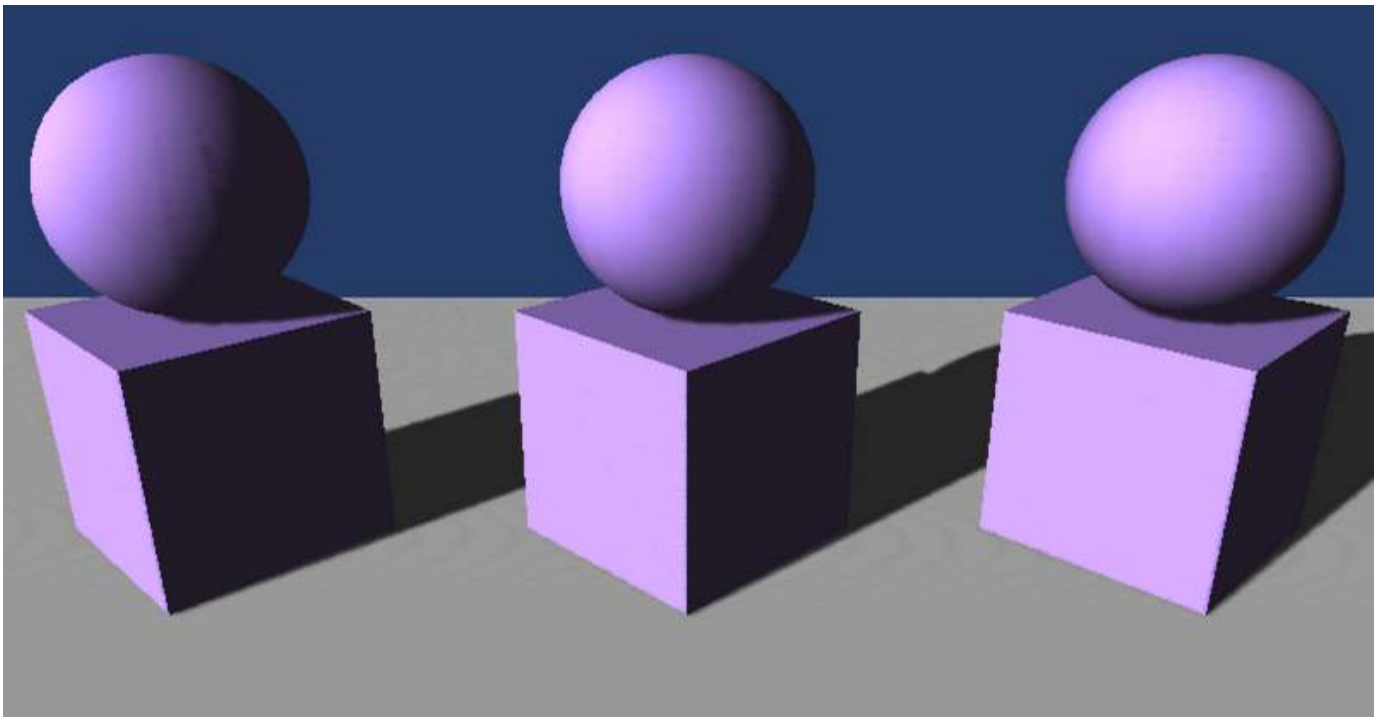
猫都能学会的Unity3D Shader入门指南（二） - zhuangyou123的专栏 - 博客频道



