[2024.02.02]Unity中的基础纹理

前言

纹理最初的目的就是使用一张图片来控制模型的外观，使用**纹理映射（textur mapping）**可以把一张纹理附着在模型表面，**逐纹素（texel）**地控制模型颜色。在美术人员建模的时候，通常会在建模软件中，利用纹理展开技术，把纹理映射坐标（texture-mapping coordinates）存储在每个顶点上。纹理映射坐标定义了该顶点在纹理中对应的2D坐标，通常，这些坐标使用一个二维变量(u,v)来表示，其中u是横向坐标，v是纵向坐标，因此纹理映射坐标也被称为UV坐标。

尽管纹理的大小可以是256x256或1024x1024，但顶点UV坐标的范围通常都被归一化到[0,1]的范围内。但需要注意，纹理采样时使用的纹理坐标不一定是在[0,1]范围内。实际上，这种不在[0,1]范围内的纹理坐标有时会非常有用，与之关系紧密的是纹理的平铺模式，它将决定渲染引擎在遇到不在[0,1]范围内的纹理坐标时，如何进行纹理采样。

单纹理

本例中，我们继续使用Blinn-Phong光照模型来计算光照。

着色器

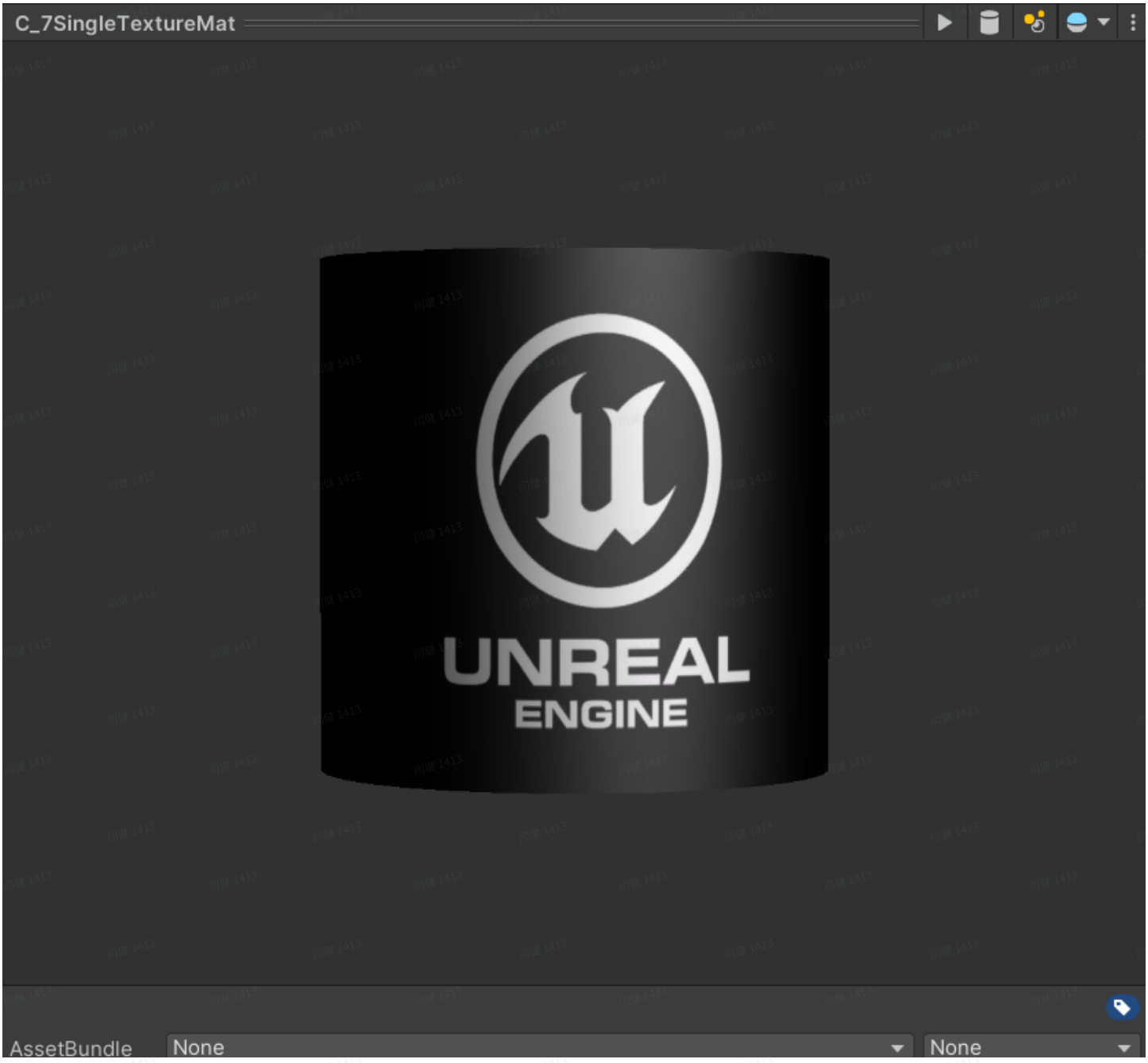
```
shader "Unity Shaders Book/C_7/C_7SingleTexture"
{
    Properties
    {
        // 叠加的颜色，默认为白色
        _Color ("Color Tint", Color) = (1, 1, 1, 1)
        // 纹理，类型为2D，没有纹理时，默认用白色覆盖物体的表面
        _MainTex("Main Tex", 2D) = "white" {}
        // 高光颜色，默认为白色
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        // 光泽度，影响高光反射区域的大小
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }

    SubShader
    {
        Pass
        {
            // 指明当前Pass的光照模式
            Tags { "LightMode" = "ForwardBase"}
```

```
21
22              CGPROGRAM
23
24              #pragma vertex vert
25              #pragma fragment frag
26
27              // 为了使用光照相关的内置变量（如：_LightColor0 光照颜色）
28              #include "Lighting.cginc"
29
30              /* 定义属性变量 */
31              fixed4 _Color;
32              sampler2D _MainTex;
33              // 与_MainTex配套的纹理缩放（scale）和平移（translation），在材质面板的纹
    理属性中可以调节
34              // 命名规范为：纹理变量名 + "_ST"
35              // _MainTex_ST.xy 存储缩放值
36              // _MainTex_ST.zw 存储偏移值
37              float4 _MainTex_ST;
38              fixed4 _Specular;
39              float _Gloss;
40
41              struct a2v
42              {
43                  float4 vertex : POSITION;
44                  float3 normal : NORMAL;
45                  // 存储模型的第一组纹理坐标，可以理解为_MainTex对应的原始纹理坐标
46                  float4 texcoord : TEXCOORD0;
47              };
48
49              struct v2f
50              {
51                  float4 pos : SV_POSITION;
52                  float3 worldNormal : TEXCOORD0;
53                  float3 worldPos : TEXCOORD1;
54                  // 存储纹理坐标的UV值，可在片元着色器中使用该坐标进行纹理采样
55                  float2 uv : TEXCOORD2;
56              };
57
58              v2f vert(a2v v)
59              {
60                  v2f o;
61
62                  // 将顶点坐标由模型空间转到裁剪空间
63                  o.pos = UnityObjectToClipPos(v.vertex);
64                  // 将法线由模型空间转到世界空间
65                  o.worldNormal = UnityObjectToWorldNormal(v.normal);
66                  // 将顶点坐标由模型空间转到世界空间
```

