-预计算的沿 X 轴方向的导数。
  - dy-预计算的沿 Y 轴方向的导数。
  - texelOff-添加给最终纹理的偏移量
  - 而其返回值，自然是查找到的纹理。
- 最后，看一个综合了本次讲解的四个函数（UnpackNormal、saturate、tex2D、dot）的 Surface Shader 中 surf 函数的示例：

```
//【2】表面着色函数的编写
void surf(Input IN, inout SurfaceOutput o) {
    // 从主纹理获取 rgb 颜色值
    o.Albedo= tex2D(_MainTex, IN.uv_MainTex).rgb;
    // 从凹凸纹理获取法线值
    o.Normal= UnpackNormal(tex2D (_BumpMap, IN.uv_BumpMap));
    // 从 _RimColor 参数获取自发光颜色
    halfrim = 1.0 - saturate(dot(normalize(IN.viewDir), o.Normal));
    o.Emission= _RimColor.rgb * pow(rim, _RimPower);
}
```

4.4 写 Shdaer 实战

- 上面都是些概念，下面我们将进行一些实战的 Shader 书写，将学到的这些概念用到实际当中去。
- 本次我们将讲解 9 个表面 SurfaceShader 的写法，从最基本的 Surface Shader，循序渐进，一点一点加功能，到最后的稍微有点复杂的“凹凸纹理 + 颜色可调 + 边缘光照 + 细节纹理“表面 Shader 的写法。本期的全部 Shader 的合照如下：

4.4.1 最基本的 Surface Shader

- 先看一个使用内建光照模式的最基本的 Surface Shader 应该怎么写：

```
Shader " 浅墨 Shader 编程/Volume6/24. 最基本的 SurfaceShader" {
    //---------------------------------【子着色器】----------------------------------
    SubShader {
        //-----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
         //-------------------开始 CG 着色器编程语言段-----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式
        #pragma surface surf Lambert
        //【2】输入结构
        struct Input {
            // 四元素的颜色值（RGBA）
            float4 color : COLOR;
        };
        //【3】表面着色函数的编写
        void surf (Input IN, inout SurfaceOutput o) {
            // 反射率
            o.Albedo = float3(0.5, 0.8, 0.3); //(0.5,0.8,0.3) 分别对应于 RGB 分量
            // 而 o.Albedo = 0.6; 等效于写 o.Albedo = float3(0.6,0.6,0.6);
        }
        //-------------------结束 CG 着色器编程语言段-----------------
        ENDCG
    }
    // “备胎”为普通漫反射
    Fallback "Diffuse"
}
```

- 可以发现，一个最基本的 Surface Shader，至少需要有光照模式的声明、输入结构和表面着色函数的编写这三部分。
- 另外，主要注意其中的 surf 函数的写法，就是把上文讲到的 Surface Output 结构体中需要用到的成员变量拿来赋值：

```
1  //【2】表面着色函数的编写
2  void surf (Input IN, inout SurfaceOutput o) {
3      // 反射率
4      o.Albedo= float3(0.5,0.8,0.3);//(0.5,0.8,0.3) 分别对应于 RGB 分量
5      // 而 o.Albedo = 0.6; 等效于写 o.Albedo =float3(0.6,0.6,0.6);
6  }
```

- 注释中已经写得很明白，且之前也已经讲过，o.Albedo = 0.6; 等效于写 o.Albedo = float3(0.6,0.6,0.6);
- 来个举一反三，则 o.Albedo 1; 等效于写 o.Albedo float3(1,1,1);

4.4.2 颜色可调

- 在最基本的 Surface Shader 的基础上，加上一点代码，就成了这里的可调颜色的 Surface Shader：

```
1  Shader " 浅墨 Shader 编程/Volume6/25. 颜色可调的 SurfaceShader" {
2      //---------------------------------【属性】----------------------------------
```

```
    Properties {
        _Color ("【主颜色】Main Color", Color) = (0.1,0.3,0.9,1)
    }
    //-------------------------------【子着色器】---------------------------------
    SubShader {
        //-----------子着色器标签----------
        Tags { "RenderType"="Opaque" }
        //-------------------开始 CG 着色器编程语言段-----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式
        #pragma surface surf Lambert
        //变量声明
        float4 _Color;
        //【2】输入结构
        struct Input {
            // 四元素的颜色值（RGBA）
            float4 color : COLOR;
        };
        //【3】表面着色函数的编写
        void surf (Input IN, inout SurfaceOutput o) {
            // 反射率
            o.Albedo = _Color.rgb;
            // 透明值
            o.Alpha = _Color.a;
        }
        //-------------------结束 CG 着色器编程语言段-----------------
        ENDCG
    }
    // “备胎”为普通漫反射
    FallBack "Diffuse"
}
```

- 我们将此 Shader 编译后赋给材质，得到如下效果，和之前的固定功能 Shader 一样，可以自由调节颜色：

4.4.3 基本纹理载入

```
Shader " 浅墨 Shader 编程/Volume6/26. 基本纹理载入" {
    //-------------------------------【属性】---------------------------------
    Properties {
        _MainTex ("【主纹理】Texture", 2D) = "white" {}
    }
    //-------------------------------【子着色器】---------------------------------
    SubShader {
        //-----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
        //-------------------开始 CG 着色器编程语言段-----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式
        #pragma surface surf Lambert
        //【2】输入结构
        struct Input {
            // 纹理的 uv 值
            float2 uv_MainTex;
        };
        // 变量声明
        sampler2D _MainTex;
        //【3】表面着色函数的编写
        void surf (Input IN, inout SurfaceOutput o) {
            //从纹理获取 rgb 颜色值
```

```
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
        }
        //--------------------结束 CG 着色器编程语言段------------------
        ENDCG
    }
    // “备胎”为普通漫反射
    Fallback "Diffuse"
}
```