关于本系列

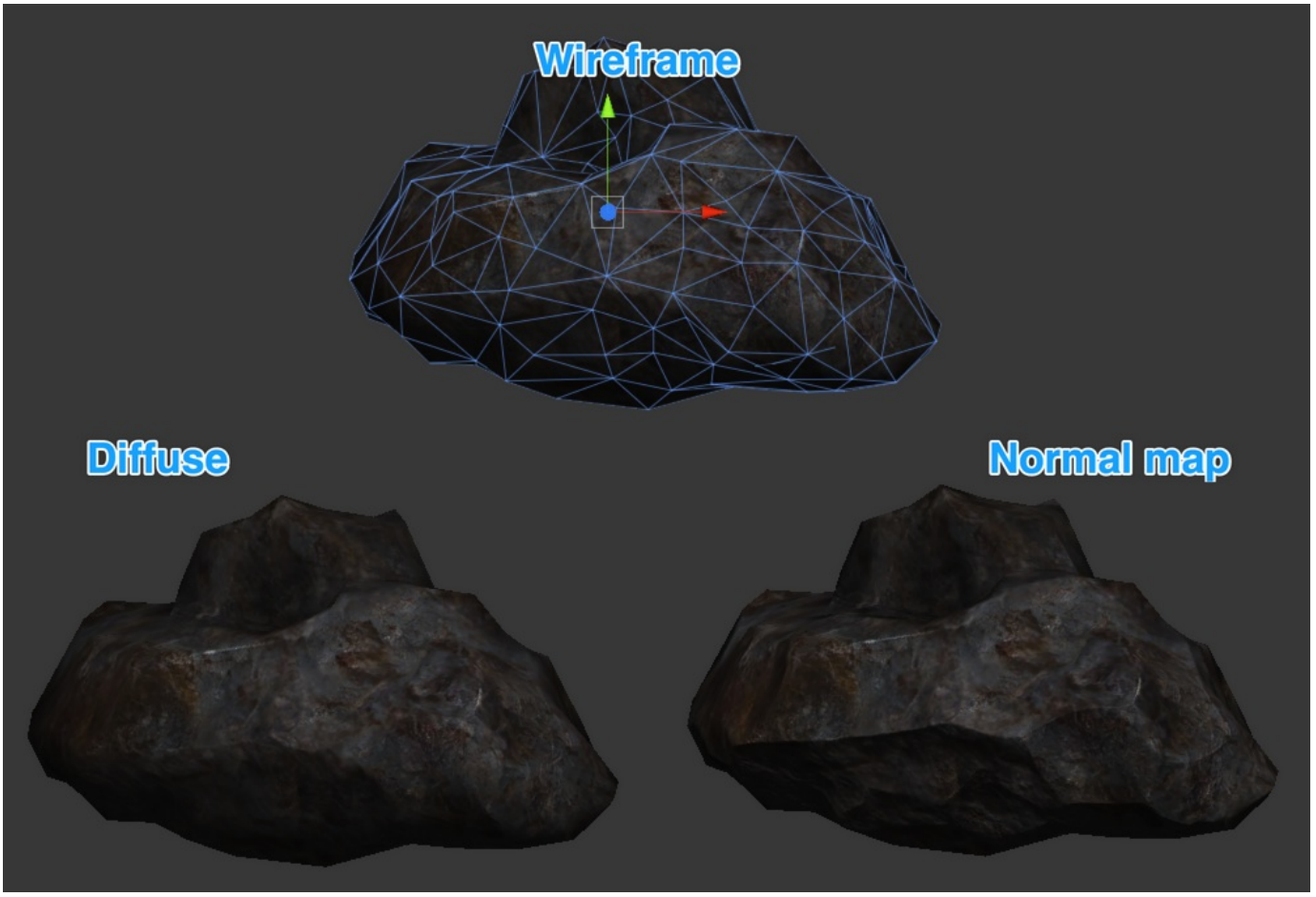
这是Unity3D Shader入门指南系列的第二篇，本系列面向的对象是新接触Shader开发的Unity3D使用者，因为我本身自己也是Shader初学者，因此可能会存在错误或者疏漏，如果您在Shader开发上有所心得，很欢迎并恳请您指出文中纰漏，我会尽快改正。在之前的开篇中介绍了一些Shader的基本知识，包括ShaderLab的基本结构和语法，以及简单逐句地讲解了一个基本的shader。在具有这些基础知识后，阅读简单的shader应该不会有太大问题，在继续教程之前简单阅读一下Unity的Surface Shader Example，以检验您是否掌握了上一节的内容。如果您对阅读大部分示例Shader并没有太大问题，可以正确地指出Shader的结构，声明和使用的话，就说明您已经准备好继续阅读本节的内容了。