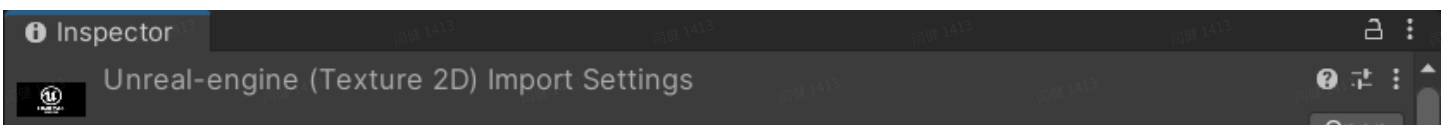
```
67                  o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
68
69                  // 通过缩放和平移后的纹理UV值
70                  o.uv = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
71                  // 也可以调用内置宏TRANSFORM_TEX，得到缩放和平移后的纹理UV值，与上面的
    计算逻辑是一致的
72                  // 内置宏的定义：
73                  // #define TRANSFORM_TEX(tex, name) (tex.xy * name##_ST.xy +
    name##_ST.zw)
74                  o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
75
76                  return o;
77              }
78
79              float4 frag(v2f i) : SV_Target
80              {
81                  fixed3 worldNormal = normalize(i.worldNormal);
82                  // 调用内置函数UnityWorldSpaceLightDir，
83                  // 得到当前坐标点在世界空间下的光照方向，并进行归一化
84                  fixed3 worldLightDir =
    normalize(UnityWorldSpaceLightDir(i.worldPos));
85
86                  // 通过内置函数tex2D，根据当前坐标点的UV值，对纹理进行采样拿到纹理颜色
87                  // 并和颜色属性的乘积得到反射率albedo
88                  fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;
89
90                  // 使用内置变量UNITY_LIGHTMODEL_AMBIENT，
91                  // 得到环境光的颜色，并和反射率相乘得到环境光部分
92                  fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;
93
94                  // 使用内置变量_LightColor0得到光照颜色，乘以反射率，得到光照部分，
95                  // 再根据兰伯特定律漫反射公式得到漫反射部分
96                  fixed3 diffuse = _LightColor0.rgb * albedo * max(0,
    dot(worldNormal, worldLightDir));
97
98                  // 使用内置函数UnityWorldSpaceViewDir得到当前坐标点的视角方向，并进行
    归一化
99                  fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
100                 // 基于Blinn-Phong光照模型，得到中间矢量
101                 fixed3 halfDir = normalize(worldLightDir + viewDir);
102                 // 基于Blinn-Phong光照模型的公式，得到高光反射部分
103                 fixed3 specular = _LightColor0.rgb * _Specular.rgb *
    pow(max(0, dot(worldNormal, halfDir)), _Gloss);
104
105                 // 环境光 + 漫反射 + 高光反射，得到最终的颜色值
106                 return fixed4(ambient + diffuse + specular, 1.0);
107             }
```

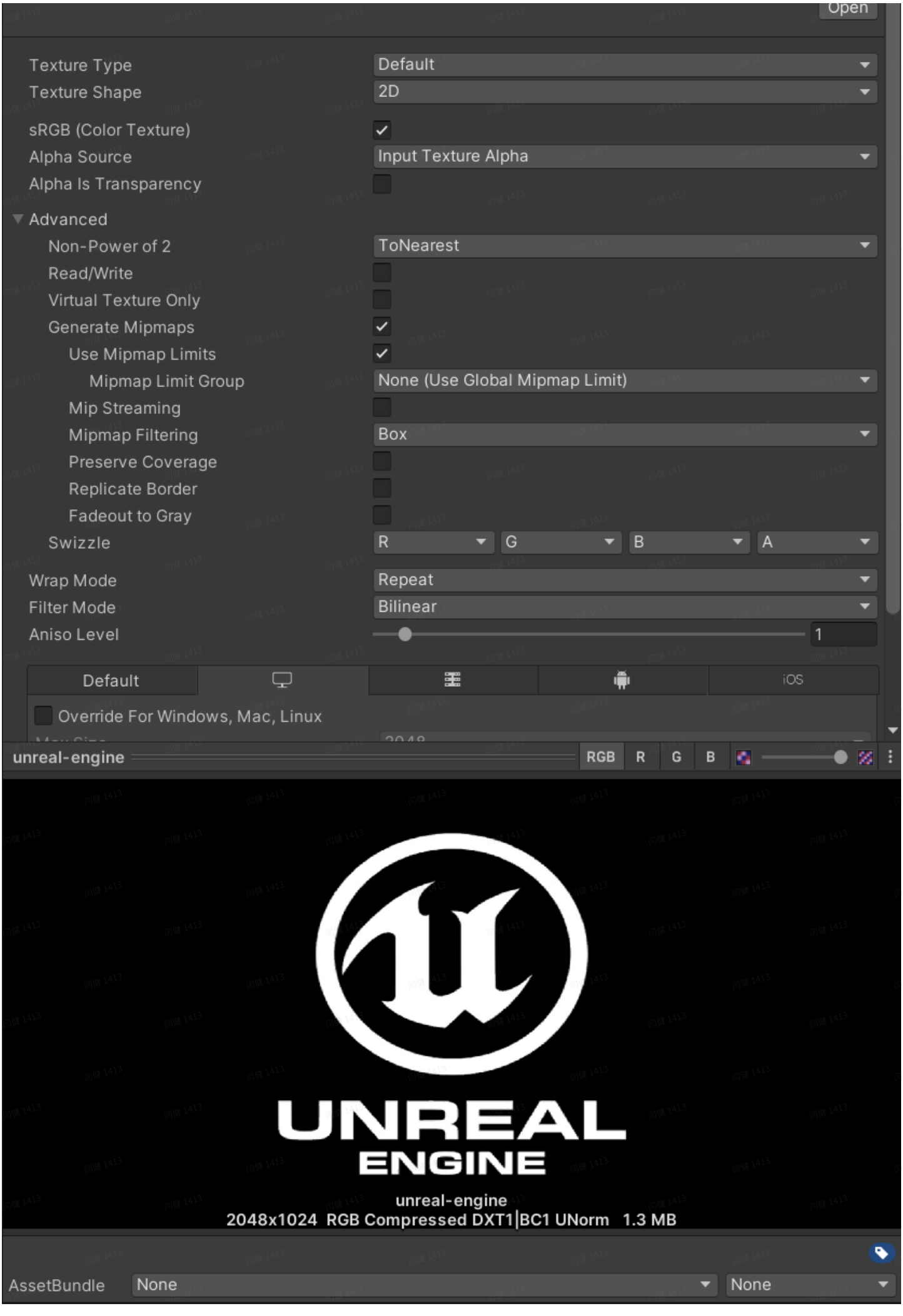
```
108
109             ENDCG
110         }
111     }
112
113     // 用系统内置的高光shader作为兜底
114     Fallback "Specular"
115 }
```

用隔壁引擎的图来代替下素材...



Texture Import Settings




Open
Texture Type
Default
Texture Shape
2D
sRGB (Color Texture)
Alpha Source
Input Texture Alpha
Alpha Is Transparency
Advanced
Non-Power of 2
ToNearest
Read/Write
Virtual Texture Only
Generate Mipmaps
Use Mipmap Limits
Mipmap Limit Group
None (Use Global Mipmap Limit)
Mip Streaming
Mipmap Filtering
Box
Preserve Coverage
Replicate Border
Fadeout to Gray
Swizzle
R
G
B
A
Wrap Mode
Repeat
Filter Mode
Bilinear
Aniso Level
1
Default
iOS
Override For Windows, Mac, Linux
unreal-engine
RGB
R
G
B
UNREAL
ENGINE
unreal-engine
2048x1024 RGB Compressed DXT1|BC1 UNorm 1.3 MB
AssetBundle
None
None

当我们向Unity导入一张纹理后，可以在属性面板上调整它的选项，Texture Type指的是要创建的纹理类型，这里我们是默认的，Texture 2D，还有其他类型的纹理属性如下：

属性	功能
Default	最常用的纹理设置，提供了大多数的纹理导入属性。
Normal map	法线贴图纹理格式化文理资源，它适用于实时法线贴图。
Editor GUI and Legacy GUI	编辑器GUI和传统GUI纹理类型格式化纹理资源，所以它适合于HUD和GUI控件。
Sprite (2D and UI)	Sprite (2D和UI)纹理类型格式化纹理资源，所以它适合在2D应用中作为Sprite使
Cursor	光标纹理类型格式化纹理资源，所以它适合用作自定义鼠标光标。
Cookie	Cookie纹理类型格式化纹理资源，所以它适合在内置渲染管道中作为轻Cookie使
光照贴图	Lightmap纹理类型格式化纹理资源，所以它适合用作Lightmap。该选项允许编道)。
Directional Lightmap	方向光照贴图纹理类型格式化纹理资源，所以它适合用作方向光照贴图。
Shadowmask	阴影遮罩纹理类型格式化纹理资源，所以它适合用作Shadowmask。
Single Channel	单通道纹理类型格式化纹理资源，所以它只有一个通道。