- sampler2D
  - sampler2D _MainTex; sampler2D 是个啥？其实在 CG 中，sampler2D 就是和 texture 所绑定的一个数据容器接口。等等.. 这个说法还是太复杂了，简单理解的话，所谓加载以后的 texture（贴图）说白了不过是一块内存存储的，使用了 RGB（也许还有 A）通道，且每个通道 8bits 的数据。而具体地想知道像素与坐标的对应关系，以及获取这些数据，我们总不能一次一次去自己计算内存地址或者偏移，因此可以通过 sampler2D 来对贴图进行操作。更简单地理解，sampler2D 就是 GLSL 中的 2D 贴图的类型，相应的，还有 sampler1D，sampler3D，samplerCube 等等格式。
- 对 $MainTex$
  - 为什么在这里需要一句对 $MainTex\ Properties$
  - 我们用来实例的这个 shader 其实是由两个相对独立的块组成的，外层的属性声明，回滚等等是 Unity 可以直接使用和编译的 ShaderLab；而现在我们是在 CGPROGRAM…ENDCG 这样一个代码块中，这是一段 CG 程序。对于这段 CG 程序，**要想访问在 Properties 中所定义的变量的话，必须使用和之前变量相同的名字进行声明。** 于是其实 sampler2D _MainTex; 做的事情就是再次声明并链接了 $MainTex\ CG$
- surf 着色器
  - 着色器就是给定了输入，然后给出输出进行着色的代码。CG 规定了声明为表面着色器的方法（就是我们这里的 surf）的参数类型和名字，因此我们没有权利决定 surf 的输入输出参数的类型，只能按照规定写。这个规定就是第一个参数是一个 Input 结构，第二个参数是一个 inout 的 SurfaceOutput 结构。
  - Input 其实是需要我们去定义的结构，这给我们提供了一个机会，可以把所需要参与计算的数据都放到这个 Input 结构中，传入 surf 函数使用；
  - SurfaceOutput 是已经定义好了里面类型输出结构，但是一开始的时候内容暂时是空白的，我们需要向里面填写输出，这样就可以完成着色了。
  - SurfaceOutput 是预定义的输出结构，我们的 surf 函数的目标就是根据输入把这个输出结构填上。SurfaceOutput 结构体的定义如下

```
struct SurfaceOutput {
    half3 Albedo;      // 像素的颜色
    half3 Normal;      // 像素的法向值
    half3 Emission;    // 像素的发散颜色/辐射光。辐射光是最简单的一种光，它直接从物体发出并且不
    half Specular;     // 像素的镜面高光
    half Gloss;        // 像素的发光强度
    half Alpha;        // 像素的透明度
};
```

- $\text{uv}_{MainTex}$
  - UV mapping 的作用是将一个 2D 贴图上的点按照一定的顺序映射到 3D 模型上，是 3D 渲染中最常用的一种顶点处理手段。
  - $\text{uv}_{MainTex}$: 在 CG 程序中,我们有这样的约定,在一个贴图变量(在我们的例子中是 $MainTex\ uv\ uv\quad surfuv$

4.4.4 凹凸纹理载入

- 让我们慢慢添加特性，使得到的 Surface Shader 的效果与功能越来越强大。接着来看看 Surface Shader 的凹凸纹理如何实现：

```
Shader " 浅墨 Shader 编程/Volume6/27. 凹凸纹理载入" {
    //------------------------------【属性】----------------------------------
    Properties {
        _MainTex ("【主纹理】Texture", 2D) = "white" {}
        _BumpMap ("【凹凸纹理】Bumpmap", 2D) = "bump" {}
    }
    //------------------------------【子着色器】--------------------------------
    SubShader {
        //----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
        //------------------开始 CG 着色器编程语言段----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式
        #pragma surface surf Lambert
        //【2】输入结构
        struct Input {
            // 主纹理的 uv 值
            float2 uv_MainTex;
            // 凹凸纹理的 uv 值
            float2 uv_BumpMap;
        };
        // 变量声明
        sampler2D _MainTex;//主纹理
        sampler2D _BumpMap;//凹凸纹理
        //【3】表面着色函数的编写
        void surf (Input IN, inout SurfaceOutput o) {
            // 从主纹理获取 rgb 颜色值
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
            // 从凹凸纹理获取法线值
            o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
        }
        //------------------结束 CG 着色器编程语言段----------------
        ENDCG
    }
    // “备胎”为普通漫反射
    Fallback "Diffuse"
}
```

4.4.5 纹理载入 + 颜色可调

- 接着看一看纹理如何通过一个 finalcolor 关键字自定义函数，来达到调色的目的：

```
Shader " 浅墨 Shader 编程/Volume6/28. 纹理 + 颜色修改" {
    //------------------------------【属性】----------------------------------
    Properties {
        _MainTex ("【主纹理】Texture", 2D) = "white" {}
        _ColorTint ("【色泽】Tint", Color) = (0.6, 0.3, 0.6, 0.3)
    }
    //------------------------------【子着色器】--------------------------------
    SubShader {
        //----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
        //------------------开始 CG 着色器编程语言段----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式 + 自定义颜色函数
        #pragma surface surf Lambert finalcolor:setcolor
        //【2】输入结构
        struct Input {
            // 纹理的 uv 值
```

```
        float2 uv_MainTex;
    };
    // 变量声明
    fixed4 _ColorTint;
    sampler2D _MainTex;
    //【3】自定义颜色函数 setcolor 的编写
    void setcolor (Input IN, SurfaceOutput o, inout fixed4 color) {
        // 将自选的颜色值乘给 color
        color *= _ColorTint;
    }
    //【4】表面着色函数的编写
    void surf (Input IN, inout SurfaceOutput o) {
        // 从主纹理获取 rgb 颜色值
        o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
    }
    //--------------------结束 CG 着色器编程语言段------------------
    ENDCG
}
// “备胎”为普通漫反射
Fallback "Diffuse"
}
```

4.4.6 凹凸纹理 + 边缘光照

- 在之前凹凸纹理的基础上让我们加上喜闻乐见的边缘光照：

```
Shader " 浅墨 Shader 编程/Volume6/29. 凹凸纹理 + 边缘光照" {
    //---------------------------------【属性】----------------------------------------
    Properties {
        _MainTex ("【主纹理】Texture", 2D) = "white" {}
        _BumpMap ("【凹凸纹理】Bumpmap", 2D) = "bump" {}
        _RimColor ("【边缘颜色】Rim Color", Color) = (0.26,0.19,0.16,0.0)
        _RimPower ("【边缘颜色强度】Rim Power", Range(0.5,8.0)) = 3.0
    }
    //---------------------------------【子着色器】---------------------------------
    SubShader {
        //-----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
        //--------------------开始 CG 着色器编程语言段-----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式 + 自定义颜色函数
        #pragma surface surf Lambert
        //【2】输入结构
        struct Input {
            // 主纹理的 uv 值
            float2 uv_MainTex;
            // 凹凸纹理的 uv 值
            float2 uv_BumpMap;
            // 当前坐标的视角方向
            float3 viewDir;
        };
        // 变量声明
        sampler2D _MainTex; // 主纹理
        sampler2D _BumpMap; // 凹凸纹理
        float4 _RimColor;   // 边缘颜色
        float _RimPower;    // 边缘颜色强度
        //【3】表面着色函数的编写
        void surf (Input IN, inout SurfaceOutput o) {
            // 从主纹理获取 rgb 颜色值
```

```
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
            // 从凹凸纹理获取法线值
            o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
            // 从 _RimColor 参数获取自发光颜色
            half rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
            o.Emission = _RimColor.rgb * pow (rim, _RimPower);
        }
        //------------------结束 CG 着色器编程语言段-----------------
        ENDCG
    }
    // “备胎”为普通漫反射
    Fallback "Diffuse"
}
```