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

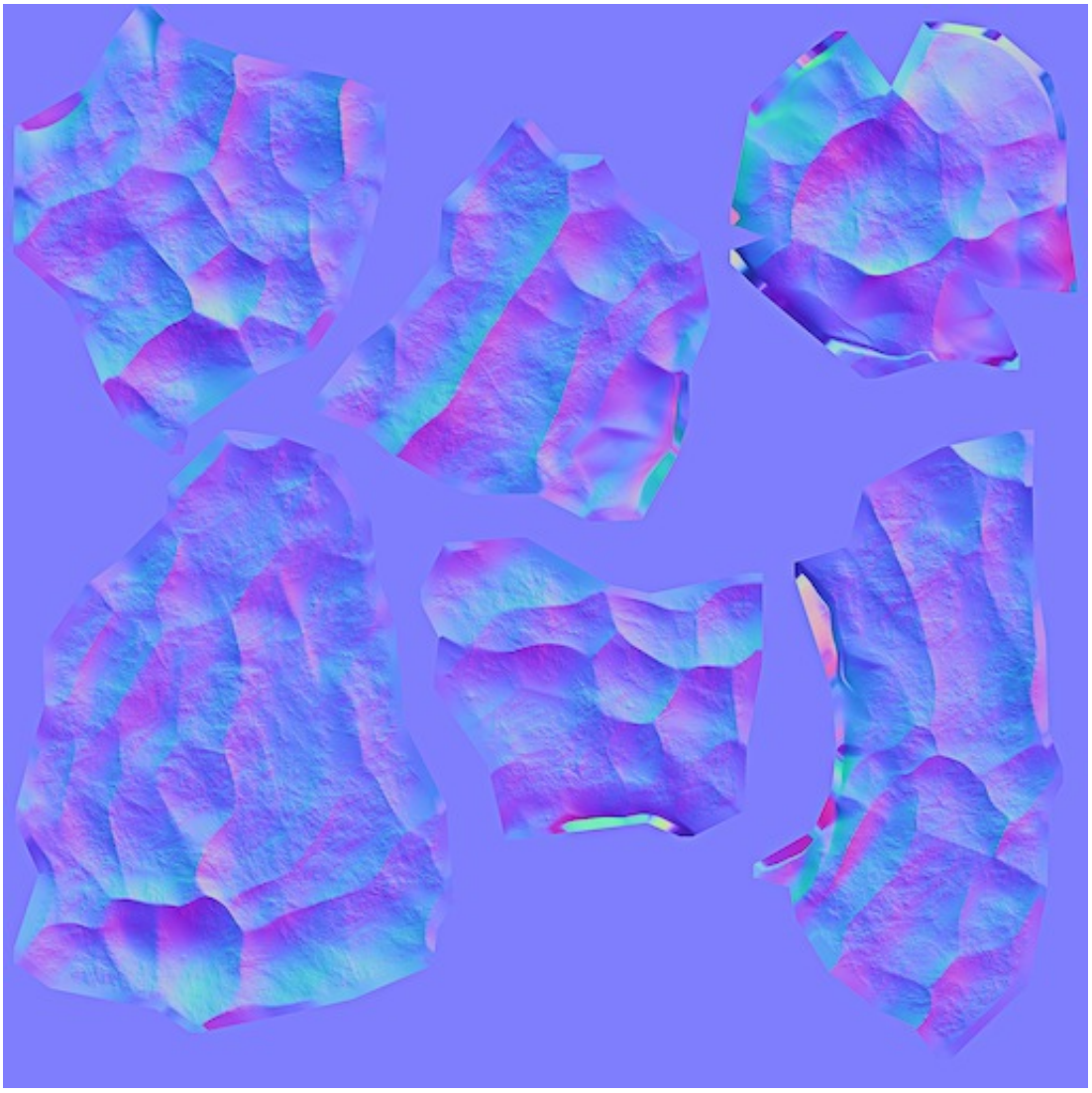
普通的贴图的基础上，再另外提供一张对应原来贴图的，可以表示渲染浓淡的贴图。通过将这张附加的表示表面凸凹的贴图的因素于实际的原贴图进行运算后，可以得到新的细节更加丰富富有立体感的渲染效果。在本节中，我们将首先实现一个法线贴图的Shader，然后对Unity Shader的光照模型进行一些讨论，并实现一个自定义的光照模型。最后再通过更改shader模拟一个石头上的积雪效果，并对模型顶点进行一些修改使积雪效果看起来比较真实。在本节结束的时候，我们就会有一个比较强大的可以满足一些真实开发工作时可用的shader了，而且更重要的是，我们将会掌握它是如何被创造出来的。