我们这里单看Default类型下，属性面板的一些额外设置：

在原书中，提到了 `Alpha from Grayscale` 的复选框，勾选之后，透明通道的值将会由每个像素的灰度图生成。这点，在高版本内被整合进了 `Alpha Source` ：

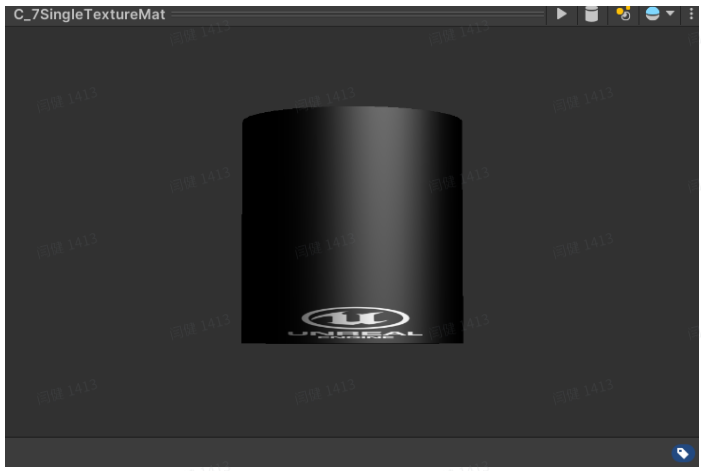
Alpha Source		Specifies how Unity generates the alpha value for the texture asset
	None	The texture asset doesn’ t have an alpha channel, whether or not t
	Input Texture Alpha	Unity applies the alpha channel from the texture source file to the t channel.
	From Gray Scale	Unity generates the alpha channel for the texture asset from the ave

除此之外，原书中还提到了另一个属性是 `Wrap Mode` ，它决定了当纹理坐标超过[0,1]范围后，会如何被平铺；它可供选择的模式有以下几种：

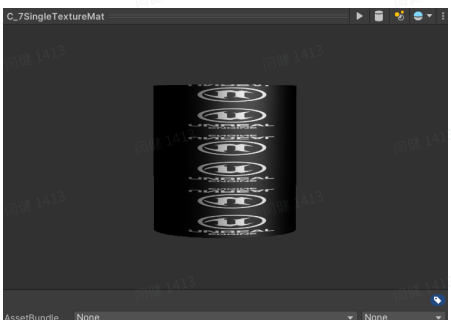
Wrap Mode		Specifies how the texture behaves when it tiles.
	Repeat	Repeats the texture in tiles.
	Clamp	Stretches the texture’ s edges.
	Mirror	Mirrors the texture at every integer boundary to create a repeating pattern.
	Mirror Once	Mirrors the texture once, then clamps it to edge pixels. **Note**: Some mobile devices don’ t support Mirror Once. In this case, Unity use instead.
	Per-axis	Provides options you can use to individually control how Unity wraps textures axis.

在Repeat模式下，如果纹理坐标超过了1，那么它的整数部分将会被舍弃，而直接使用小数部分进行采样，这样的结果是纹理将会不断重复。

在Clamp模式下，如果纹理坐标大于1，那么将会截取到1，如果小于0，那么将会截取到0；通过修改纹理的Wrap Mode，也可以直观感受到材质球发生的变化如下：



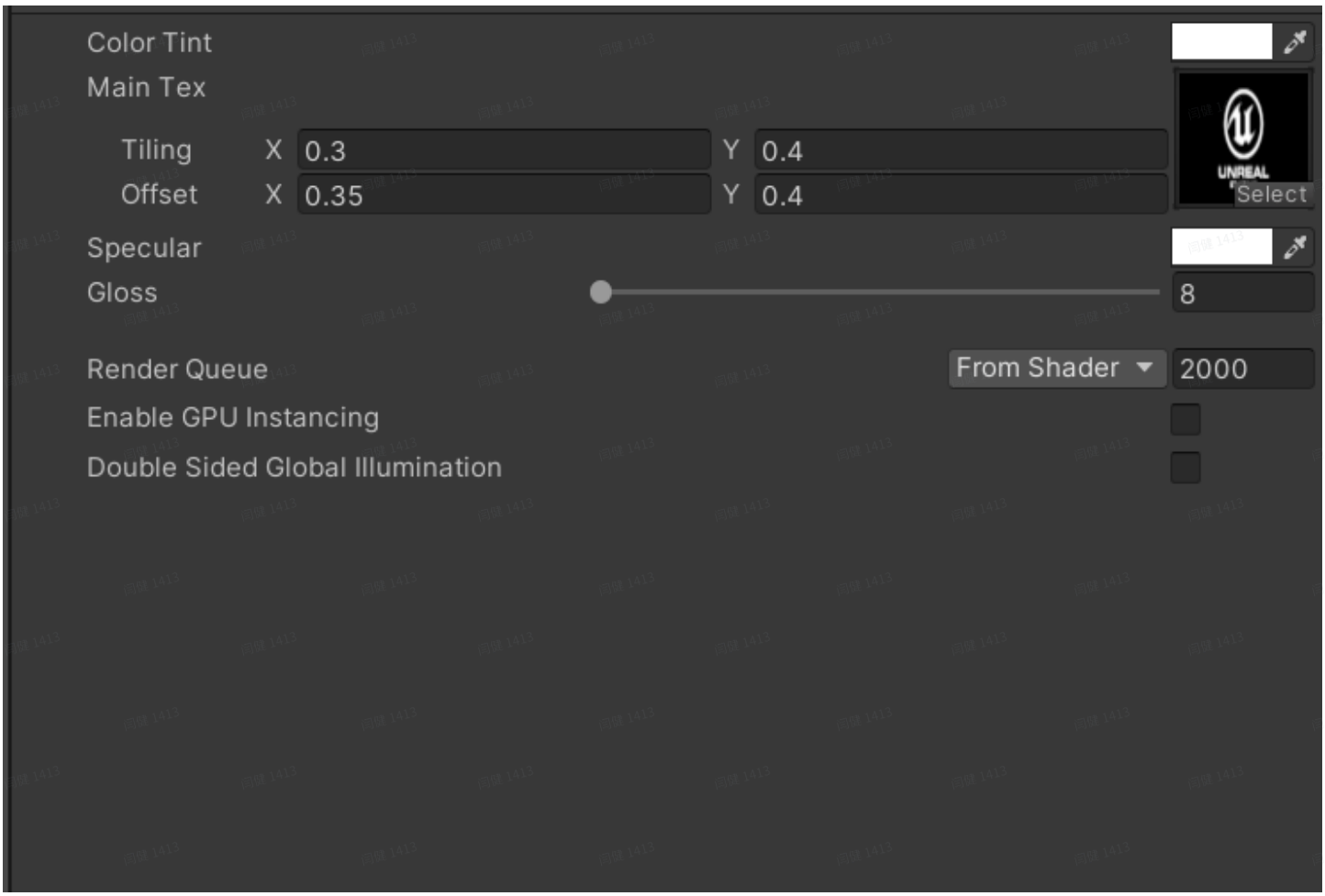
此外还有Mirror、Mirror Once、Per-axis(是一个复合选项)，也可以从下面直观感受到变化（按列举顺序）：