- 其中的 viewDir 意为 WorldSpace View Direction，也就是当前坐标的视角方向：
- 关于 surf 中的两句新加的代码在这里也讲一下。
- 上面已经提到过，Normalize 函数，用于获取到的 viewDir 坐标转成一个单位向量且方向不变，外面再与点的法线做点积。最外层再用 saturate 算出一 [0,1] 之间的最靠近的值。这样算出一个 rim 边界。原理可以这样解释：
- 这里 o.Normal 就是单位向量。外加 Normalize 了 viewDir。因此求得的点积就是夹角的 cos 值。因为 cos 值越大，夹角越小，所以，这时取反来。这样，夹角越大，所反射上的颜色就越多。于是就得到的两边发光的效果。哈哈这样明了吧。
- 这里再介绍一下这个 half。CG 里还有类似的 float 和 fixed。half 是一种低精度的 float，但有时也会被选择成与 float 一样的精度。先就说这么多吧，后面还会遇到的，到时候再讲。
- 我们将此 Shader 编译后赋给材质，得到如下效果，除了凹凸纹理的选择之外，还有边缘发光颜色和强度可供自由定制：

4.4.7 凹凸纹理 + 颜色可调

- 接下来我们看看凹凸纹理 + 颜色可调的 Shader 怎么写：

```
Shader " 浅墨 Shader 编程/Volume6/30. 凹凸纹理 + 颜色可调 + 边缘光照" {
    //-------------------------------【属性】-------------------------------------
    Properties {
        _MainTex ("【主纹理】Texture", 2D) = "white" {}
        _BumpMap ("【凹凸纹理】Bumpmap", 2D) = "bump" {}
        _ColorTint ("【色泽】Tint", Color) = (0.6, 0.3, 0.6, 0.3)
        _RimColor ("【边缘颜色】Rim Color", Color) = (0.26,0.19,0.16,0.0)
        _RimPower ("【边缘颜色强度】Rim Power", Range(0.5,8.0)) = 3.0
    }
    //-------------------------------【子着色器】---------------------------------
    SubShader {
        //----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
        //------------------开始 CG 着色器编程语言段-----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式 + 自定义颜色函数
        #pragma surface surf Lambert finalcolor:setcolor
        //【2】输入结构
        struct Input {
            // 主纹理的 uv 值
            float2 uv_MainTex;
            // 凹凸纹理的 uv 值
            float2 uv_BumpMap;
            // 当前坐标的视角方向
            float3 viewDir;
```

```
        };
        // 变量声明
        sampler2D _MainTex;
        sampler2D _BumpMap;
        fixed4 _ColorTint;
        float4 _RimColor;
        float _RimPower;
        //【3】自定义颜色函数 setcolor 的编写
        void setcolor (Input IN, SurfaceOutput o, inout fixed4 color) {
            color *= _ColorTint;
        }
        //【4】表面着色函数的编写
        void surf (Input IN, inout SurfaceOutput o) {
            // 从主纹理获取 rgb 颜色值
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
            // 从凹凸纹理获取法线值
            o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
            // 从 _RimColor 参数获取自发光颜色
            half rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
            o.Emission = _RimColor.rgb * pow (rim, _RimPower);
        }
        //-------------------结束 CG 着色器编程语言段------------------
        ENDCG
    }
    // “备胎”为普通漫反射
    Fallback "Diffuse"
}
```

- 我们将此 Shader 编译后赋给材质，得到如下非常赞的效果。除了载入纹理，还有色泽，边缘颜色和强度可供调节：

4.4.8 细节纹理

- 接着我们来看一个在屏幕上显示纹理细节的 Shader：

```
Shader " 浅墨 Shader 编程/Volume6/31. 细节纹理" {
    //-------------------------------【属性】---------------------------------------
    Properties {
        _MainTex ("【主纹理】Texture", 2D) = "white" {}
        _Detail ("【细节纹理】Detail", 2D) = "gray" {}
    }
    //-------------------------------【子着色器】-----------------------------------
    SubShader {
        //-----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
        //--------------------开始 CG 着色器编程语言段-----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式
        #pragma surface surf Lambert
        //【2】输入结构
        struct Input {
            // 主纹理的 uv 值
            float2 uv_MainTex;
            // 细节纹理的 uv 值
            float2 uv_Detail;
        };
         // 变量声明
        sampler2D _MainTex;
        sampler2D _Detail;
         //【3】表面着色函数的编写
```

```
        void surf (Input IN, inout SurfaceOutput o) {
            // 先从主纹理获取 rgb 颜色值
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
            // 设置细节纹理
            o.Albedo *= tex2D (_Detail, IN.uv_Detail).rgb * 2;
        }
        //-------------------结束 CG 着色器编程语言段------------------
        ENDCG
    }
    // “备胎”为普通漫反射
    Fallback "Diffuse"
}
```

4.4.9 凹凸纹理 + 颜色可调 + 边缘光照 + 细节纹理

- 结合上面的 8 个 Shader，我们可以完成本期文章这个结合了凹凸纹理 + 颜色可调 + 边缘光照 + 细节纹理的稍微复杂一点的 Surface Shader：

```
Shader " 浅墨 Shader 编程/Volume6/32. 凹凸纹理 + 颜色可调 + 边缘光照 + 细节纹理" {
    Properties {
        _MainTex ("【主纹理】Texture", 2D) = "white" {}
        _BumpMap ("【凹凸纹理】Bumpmap", 2D) = "bump" {}
        _Detail ("【细节纹理】Detail", 2D) = "gray" {}
        _ColorTint ("【色泽】Tint", Color) = (0.6, 0.3, 0.6, 0.3)
        _RimColor ("【边缘颜色】Rim Color", Color) = (0.26,0.19,0.16,0.0)
        _RimPower ("【边缘颜色强度】Rim Power", Range(0.5,8.0)) = 3.0
    }
    //---------------------------------【子着色器】----------------------------------
    SubShader {
        //-----------子着色器标签----------
        Tags { "RenderType" = "Opaque" }
        //-------------------开始 CG 着色器编程语言段-----------------
        CGPROGRAM
        //【1】光照模式声明：使用兰伯特光照模式 + 自定义颜色函数
        #pragma surface surf Lambert finalcolor:setcolor
        //【2】输入结构
        struct Input {
            // 主纹理的 uv 值
            float2 uv_MainTex;
            // 凹凸纹理的 uv 值
            float2 uv_BumpMap;
            // 细节纹理的 uv 值
            float2 uv_Detail;
            // 当前坐标的视角方向
            float3 viewDir;
        };
        // 变量声明
        sampler2D _MainTex;
        sampler2D _BumpMap;
        sampler2D _Detail;
        fixed4 _ColorTint;
        float4 _RimColor;
        float _RimPower;
        //【3】自定义颜色函数 setcolor 的编写
        void setcolor (Input IN, SurfaceOutput o, inout fixed4 color) {
            color *= _ColorTint;
        }
        //【4】表面着色函数的编写
        void surf (Input IN, inout SurfaceOutput o) {
```

```
            // 先从主纹理获取 rgb 颜色值
            o.Albedo = tex2D (_MainTex, IN.uv_MainTex).rgb;
            // 设置细节纹理
            o.Albedo *= tex2D (_Detail, IN.uv_Detail).rgb * 2;
            // 从凹凸纹理获取法线值
            o.Normal = UnpackNormal (tex2D (_BumpMap, IN.uv_BumpMap));
            // 从 _RimColor 参数获取自发光颜色
            half rim = 1.0 - saturate(dot (normalize(IN.viewDir), o.Normal));
            o.Emission = _RimColor.rgb * pow (rim, _RimPower);
        }
        //-------------------结束 CG 着色器编程语言段------------------
        ENDCG
    }
    // “备胎”为普通漫反射
    Fallback "Diffuse"
}
```