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。

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。

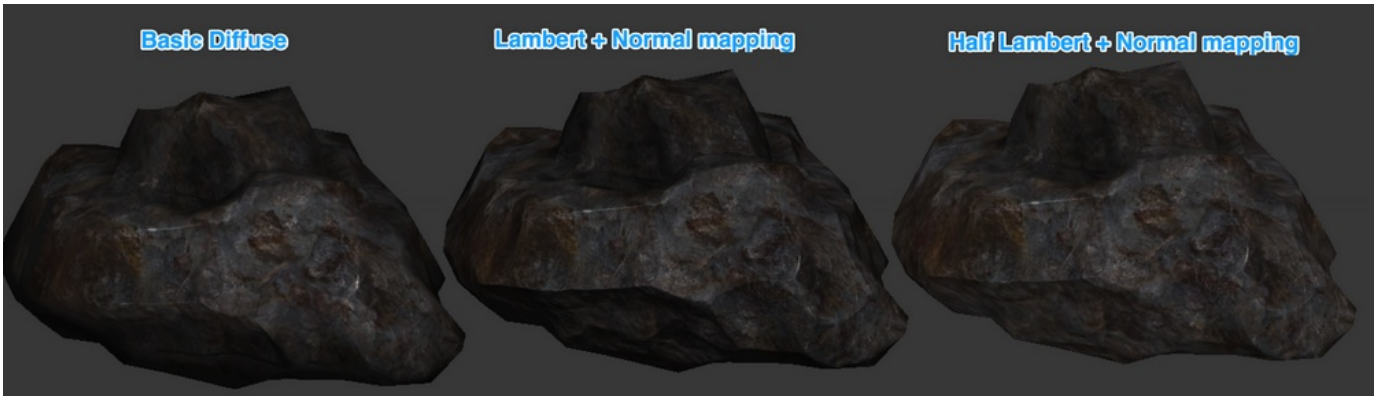
首先来看输入量，`SurfaceOutput s`这个就是经过表面计算函数surf处理后的输出，我们讲对其上的点根据光线进行处理，`fixed3 lightDir`是光线的方向，`fixed atten`表示光衰减的系数。在计算光照的代码中，我们先将输入的s的法线值（在Normal mapping中的话这个值已经是法线图中的对应量了）和输入光线进行点积（dot函数是CG中内置的数学函数，希望你还记得，可以参考这里）。点积的结果在-1至1之间，这个值越大表示法线与光线间夹角越小，这个点也就应该越亮。之后使用max来将这个系数结果限制在0到1之间，是为了避免负数情况的存在而导致最终计算的颜色变为负数，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。



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设定的默认值为(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`，如果SurfaceOutput中设定了Normal值的话，通过worldNormal可以获取当前点在世界中的法线值。详细的解说可以参见Unity的Shader文档。接下来可以改变surf函数，实装积雪效果了。

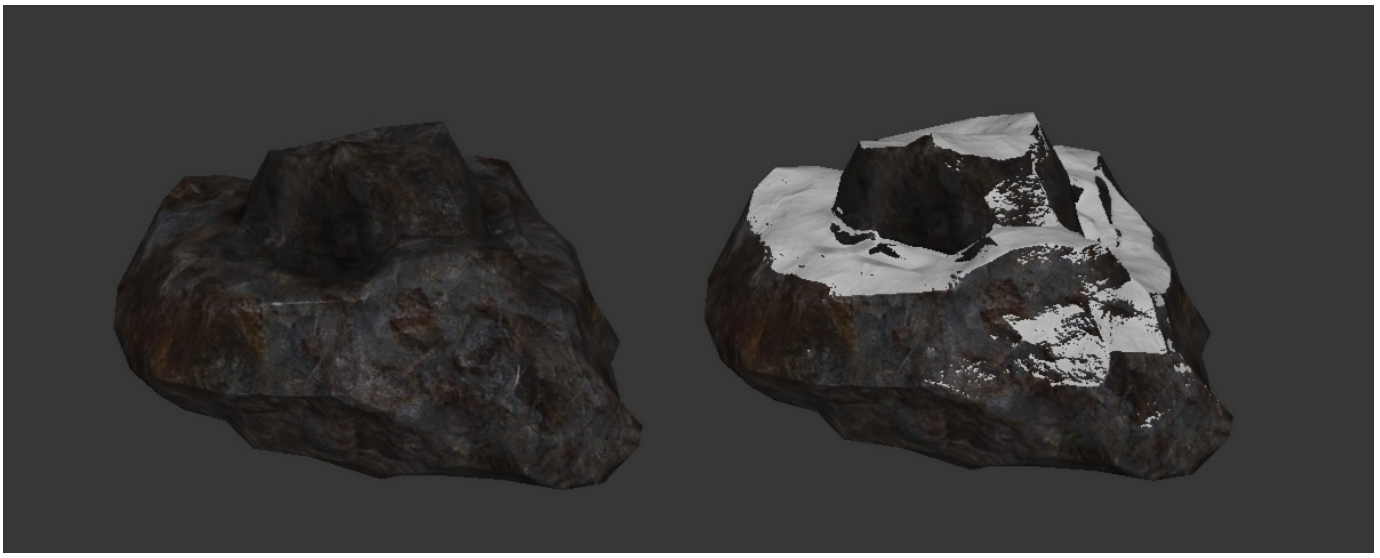
```
void surf (Input IN, inout SurfaceOutput o) {
    half4 c = tex2D (_MainTex, IN.uv_MainTex);
    o.Normal = UnpackNormal(tex2D(_Bump, IN.uv_Bump));

    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。



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`变量，表示积雪的厚度，当然我们也需要在CG段中进行声明：