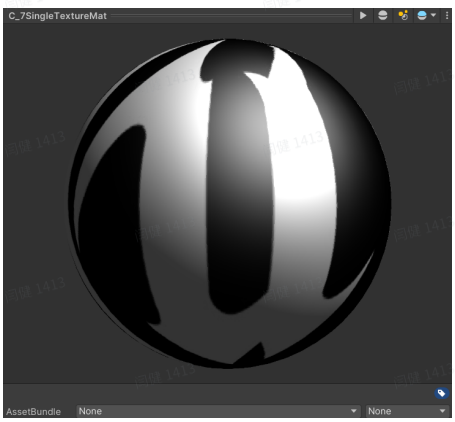


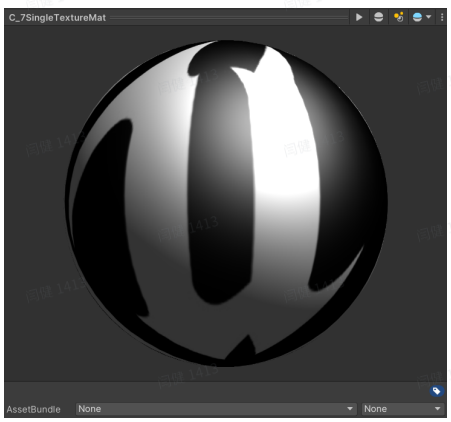
还有Filter Mode，它决定了当纹理由于变换而产生拉伸时，将会采用哪种滤波模式(这么多年了也还是这仨)：

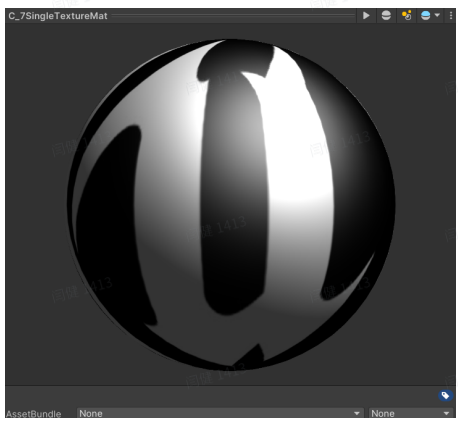
Filter Mode		Specifies how Unity filters the texture when the texture stretches during 3D transformations.
	Point (no filter)	The texture appears blocky up close.
	Bilinear	The texture appears blurry up close.
	Trilinear	Like **Bilinear**, but the texture also blurs between the different MIP levels.

他们得到的图片滤波效果是以此提升的，但消耗也是依次增大的。**纹理滤波**会影响放大或缩小纹理时得到的图片质量；我们对材质球进行一定量的缩放和偏移，来看看不同的模式下，纹理的质量：