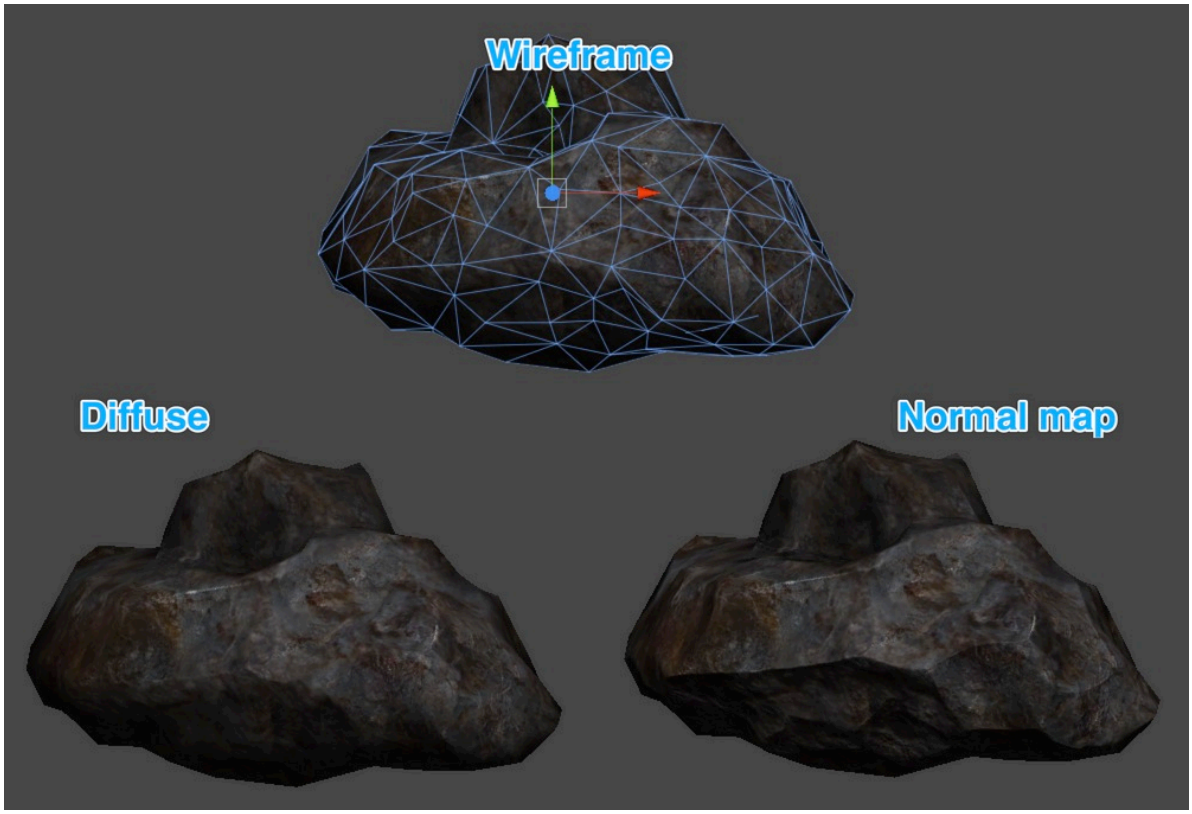
4.4.10 选择 Unity Shader 的建议

- 1. 除非设备是不支持可编程管线的着色器的，才考虑使用固定函数着色器，否则都使用可编程管线的着色器。
- 2. 想和各种光源打资产，则可以使用表面着色器，但要注意他在移动端的表现。
- 3. 如果需要使用的光照数目特别少，那么顶点/片元着色器是一个更好的选择。
- 4. 如果有很多自定义的渲染效果，那么选择顶点/片元着色器。

5 关于 shader 的一些基本原理 (系统化、高端架构、深入浅出)

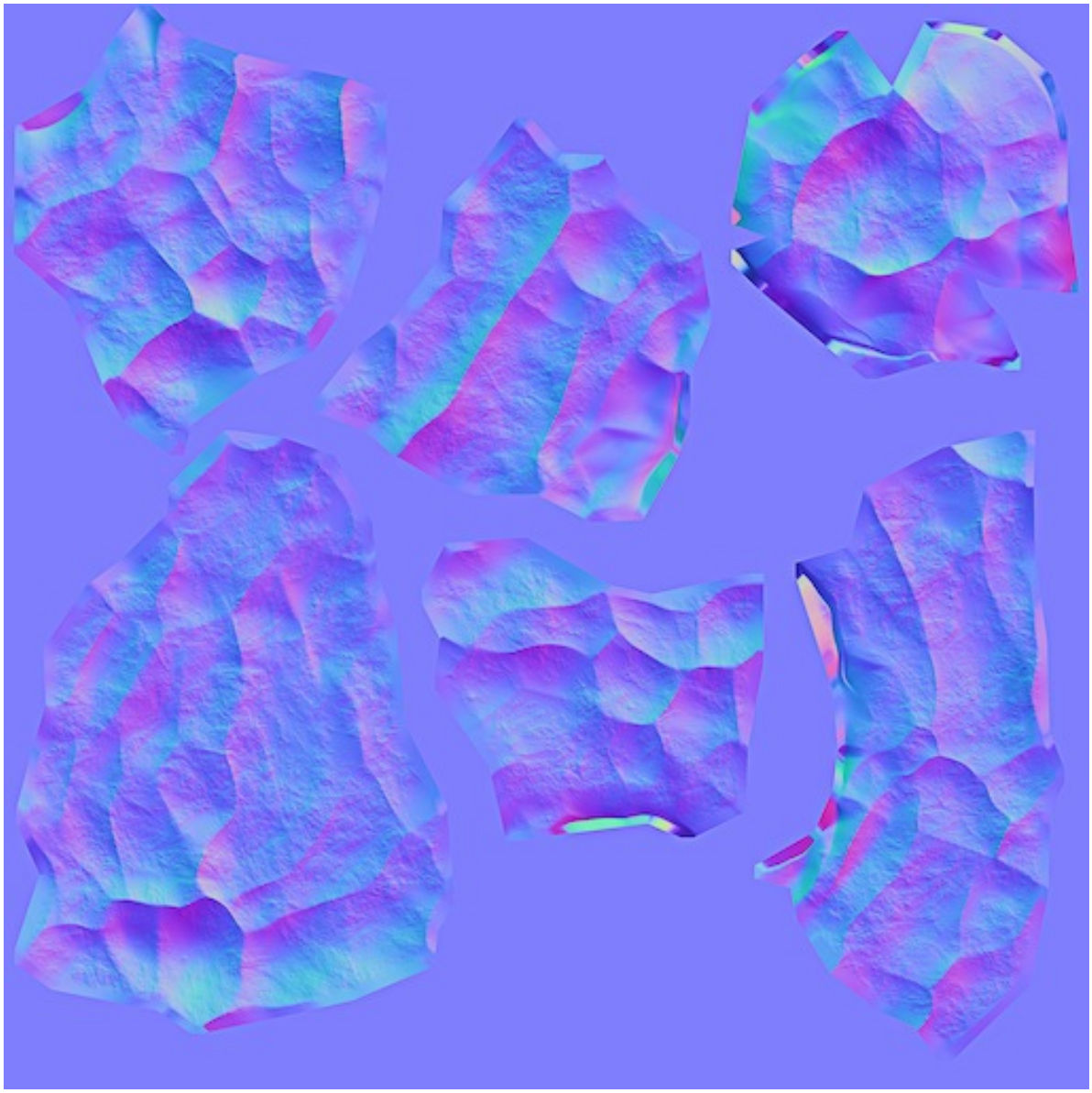
5.1 法线贴图 (Normal Mapping)