```
//In Properties{…}
_SnowDepth ("Snow Depth", Range(0,0.3)) = 0.1
```

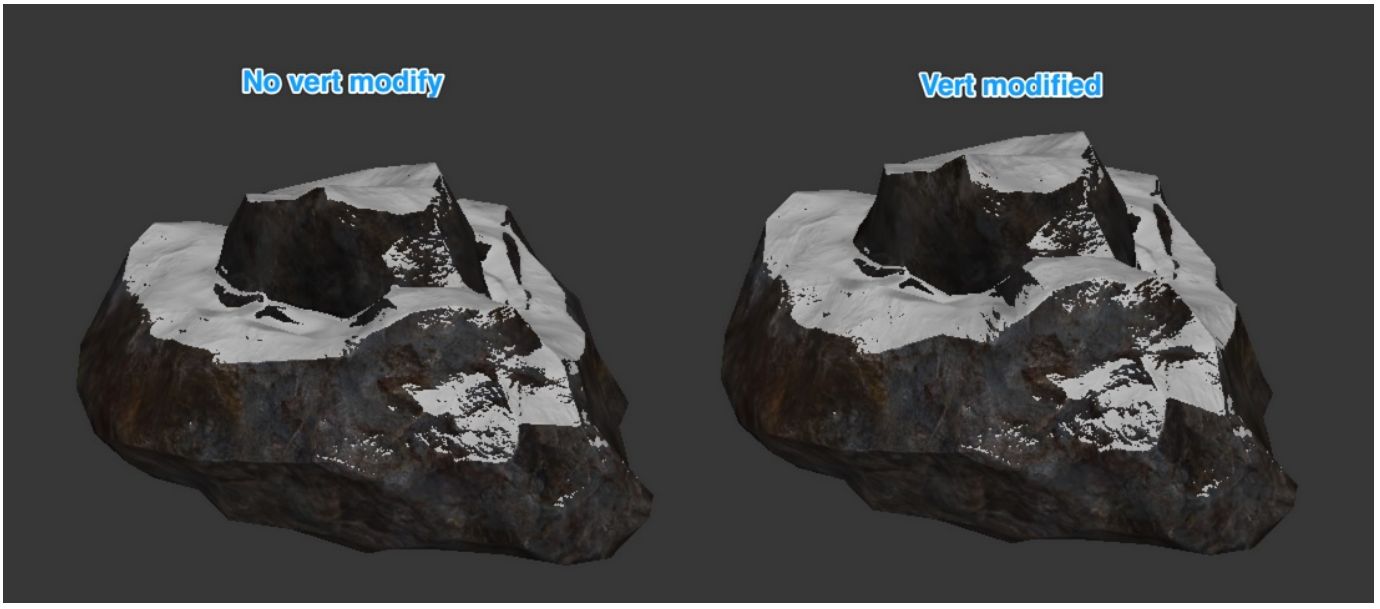
```
//In CG declare
float _SnowDepth;
```

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_Object2World是Unity 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。



这节就到这里吧。本节中实现的Shader可以在这里找到完整版本进行参考，希望大家周末愉快～

原文：http://onevcat.com/2013/08/shader-tutorial-2/