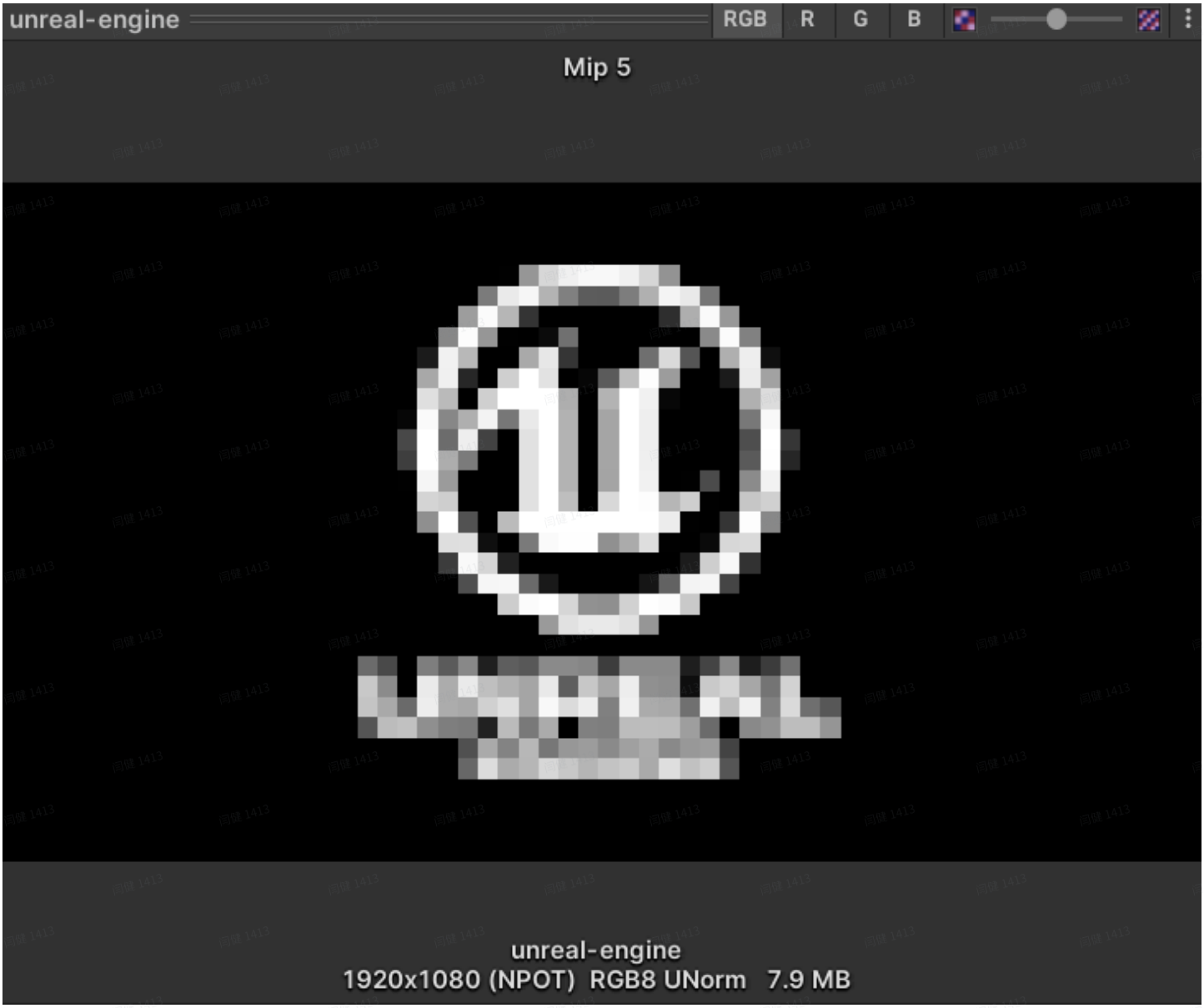




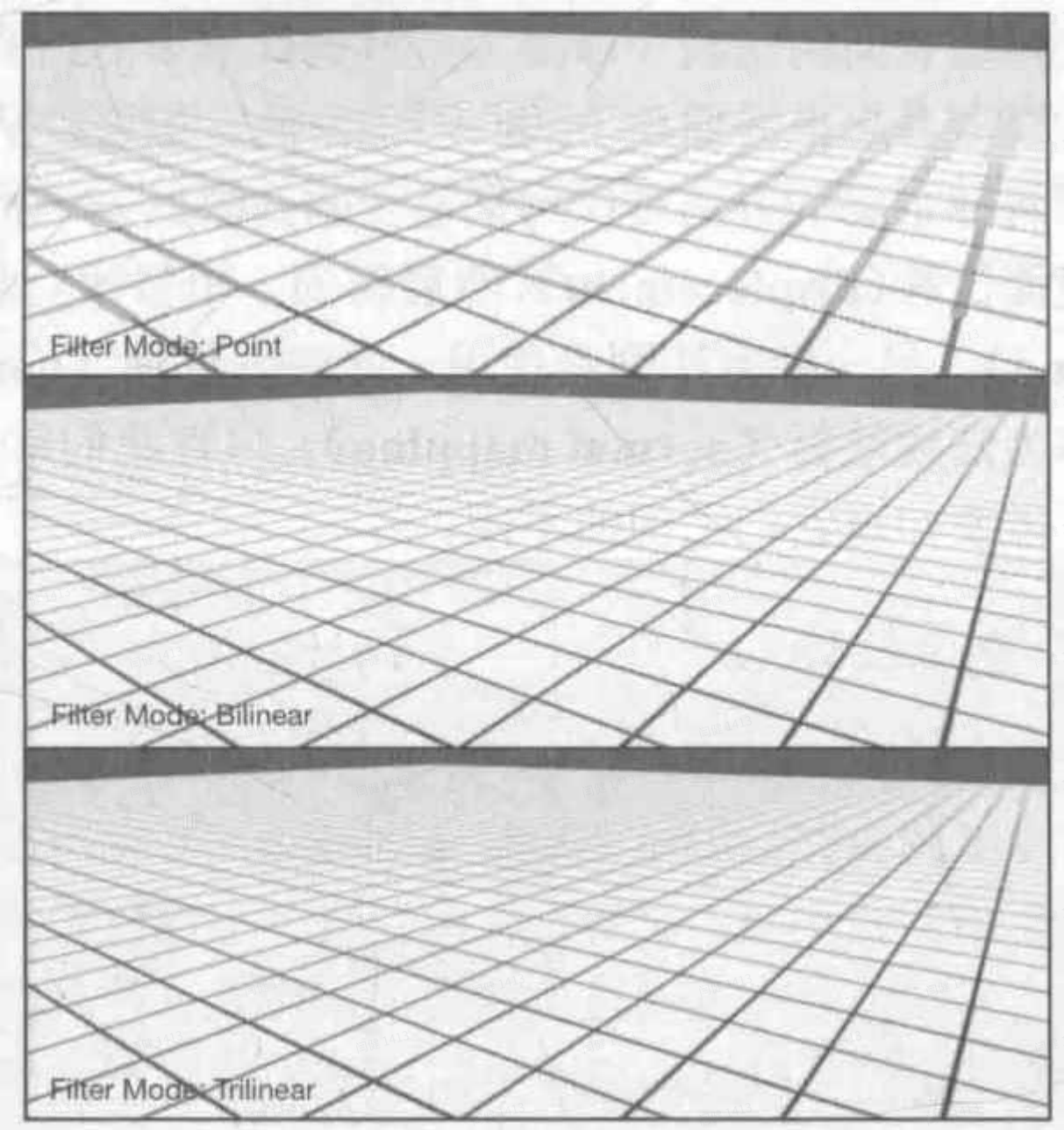
MipMap

纹理缩小的过程比放大更加复杂一些，此时原纹理中的多个像素将会对应同一个目标像素。纹理缩放更加复杂的原因在于我们往往需要处理抗锯齿问题；一个最常用的方法就是**多级渐远纹理（mipmapping）技术**。

mipmap是将原纹理提前用滤波处理来得到很多更小的图像，形成了一个图像金字塔，每一层都是对上一层图像降采样的结果。这样在实时运行时，就可以快速得到结果像素；但缺点是通常会多占用33%的内存空间，也就是空间换时间了。



以一个网格场景为例：



在内部实现上，`Point` 模式使用了**最近邻滤波**，在放大或者缩小时，它的采样像素数量通常只有一个；

`Bilinear` 滤波则使用了线性滤波，对于每个像素目标，它会找到4个临近像素，然后对他们进行线性插值混合后得到最终像素，因此图像看起来像是被模糊了。

而 `Trilinear` 滤波几乎是和 `Bilinear` 一样，只是 `Trilinear` 还会在mipmap之间进行混合。如果一张纹理没有开mipmap，那么 `Trilinear` 和 `Bilinear` 的效果就是一样的。

一般我们会选择 `Bilinear` 滤波，如果有一种类似于棋盘的纹理，我们希望它是像素风的，这时就会选择 `Point` 模式。

纹理尺寸

2022上，Unity允许使用的最大纹理尺寸已经到16384了，但我觉得一般不会有人敢用这么大的纹理。导入的纹理，在长和宽上都应该是2的整数次幂，如 `2、4、8、16、32、64、128、256、512、`

`1024、2048、4096` 等。如果使用了像图中的非对齐（非2的幂次方 Non Power Of Tow，NPOT）的纹理，那么这些纹理会占用更多的内存空间，且读取效率也会下降。

凹凸映射

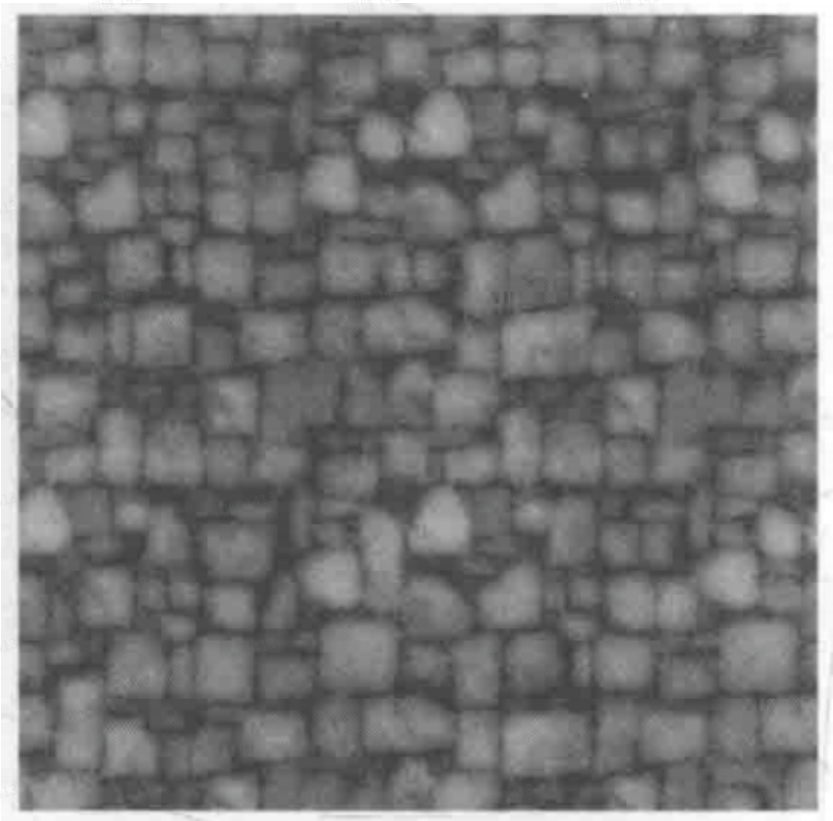
纹理的另一种常见的应用就是**凹凸映射（bump mapping）**。凹凸映射的目的是使用一张纹理来修改模型表面的法线，以便为模型提供更多的细节。但这种方法不会真的改变模型的顶点位置，只是让模型看起来好像是凹凸不平的。有两种主要的方法可以用来进行凹凸映射：

1. 使用一张**高度纹理（height map）**来模拟**表面位移（displacement）**，然后得到一个修改后的法线值，也被称为**高度映射（height mapping）**。
2. 或者使用一张**法线纹理（normal map）**来直接存储表面法线，也称为**法线映射（normal mapping）**。

高度纹理

高度图中存储的是强度值（intensity），它用于表示模型表面局部的海拔高度。因此，颜色越浅越表明该位置的表面越向外凸起，而颜色越深越表明该位置越向里凹。这种方法的好处是非常直观，我们可以从高度图中明确地知道一个模型表面的凹凸情况，但缺点是计算更加复杂，在实时计算时，不能直接得到表面法线，而是需要由像素的灰度值计算而得，因此需要消耗更多的性能。