- 法线贴图是凸凹贴图 (Bump mapping) 的一种常见应用，简单说就是在不增加模型多边形数量的前提下，通过渲染暗部和亮部的不同颜色深度，来为原来的贴图和模型增加视觉细节和真实效果。简单原理是在普通的贴图的基础上，再另外提供一张对应原来贴图的，可以表示渲染浓淡的贴图。通过将这张附加的表示表面凸凹的贴图的应用于实际的原贴图进行运算后，可以得到新的细节更加丰富富有立体感的渲染效果。在本节中，我们将首先实现一个法线贴图的 Shader，然后对 Unity Shader 的光照模型进行一些讨论，并实现一个自定义的光照模型。最后再通过更改 shader 模拟一个石头上的积雪效果，并对模型顶点进行一些修改使积雪效果看起来比较真实。在本节结束的时候，我们就会有一个比较强大的可以满足一些真实开发工作时可用的 shader 了，而且更重要的是，我们将会掌握它是如何被创造出来的。
- 关于法线贴图的效果图，可以对比看看下面。模型面数为 500，左侧只使用了简单的 Diffuse 着色，右侧使用了法线贴图。比较两张图片不难发现，使用了法线贴图的石头在暗部和亮部都有着更好的表现。整体来说，凸凹感比 Diffuse 的结果增强许多，石头看起来更真实也更具有质感。



- 本节中需要用到的上面的素材可以在这里下载，其中包括上面的石块的模型，一张贴图以及对应的法线贴图。将下载的 package 导入到工程中，并新建一个 material，使用简单的 Diffuse 的 Shader（比如上一节我们实现的），再加上一个合适的平行光光源，就可以得到我们左图的效果。另外，本节以及以后都会涉及到一些 Unity 内建的 Shader 的内容，比如一些标准常用函数和常量定义等，相关内容可以在 Unity 的内建 Shader 中找到，内建 Shader 可以在 Unity 下载页面的版本右侧找到。

- 接下来我们实现法线贴图。在实现之前，我们先简单地稍微多了解一些法线贴图的基本知识。大多数法线图一般都和下面的图类似，是一张以蓝紫色为主的图。这张法线图其实是一张 RGB 贴图，其中红，绿，蓝三个通道分别表示由高度图转换而来的该点的法线指向：Nx、Ny、Nz。在其中绝大部分点的法线都指向 z 方向，因此图更偏向于蓝色。在 shader 进行处理时，我们将光照与该点的法线值进行点积后即可得到在该光线下应有的明暗特性，再将其应用到原图上，即可反应在一定光照环境下物体的凹凸关系了。关于法向贴图的更多信息，可以参考 wiki 上的相关条目。



- 回到正题，我们现在考虑的主要是 Shader 入门，而不是图像学的原理。再上一节我们写的 Shader 的基础上稍微做一些修改，就可以得到适应并完成法线贴图渲染的新 Shader。新加入的部分进行了编号并在之后进行说明。

```
Shader "Custom/Normal Mapping" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" { }
        //1
        _Bump ("Bump", 2D) = "bump" {}
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200
        CGPROGRAM
        #pragma surface surf Lambert
        sampler2D _MainTex;
        //2
        sampler2D _Bump;
        struct Input {
            float2 uv_MainTex;
            //3
```

```
            float2 uv_Bump;
        };
        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
            //4
            o.Normal = UnpackNormal(tex2D(_Bump, IN.uv_Bump));
            o.Albedo = c.rgb;
            o.Alpha = c.a;
        }
    ENDCG
    }
    FallBack "Diffuse"
}
```

- 1. 声明并加入一个显示名称为 Bump 的贴图，用于放置法线图
- 2. 为了能够在 CG 程序中使用这张贴图，必须加入一个 sample，希望你还记得～
- 3. 获取 Bump 的 uv 信息作为输入
- 4. 从法线图中提取法线信息，并将其赋予相应点的输出的 Normal 属性。
  - UnpackNormal 是定义在 UnityCG.cginc 文件中的方法，这个文件中包含了一系列常用的 CG 变量以及方法。
  - UnpackNormal 接受一个 fixed4 的输入，并将其转换为所对应的法线值（fixed3）。
  - 在解包得到这个值之后，将其赋给输出的 Normal，就可以参与到光线运算中完成接下来的渲染工作了。
- 现在保存并且编译这个 Shader，创建新的 material 并使用这个 shader，将石头的材质贴图和法线图分别拖放到 Base 和 Bump 里，再将其应用到石头模型上，应该就可以看到右侧图的效果了。

5.2 光照模型

- 在我们之前的看到的 Shader 中（其实也就上一节的基本 diffuse 和这里的 normal mapping），都只使用了 Lambert 的光照模型（#pragma surface surf Lambert），这是一个很经典的漫反射模型，光强与入射光的方向和反射点处表面法向夹角的余弦成正比。关于 Lambert 和漫反射的一些详细的计算和推论，可以参看 wiki（Lambert，漫反射）或者其他地方的介绍。一句话的简单解释就是一个点的反射光强是和该点的法线向量和入射光向量和强度和夹角有关系的，其结果就是这两个向量的点积。既然已经知道了光照计算的原理，我们先来看看如何实现一个自己的光照模型吧。
- 在刚才的 Shader 上进行如下修改。
  - 首先将原来的 #pragma 行改为这样

```
#pragma surface surf CustomDiffuse
```

  - 然后在 SubShader 块中添加如下代码

```
inline float4 LightingCustomDiffuse (SurfaceOutput s, fixed3 lightDir, fixed atten) {
    float difLight = max(0, dot (s.Normal, lightDir));
    float4 col;
    col.rgb = s.Albedo * _LightColor0.rgb * (difLight * atten * 2);
    col.a = s.Alpha;
    return col;
}
```

  - 最后保存，回到 Unity。Shader 将编译，如果一切正常，你将不会看到新的 shader 和之前的在材质表现上有任何不同。但是事实上我们现在的 shader 已经与 Unity 内建的 diffuse 光照模型撇清了关系，而在使用我们自己设定的光照模型了。
- 喵的，这些代码都干了些什么！相信你一定会有这样的疑惑…没问题，没有疑惑的话那就不叫初学了，还是一行行讲来。首先正像我们上一篇所说，#pragma 语句在这里声明了接下来的 Shader 的类型，计算调用的方法名，以及指定光照模型。在之前我们一直指定 Lambert 为光照模型，而现在我们将其换为了 CustomDiffuse。
- 接下来添加的代码是计算光照的实现。

  - shader 中对于方法的名称有着比较严格的约定，想要创建一个光照模型，首先要做的是按照规则声明一个光照计算的函数名字，即 Lighting<Your Chosen Name>。对于我们的光照模型 CustomDiffuse，其计算函数的名称自然就是 LightingCustomDiffuse 了。
  - 光照模型的计算是在 surf 方法的表面颜色之后，根据输入的光照条件来对原来的颜色在这种光照下的表现进行计算，最后输出新的颜色值给渲染单元完成在屏幕的绘制。

- 也许你已经猜到了，我们之前用的 Lambert 光照模型是不是也有一个名字叫 LightingLambert 的光照计算函数呢？Bingo。在 Unity 的内建 Shader 中，有一个 Lighting.cginc 文件，里面就包含了 LightingLambert 的实现。也许你也注意到了，我们所实现的 LightingCustomDiffuse 的内容现在和 Unity 内建中的 LightingLambert 是完全一样的，这也就是使用新的 shader 的原来视觉上没有区别的原因，因为实现确实是完全一样的。

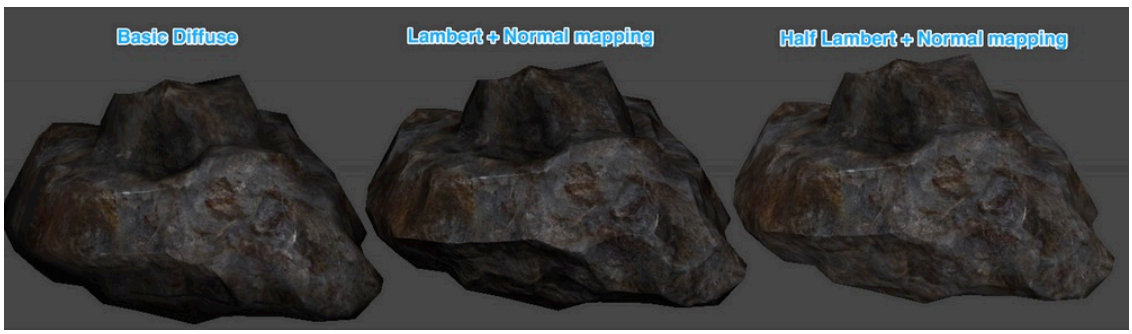
  - 首先来看输入量，SurfaceOutput s 这个就是经过表面计算函数 surf 处理后的输出，我们讲对其上的点根据光线进行处理，fixed3 lightDir 是光线的方向，fixed atten 表示光衰减的系数。
  - 在计算光照的代码中，我们先将输入的 s 的法线值（在 Normal mapping 中的话这个值已经是法线图中的对应量了）和输入光线进行点积（dot 函数是 CG 中内置的数学函数，希望你还记得，可以参考这里）。点积的结果在-1 至 1 之间，这个值越大表示法线与光线间夹角越小，这个点也就应该越亮。
  - 之后使用 max 来将这个系数结果限制在 0 到 1 之间，是为了避免负数情况的存在而导致最终计算的颜色变为负数，输出一团黑，一般来说这是我们不愿意看到的。
  - 接下来我们将 surf 输出的颜色与光线的颜色 $_{LightColor0}$.rgb（由 Unity 根据场景中的光源得到的，它在 Lighting.cginc 中有声明）进行乘积，然后再与刚才计算的光强系数和输入的衰减系数相乘，最后得到在这个光线下的颜色输出（关于 difLight * atten * 2 中为什么有个乘 2，这是一个历史遗留问题，主要是为了进行一些光强补偿，可以参见这里的讨论）。

- 在了解了基本实现方式之后，我们可以看看做一些修改玩玩儿。最简单的比如将这个 Lambert 模型改亮一些，比如换成 Half Lambert 模型。Half Lambert 是由 Valve 创造的可以使物体在低光线条件下增亮的技术，最早被用于半条命（Half Life）中以避免在低光下物体的走形。简单说就是把光强系数先取一半，然后在加 0.5，代码如下：

```
inline float4 LightingCustomDiffuse (SurfaceOutput s, fixed3 lightDir, fixed atten) {
    float difLight = dot (s.Normal, lightDir);
    float hLambert = difLight * 0.5 + 0.5;
    float4 col;
    col.rgb = s.Albedo * _LightColor0.rgb * (hLambert * atten * 2);
    col.a = s.Alpha;
    return col;
}
```

- 这样一来，原来光强 0 的点，现在对应的值变为了 0.5，而原来是 1 的地方现在将保持为 1。也就是说模型贴图的暗部被增强变亮了，而亮部基本保持和原来一样，防止过曝。使用 Half Lambert 前后的效果图如下，注意最右侧石头下方的阴影处细节更加明显了，而这一切都只是视觉效果的改变，不涉及任何贴图和模型的变化。

- Half Lambert 下发现贴图的表现



5.3 表面贴图的追加效果

- OK，对于光线和自定义光照模型的讨论暂时到此为止，因为如果展开的话这将会一个庞大的图形学和经典光学的话题了。我们回到 Shader，并且一起实现一些激动人心的效果吧。比如，在你的游戏场景中有一幕是雪地场景，而你希望做一些石头上白雪皑皑的覆盖效果，应该怎么办呢？难道让你可爱的 3D 设计师再去出

一套覆雪的贴图然后使用新的贴图？当然不，不是不能，而是不该。因为新的贴图不仅会增大项目的资源包体积，更会增大之后修改和维护的难度，想想要是有好多石头需要实现同样的覆雪效果，或者是要随着游戏时间堆积的雪逐渐变多的话，你应该怎么办？难道让设计师再把所有的石头贴图都盖上雪，然后再按照雪的厚度出 5 套不同的贴图么？相信我，他们会疯的。

- 于是，我们考虑用 Shader 来完成这件工作吧！先考虑下我们需要什么，积雪效果的话，我们需要积雪等级（用来表示积雪量），雪的颜色，以及积雪的方向。基本思路和实现自定义光照模型类似，通过计算原图的点在世界坐标中的法线方向与积雪方向的点积，如果大于设定的积雪等级的阈值的话则表示这个方向与积雪方向是一致的，其上是可以积雪的，显示雪的颜色，否则使用原贴图的颜色。废话不再多说，上代码，在上面的 Shader 的基础上，更改 Properties 里的内容为

```
Properties {
    _MainTex ("Base (RGB)", 2D) = "white" {}
    _Bump ("Bump", 2D) = "bump" {}
    _Snow ("Snow Level", Range(0,1) ) = 0
    _SnowColor ("Snow Color", Color) = (1.0,1.0,1.0,1.0)
    _SnowDirection ("Snow Direction", Vector) = (0,1,0)
}
```