高度图通常会和法线映射一起使用，用于给出表面凹凸的额外信息，换言之，我们通常会使用法线映射来修改光照。



法线纹理

而法线纹理中存储的就是表面的法线方向。由于法线方向的分量范围在[-1,1]，而像素的分量范围为[0,1]，因此我们需要做一个映射，通常使用的映射就是：

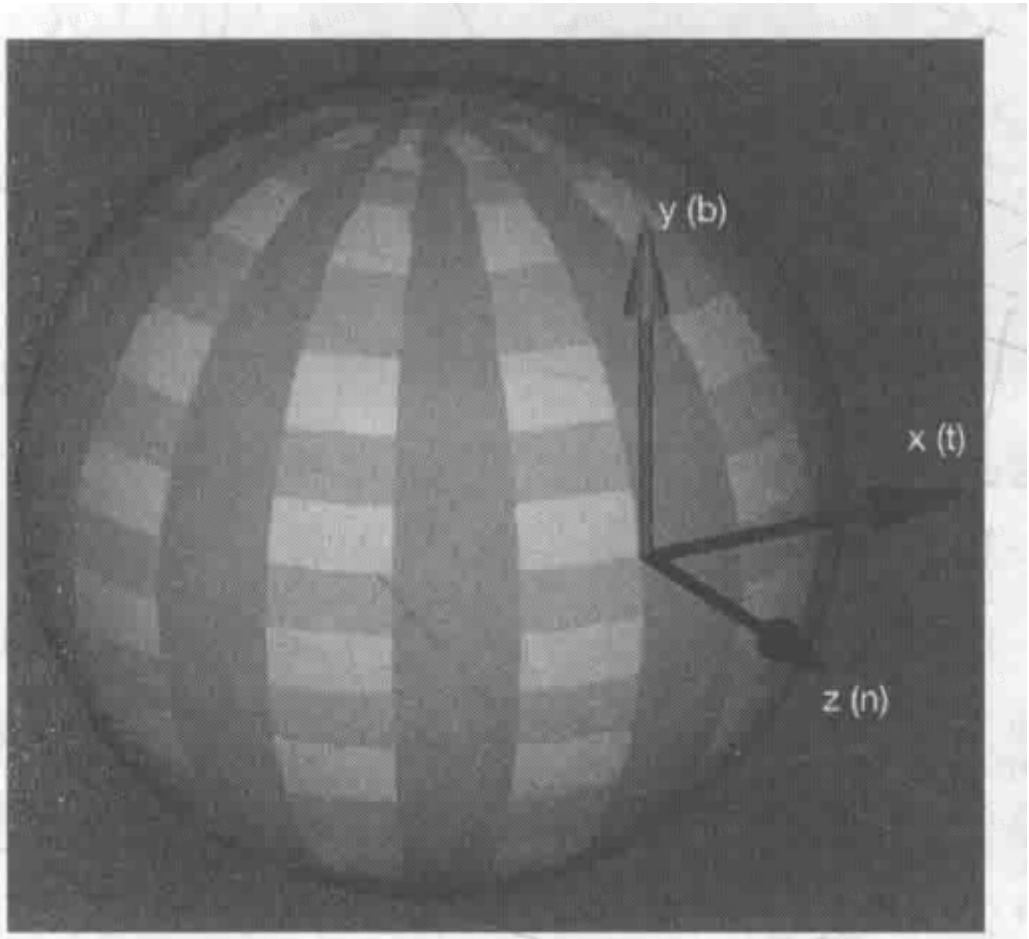
$$pixel = \frac{normal + 1}{2}$$

这就要求，我们在Shader中对法线纹理进行纹理采样后，还需要对结果进行一次反映射的过程，以得到原先的法线方向。反映射的过程实际上就是使用上面映射函数的逆函数：

$$normal = pixel \times 2 - 1$$

然而，由于方向是相对于坐标空间来说的，法线纹理中存储的法线方向也是有区别的。对于模型顶点自带的法线，他们是定义在模型空间中的，一种直接的方式是，可以直接将修改后的模型空间中的表面法线存储在一张纹理中，这种纹理被称为是**模型空间的法线纹理（object-space normal map）**。

然而，在实际制作中，我们往往会采用另一种坐标空间，即模型顶点的**切线空间（tangent space）**来存储法线。对于每个模型的每个定点，它都有一个属于自己的切线方向（t），而y轴可由法线和切线叉积而得，也被称为是副切线（bitangent，b）或副法线。



▲图 7.12　模型顶点的切线空间。其中，原点对应了顶点坐标，x 轴是切线方向（t），y 轴是副切线方向（b），z 轴是法线方向（n）

这种纹理被称为是**切线空间的法线纹理（tangent-space normal map）**。可以回忆一下法线纹理和切线纹理的样子；

在模型空间下，法线纹理看起来是"五颜六色"的，这是因为所有法线所在的坐标空间是同一个坐标空间，即模型空间，而每个点存储的法线方向是各异的，有的是(0,1,0)，经过映射后存储到纹理中就对应了RGB(0.5,1,0.5)浅绿色；有的是(0,-1,0)，经过映射后存储到纹理中就对应了(0.5,0,0.5)紫色。

而切线空间下的法线纹理看起来几乎就全部是浅蓝色的。这是因为，每个法线方向坐在的坐标空间是不一样的，即是表面每点各自的切线空间。这种法线纹理其实就是存储了每个点在各自的切线空间中的法线扰动方向。也就是说，如果一个点的法线方向不变，那么在它的切线空间中，新的法线方向就是z轴方向，即值为(0,0,1)，经过映射后存储在纹理中就对应了RGB(0.5,0.5,1)浅蓝色。这个颜色就是法线纹理中大片的蓝色，这些蓝色实际上说明顶点的大部分法线是和模型本身法线一样的，不需要改变。

总体来说，存储法线只是手段，我们最终的目的还是为了后续计算光照，而存储的坐标系选择意味着后续坐标系转换的步骤。比如如果选择了切线空间，后续就需要把从法线纹理中得到的法线方向从切线空间转换到世界空间（或其他空间）中。

使用模型空间来存储法线的优点如下：

1. 简单且直观，我们甚至都不需要模型原始的法线和切线信息，计算也更少。但如果要生成切线空间下的法线纹理，由于模型的切线一般是和UV方向相同，因此想要得到效果比较好的法线映射就要求纹理映射也是连续的。
2. 在纹理坐标的缝合处和边角部分，可见的缝隙会较少，因为模型空间下的法线纹理存储的是同一坐标系下的的法线信息，因此在边界处通过插值得到的法线可以平滑变换；而切线空间下的法线信息都是依靠纹理坐标的方向得到的结果，可能会在边缘处造成可见的"缝合迹象"。

但使用切线空间有更多的优点：

1. 自由度高，因为模型空间下记录的是绝对法线信息，仅可用于创建它时的那个模型；而切线空间下的法线纹理记录的是相对法线信息，即便套用一个完全不同的网格，也可堪一用。
2. 可以进行UV动画。比如我们可以移动一个纹理的UV坐标来实现一个凹凸移动的效果；但使用模型空间下的法线纹理将会得到完全错误的结果，原因是绝对法线信息。
3. 可以重用法线纹理，比如一个砖块，我们仅用一张法线纹理就可以用到6个面上，原因同上。
4. 可压缩。由于切线空间下的法线纹理中的法线的Z方向总是正方向，因此我们可以仅存储XY方向，而推导得到Z方向。而模型空间下的法线纹理，由于每个方向都是可能的，因此必须存储3个方向的值，不可压缩。

实践

我们需要计算光照模型中统一各个方向矢量所在的坐标空间。由于法线纹理中存储的法线是切线空间下的方向，因此我们通常有两种选择：

1. 一种选择是在切线空间下进行光照计算，此时我们需要把光照方向、视角方向变换到切线空间下；
2. 另一种选择是在世界空间下进行光照计算，此时我们需要把采样的到的法线方向变换到世界空间下，再和世界空间下的光照方向和视角方向进行计算。

从效率上来说，第一种方法往往要优于第二种方法，因为我们可以再顶点着色器中就完成对光照方向和视角方向的变换；而第二种方法由于要先对法线进行纹理采样，所以变换过程必须在片元着色器中实现，意味着我们需要在片元着色器中进行一次矩阵操作。

但从通用性上来说，第二种方法要优于第一种方法，因为有时我们需要在世界空间下进行一些计算，例如在使用Cubemap进行环境映射时，我们就需要把法线方向变换到世界空间下。

在切线空间下计算

首先在切线空间下计算光照模型。基本思路是：在片元着色器中通过纹理采样得到切线空间下的法线，然后再与切线空间下的视角方向、光照方向等进行计算，得到最终的光照结果。

为此，我们首先需要在顶点着色器中把视角方向和光照方向从模型空间变换到切线空间中，即我们需要知道从模型空间到切线空间的变换矩阵。这个变换矩阵的逆矩阵，即从切线空间到模型空间的变换矩阵是非常容易求得的，我们在顶点着色器中，按切线（x轴）、副切线（y轴）、法线（z轴）的顺序**按列排列**即可得到。