- 没有太多值得说的，唯一要提一下的是 $_{SnowDirection}$(0,1,0)，这表示我们希望雪是垂直落下的。对应地，在 CG 程序中对这些变量进行声明：

```
sampler2D _MainTex;
sampler2D _Bump;
float _Snow;
float4 _SnowColor;
float4 _SnowDirection;
```

- 接下来改变 Input 的内容：

```
struct Input {
    float2 uv_MainTex;
    float2 uv_Bump;
    float3 worldNormal; INTERNAL_DATA
};
```

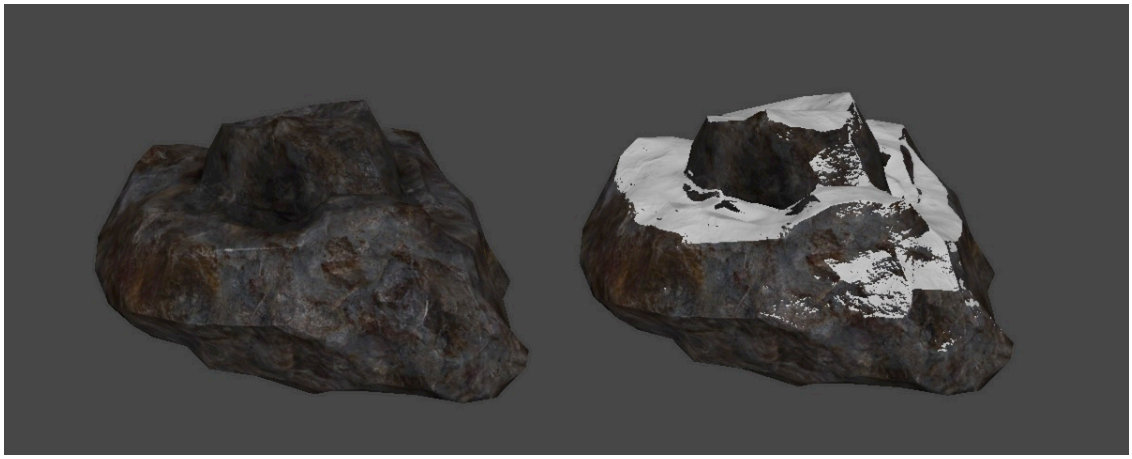
- 相对于上面的 Shader 输入来说，加入了一个 float3 worldNormal; INTERNAL$_{DATA\ SurfaceOutputNormal\ worldN}$

```
void surf (Input IN, inout SurfaceOutput o) {
    half4 c = tex2D (_MainTex, IN.uv_MainTex);
    o.Normal = UnpackNormal(tex2D(_Bump, IN.uv_Bump));
    if (dot(WorldNormalVector(IN, o.Normal), _SnowDirection.xyz) > lerp(1,-1,_Snow)) {
        o.Albedo = _SnowColor.rgb;
    } else {
        o.Albedo = c.rgb;
    }
    o.Alpha = c.a;
}
```

- 和上面相比，加入了一个 if…else…的判断。首先看这个条件的不等式的左侧，我们对雪的方向和和输入点的世界法线方向进行点积。WorldNormalVector 通过输入的点及这个点的法线值，来计算它在世界坐标中的方向；右侧的 lerp 函数相信只要对插值有概念的同学都不难理解：当 _Snow 取最小值 0 时，这个函数将返回 1，而 _Snow 取最大值时，返回-1。这样我们就可以通过设定 _Snow 的值来控制积雪的阈值，要是积雪等级 _Snow 是 0 时，不等式左侧不可能大于右侧，因此完全没有积雪；相反要是 _Snow 取最大值 1 时，由于左侧必定大于-1，所以全模型积雪。而随着取中间值的变化，积雪的情况便会有所不同。

- 应用这个 Shader，并且适当地调节一下积雪等级和颜色，可以得到如下右边的效果。

- 添加了积雪效果的 Shader



5.4 更改顶点模型

- 到现在位置，我们还仅指是在原贴图上进行操作，不管是用法线图使模型看起来凸凹有致，还是加上积雪，所有的计算和颜色的输出都只是“障眼法”，并没有对模型有任何实质的改动。但是对于积雪效果来说，实际上积雪是附加到石头上面，而不应当简单替换掉原来的颜色。但是具体实施起来，最简单的办法还是直接替换颜色，但是我们可以稍微变更一下模型，使原来的模型在积雪的方向稍微变大一些，这样来达到一种雪是附加到石头上的效果。

- 我们继续修改之前的 Shader，首先我们需要告诉 surface shadow 我们要改变模型的顶点。首先将 #param 行改为

```
#pragma surface surf CustomDiffuse vertex:vert
```

- 这告诉 Shader 我们想要改变模型顶点，并且我们会写一个叫做 vert 的函数来改变顶点。接下来我们再添加一个参数，在 Properties 中声明一个 $_{SnowDepth\ CG}$

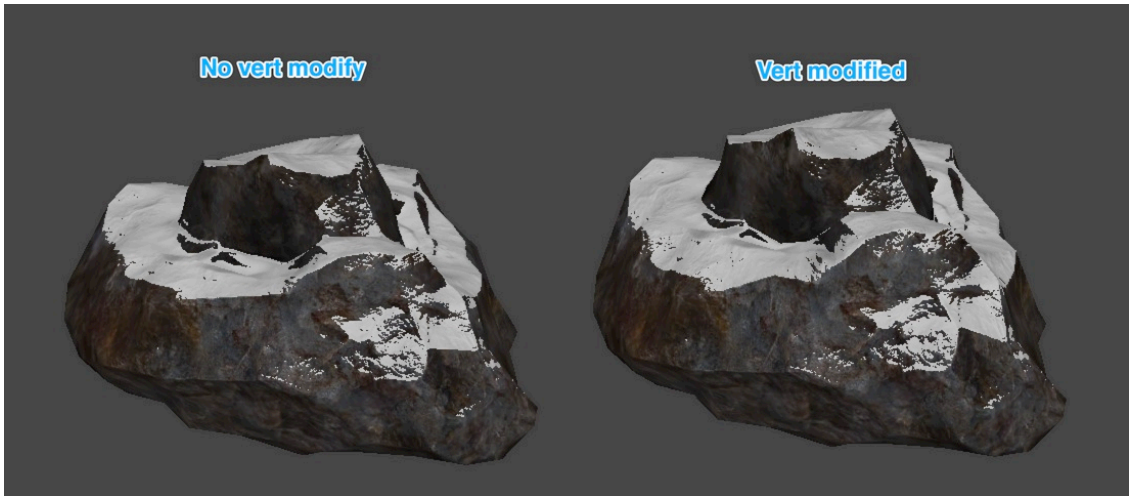
```
// In Properties{...}
_SnowDepth ("Snow Depth", Range(0,0.3)) = 0.1
// In CG declare
float _SnowDepth;
```

- 接下来实现 vert 方法，和之前积雪的运算其实比较类似，判断点积大小来决定是否需要扩大模型以及确定模型扩大的方向。在 CG 段中加入以下 vert 方法

```
void vert (inout appdata_full v) {
    float4 sn = mul(transpose(_Object2World) , _SnowDirection);
    if(dot(v.normal, sn.xyz) >= lerp(1,-1, (_Snow * 2) / 3)) {
        v.vertex.xyz += (sn.xyz + v.normal) * _SnowDepth * _Snow;
    }
}
```

- 和 surf 的原理差不多，系统会输入一个当前的顶点的值，我们根据需要计算并填上新的值作为返回即可。上面第一行中使用 transpose 方法输出原矩阵的转置矩阵，在这里 $_{Object2WorldUnity}$ ShaderLab 的内建值，它表示将当前模型转换到世界坐标中的矩阵，将其与积雪方向做矩阵乘积得到积雪方向在物体的世界空间中的投影（把积雪方向转换到世界坐标中）。之后我们计算了这个世界坐标中实际的积雪方向和当前点的法线值的点积，并将结果与使用积雪等级的 2/3 进行比较 lerp 后的阈值比较。这样，当前点如果和积雪方向一致，并且积雪较为完整的话，将改变该点的模型顶点高度。

- 加入模型更改前后的效果对比如下图，加入模型调整的右图表现要更为丰满真实。



- 这个完整的代码如下

```
Shader "Custom/VVSnowShader" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        _Bump ("Bump", 2D) = "bump" {}
        _Snow ("Snow Level", Range(0,1) ) = 0
            _SnowColor ("Snow Color", Color) = (1.0,1.0,1.0,1.0)
            _SnowDirection ("Snow Direction", Vector) = (0,1,0)
            _SnowDepth ("Snow Depth", Range(0,0.3)) = 0.1
    }
    SubShader {
        Tags { "RenderType"="Opaque" }
        LOD 200

        CGPROGRAM
        #pragma surface surf CustomDiffuse vertex:vert

        sampler2D _MainTex;
        sampler2D _Bump;
        float _Snow;
        float4 _SnowColor;
        float4 _SnowDirection;
        float _SnowDepth;

        struct Input {
            float2 uv_MainTex;
            float2 uv_Bump;
            float3 worldNormal;
            INTERNAL_DATA
        };

        inline float4 LightingCustomDiffuse (SurfaceOutput s, fixed3 lightDir, fixed atten
            float difLight = dot (s.Normal, lightDir);
            float hLambert = difLight * 0.5 + 0.5;
            float4 col;
            col.rgb = s.Albedo * _LightColor0.rgb * (hLambert * atten * 2);
            col.a = s.Alpha;
            return col;
        }

        void vert (inout appdata_full v) {
            float4 sn = mul(transpose(_Object2World) , _SnowDirection);
```

```
            if(dot(v.normal, sn.xyz) >= lerp(1,-1, (_Snow * 2) / 3)) {
                v.vertex.xyz += (sn.xyz + v.normal) * _SnowDepth * _Snow;
            }
        }

        void surf (Input IN, inout SurfaceOutput o) {
            half4 c = tex2D (_MainTex, IN.uv_MainTex);
            o.Normal = UnpackNormal(tex2D(_Bump, IN.uv_Bump));
            if (dot(WorldNormalVector(IN, o.Normal), _SnowDirection.xyz) > lerp(1,-1,_Snow
                o.Albedo = _SnowColor.rgb;
            } else {
                o.Albedo = c.rgb;
            }
            o.Alpha = 1;
        }
        ENDCG
    }
    FallBack "Diffuse"
}
```

6 移动设备 GPU 架构简述

《The Mali GPU: An Abstract Machine》系列以 Arm Mali GPU 为例子给出了全面的讨论，现简述如下：

6.1 Part 1 - Frame Pipelining

- Application/Geometry/Fragment 三阶段组成，三者中最大才是瓶颈
- OpenGL 的同步 API 是个“illusion”，事实上是 CommandQueue（直到遇到 Fence 会被强制同步），以减少 CPU/GPU 之间的互相等待
- Pipeline Throttle，为了更低的延迟，当 GPU 累积了多帧（往往是 3 帧，以 eglSwapBuffers() 或 Present() 来区分帧）的 Command 时，OS 会通过 eglSwapBuffers() 或 Present() 来阻塞 CPU 让其进入 idle，从而防止更多后续 Command 的提交

6.2 Part 2 - Tile-based Rendering

- tile-based deferred rendering （Wiki，PowerVR/Mali/Adreno）是重要的概念。其将 Fragment 一帧处理多个比如 16x16 的单元，并为 Shader 集成一个小但快的 cache，从而大幅避免 Shader 和主内存之间带宽消耗（电量消耗）

6.3 Part 3 - The Midgard Shader Core

- GPU 包含数个（当前常见为 4-8 个）Unified Shading Core，可动态分配用于 Vertex Shader、Fragment Shader 或 Compute Kernel
- 每个 Unified Shader Core 包含数个（当前常见为 2 个）用于 SIMD 计算的运算器 Arithmetic Pipeline（A-pipe），1 个用于纹理采样的 Texutre Pipeline（T-pipe），1 个用于非纹理类的内存读写的 Load/Store Pipeline（LS-pipe）比如顶点属性写读、变量访问等
- 会进行 Early-ZS 测试尝试减少 Overdraw（依赖于渲染物体提交顺序由前至后）
- Arm 的 Forward Pixel Kill 和 PowerVR 的 Hidden Surface Removal 做到像素级别的 Overdraw 减少（不用依赖于渲染物体提交顺序由前至后）
- 当 Shader 使用 discard 或 clip、在 Fragment Shader 里修改深度值、半透明，将不能进行 Early-ZS，只好使用传统的 Late-ZS

6.4 Part 4 - The Bifrost Shader Core

- 2016 年的新型号，对架构作出了优化

7 References

7.1 Implementing a Loading Bar in Unity

- http://www.alanzucconi.com/2016/03/30/loading-bar-in-unity/
-
-
-

7.2 shader

- http://blog.csdn.net/poem_qianmo/article/details/40723789 【浅墨 Unity3D Shader 编程】之一夏威夷篇：游戏场景的创建 & 第一个 Shader 的书写
- http://www.jianshu.com/p/7b9498e58659 Unity ShaderLab 学习总结
- 猫都能学会的 Unity3D Shader 入门指南（一）https://onevcat.com/2013/07/shader-tutorial-1/
- Youtube：https://www.youtube.com/watch?v=hDJQXzajiPg （包括 part1-6）。视频是最佳的入门方式没有之一，所以墙裂建议就算不看下文的所有内容，都要去看一下 part1。
- 书籍：《Unity 3D ShaderLab 开发实战详解》
- 表面着色器的写法 9 种：http://www.unity.5helpyou.com/2381.html
- 在 unity 向量空间内绘制几何 (2)：球面—重构《黑客帝国》的'上帝机器'，Deus Ex Machina http://blog.csdn.net/liu_if_else/article/details/51554940

7.3 Unity 动画

- http://www.360doc.com/content/13/0225/17/10941785_267831975.shtml
- 用好 Lua+Unity，让性能飞起来—LuaJIT 性能坑详解 https://zhuanlan.zhihu.com/p/26528101