因为，如果一个变换中仅存在平移和旋转变换，那么这个变换的逆矩阵就等于它的转置矩阵；而从切线空间到模型空间的变换正是符合这样要求的变换。因此，从模型空间到切线空间的变换矩阵，就是从切线空间到模型空间的变换矩阵的转置矩阵，我们把切线（x轴）、副切线（y轴）、法线（z轴）的顺序**按行排列**即可得到。

```
shader "Unity Shaders Book/C_7/C_7NormalMapTangentSpace"
{
    Properties
    {
        // 叠加的颜色，默认为白色
        _Color ("Color Tint", Color) = (1, 1, 1, 1)
        // 纹理，类型为2D，没有纹理时，默认用白色覆盖物体的表面
        _MainTex("Main Tex", 2D) = "white" {}
        //法线纹理，使用bump作为它的默认值，是Unity内置的法线纹理
        _BumpMap("Normal Map", 2D) = "bump" {}
        //用于控制凹凸程度，当它为0是，该法线不会对光照产生任何影响。
        _BumpScale("Bump Scale", Float) = 1.0
        // 高光颜色，默认为白色
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        // 光泽度，影响高光反射区域的大小
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }

    SubShader
    {
        Pass
        {
            // 指明当前Pass的光照模式
            Tags { "LightMode" = "ForwardBase"}

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            // 为了使用光照相关的内置变量（如：_LightColor0 光照颜色）
            #include "Lighting.cginc"

            /* 定义属性变量 */
            fixed4 _Color;
            sampler2D _MainTex;
            // 与_MainTex配套的纹理缩放（scale）和平移（translation），在材质面板的纹理属性中可以调节
            // 命名规范为：纹理变量名 + "_ST"
            // _MainTex_ST.xy 存储缩放值
            // _MainTex_ST.zw 存储偏移值
```

```
41              float4 _MainTex_ST;
42              sampler2D _BumpMap;
43              //与_BumpMap配套的纹理缩放平移变量
44              float4 _BumpMap_ST;
45              float _BumpScale;
46              fixed4 _Specular;
47              float _Gloss;
48
49              struct a2v
50              {
51                  float4 vertex : POSITION;
52                  float3 normal : NORMAL;
53                  //切线空间是切线和法线构建出的坐标空间，切线是float4，因为我们需要使用tangent.w来决定切线空间中的第三个坐标轴——副切线的方向性。
54                  float4 tangent : TANGENT;
55                  // 存储模型的第一组纹理坐标，可以理解为_MainTex对应的原始纹理坐标
56                  float4 texcoord : TEXCOORD0;
57              };
58              //要在顶点着色器中计算切线空间下的光照和视角方向，因此要在v2f接头体中添加两个变量来存储变换后的光照和视角方向
59              struct v2f
60              {
61                  float4 pos : SV_POSITION;
62                  float4 uv : TEXCOORD0;
63                  float3 lightDir : TEXCOORD1;
64                  // 存储纹理坐标的UV值，可在片元着色器中使用该坐标进行纹理采样
65                  float3 viewDir : TEXCOORD2;
66              };
67
68              v2f vert(a2v v)
69              {
70                  v2f o;
71
72                  // 将顶点坐标由模型空间转到裁剪空间
73                  o.pos = UnityObjectToClipPos(v.vertex);
74                  //因为我们使用了两张纹理，所以要存储两个纹理坐标
75                  //我们把uv定义为flaot4，xy存储_MainTex的，zw存储_BumpMap的
76                  //但实际上，二者通常会使用同一组纹理坐标，处于减少插值寄存器使用数量的目的，所以大多数时候，我们只计算和存储一个纹理坐标即可
77                  o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
78                  o.uv.zw = TRANSFORM_TEX(v.texcoord, _BumpMap);
79
80                  // //先计算出副切线
81                  // float3 binormal = cross(normalize(v.normal), normalize(v.tangent.xyz)) * v.tangent.w;
82                  // //再按切线、副切线、法线 按行顺序（右乘） 构建3x3矩阵
```

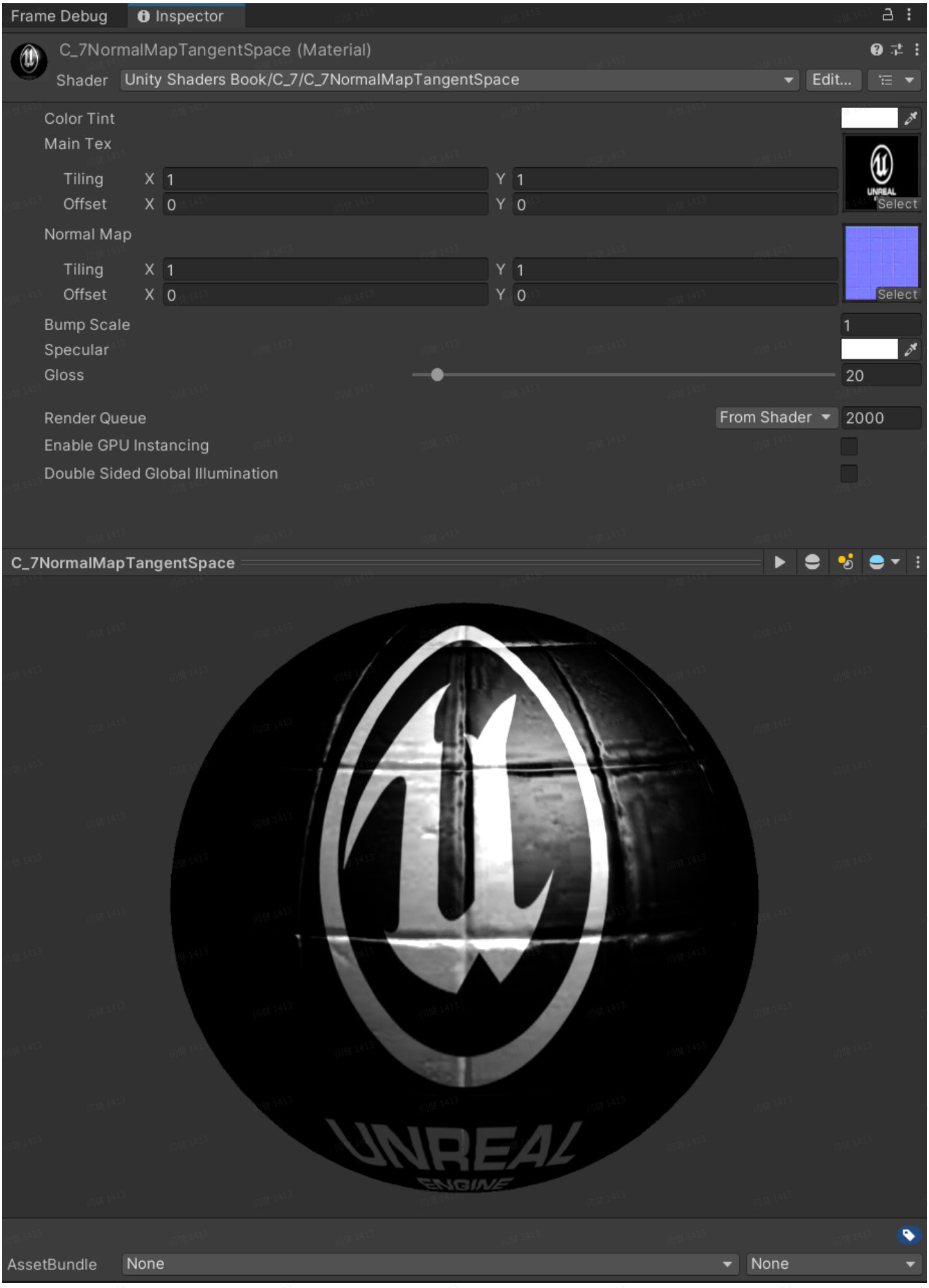
```
83                  // float3x3 rotation = float3x3(v.tangent.xyz, binormal, v.normal);
84                  //上面的代码等同于下面的宏
85                  TANGENT_SPACE_ROTATION;
86                  //把光源方向从模型空间变换到切线空间
87                  o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
88                  //把视线从模型空间变换到切线空间
89                  o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;
90
91                  return o;
92              }
93
94              float4 frag(v2f i) : SV_Target
95              {
96
97                  fixed3 tangentLightDir = normalize(i.lightDir);
98                  fixed3 tangentViewDir = normalize(i.viewDir);
99
100                 //拿到法线纹理中的纹素
101                 fixed4 packedNormal = tex2D(_BumpMap, i.uv.zw);
102                 fixed3 tangentNormal;
103
104                 //如果纹理没有被标记为"Normal Map"，即Texture Type不是法线纹理,则手动反映射得到法线方向
105                 //tangentNormal.xy = (packedNormal.xy * 2 - 1);
106                 //tangentNormal.xy = (packedNormal.xy * 2 - 1) * _BumpScale;
107
108                 //如果纹理已经被标记为"Normal Map"，Unity就会根据不同的平台来选择不同的压缩方法，需要调用UnpackNormal来进行反映射，
109                 // 如果这时再手动计算反映射就会出错，因为_BumpMap的rgb分量不再是切线空间下的法线方向xyz值了
110                 tangentNormal = UnpackNormal(packedNormal);
111                 tangentNormal.xy * _BumpScale;
112                 // 因为最终计算是得到归一化的法线，所以利用三维勾股定理得到z分量
113                 // 因为使用的是切线空间下的法线纹理，所以可以保证z分量为正
114                 tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));
115
116                 fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;
117                 fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;
118
119                 fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));
120
121                 fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
122                 fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss);
```

```
123
124                 return fixed4(ambient + diffuse + specular, 1.0);
125             }
126
127             ENDCG
128         }
129     }
130
131     // 用系统内置的高光shader作为兜底
132     Fallback "Specular"
133 }
```

上面的代码中，我们首先利用tex2D对法线纹理_BumpMap进行采样，法线纹理中存储的是把法线经过映射后得到的像素值，因此我们需要把它们反映射回来。如果我们没有在Unity里把该法线纹理类型设置为Normal map，就需要在代码中手动进行这个过程。

先把packedNormal的xy分量按之前提到的公式映射回法线方向，然后乘以_BumpScale（凹凸程度）来得到tangentNormal的xy分量。因为法线都是单位向量，因此tangentNormal.z分量可以由tangentNormal.xy计算而得。然后由于我们使用的是切线空间下的法线纹理，因此可以保证法线方向的z分量为正。

在Unity中，为了方便Unity对法线纹理的存储进行优化，我们通常会把法线纹理的纹理类型标识成Normal map，Unity会根据平台来选择不同的压缩方法。这时，如果我们再使用上面的方法来计算就会得到错误的结果，因为此时_BumpMap的rgb分量不再是切线空间下法线方向的xyz值了，所以我们需要使用 `UnpackNormal` 来得到正确的法线方向。


Frame Debug
Inspector
C_7NormalMapTangentSpace (Material)
Shader
Unity Shaders Book/C_7/C_7NormalMapTangentSpace
Edit...
Color Tint
Main Tex
Tiling X 1 Y 1
Offset X 0 Y 0
Select
Normal Map
Tiling X 1 Y 1
Offset X 0 Y 0
Select
Bump Scale 1
Specular
Gloss 20
Render Queue From Shader 2000
Enable GPU Instancing
Double Sided Global Illumination
C_7NormalMapTangentSpace
AssetBundle None None

在世界空间下计算

接着我们在世界空间下计算光照模型。我们需要在片元着色器中把法线方向从切线空间变换到世界空间下。其基本思想是：在顶点着色器中计算从切线空间到世界空间的变换矩阵，并把它传递给片元着色器。变换矩阵的计算可以由顶点的切线、副切线和法线在世界空间下的表示来得到。最后，我们只需要在片元着色器中把法线纹理中的法线方向从切线空间变换到世界空间下即可。

这种方法需要更多的计算，但如果需要使用Cubemap进行环境映射，我们就需要使用这种方法。

```
shader "Unity Shaders Book/C_7/C_7NormalMapWorldSpace"
{
    Properties
    {
        // 叠加的颜色，默认为白色
        _Color ("Color Tint", Color) = (1, 1, 1, 1)
        // 纹理，类型为2D，没有纹理时，默认用白色覆盖物体的表面
        _MainTex("Main Tex", 2D) = "white" {}
        //法线纹理，使用bump作为它的默认值，是Unity内置的法线纹理
        _BumpMap("Normal Map", 2D) = "bump" {}
        //用于控制凹凸程度，当它为0是，该法线不会对光照产生任何影响。
        _BumpScale("Bump Scale", Float) = 1.0
        // 高光颜色，默认为白色
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        // 光泽度，影响高光反射区域的大小
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }

    SubShader
    {
        Pass
        {
            // 指明当前Pass的光照模式
            Tags { "LightMode" = "ForwardBase"}

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            // 为了使用光照相关的内置变量（如：_LightColor0 光照颜色）
            #include "Lighting.cginc"

            /* 定义属性变量 */
            fixed4 _Color;
            sampler2D _MainTex;
            // 与_MainTex配套的纹理缩放（scale）和平移（translation），在材质面板的纹理属性中可以调节
            // 命名规范为：纹理变量名 + "_ST"
```

```
39              // _MainTex_ST.xy 存储缩放值
40              // _MainTex_ST.zw 存储偏移值
41              float4 _MainTex_ST;
42              sampler2D _BumpMap;
43              //与_BumpMap配套的纹理缩放平移变量
44              float4 _BumpMap_ST;
45              float _BumpScale;
46              fixed4 _Specular;
47              float _Gloss;
48
49              struct a2v
50              {
51                  float4 vertex : POSITION;
52                  float3 normal : NORMAL;
53                  //切线空间是切线和法线构建出的坐标空间，切线是float4，因为我们需要使用tangent.w来决定切线空间中的第三个坐标轴——副切线的方向性。
54                  float4 tangent : TANGENT;
55                  // 存储模型的第一组纹理坐标，可以理解为_MainTex对应的原始纹理坐标
56                  float4 texcoord : TEXCOORD0;
57              };
58              //要在顶点着色器中计算切线空间下的光照和视角方向，因此要在v2f接头体中添加两个变量来存储变换后的光照和视角方向
59              struct v2f
60              {
61                  float4 pos : SV_POSITION;
62                  float4 uv : TEXCOORD0;
63                  //一个插值寄存器最多只能存储float4大小的变量，对于矩阵这样的变量，我们可以把他们按行拆成多个变量再进行存储。
64                  //TtoW0~TtoW2依次存储了从切线空间到世界空间的变换矩阵的每一行。
65                  //对方向向量的变换只需要使用3x3大小的矩阵，即每一行都使用float3类型的变量即可。
66                  //但为了充分利用插值寄存器的存储空间,我们把世界空间下的顶点坐标存储在这些变量的w分量中。
67                  float4 TtoW0 : TEXCOORD1;
68                  float4 TtoW1 : TEXCOORD2;
69                  float4 TtoW2 : TEXCOORD3;
70              };
71
72              v2f vert(a2v v)
73              {
74                  v2f o;
75                  o.pos = UnityObjectToClipPos(v.vertex);
76
77                  o.uv.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
78                  o.uv.zw = TRANSFORM_TEX(v.texcoord, _BumpMap);
79
80                  float3 worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
```

```
81                  fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
82                  fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
83                  fixed3 worldBinormal = cross(worldNormal, worldTangent) *
    v.tangent.w;
84                  //计算从切线空间到世界空间的矩阵
85                  //然后把世界坐标放进w分量
86                  //我们将切线、副切线、法线按列拜访，就得到了从切线空间到世界空间的变换矩
    阵
87                  o.TtoW0 = float4(worldTangent.x, worldBinormal.x,
    worldNormal.x, worldPos.x);
88                  o.TtoW1 = float4(worldTangent.y, worldBinormal.y,
    worldNormal.y, worldPos.y);
89                  o.TtoW2 = float4(worldTangent.z, worldBinormal.z,
    worldNormal.z, worldPos.z);
90
91                  return o;
92              }
93
94              float4 frag(v2f i) : SV_Target
95              {
96                  //获取世界空间下的坐标
97                  float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
98                  //计算世界空间下的光照和视角方向
99                  fixed3 lightDir = normalize(UnityWorldSpaceLightDir(worldPos));
100                 fixed3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos));
101
102                 //获取切线空间下的法线
103                 fixed3 bump = UnpackNormal(tex2D(_BumpMap, i.uv.zw));
104                 bump.xy *= _BumpScale;
105                 //同样根据三维勾股定理得到z方向的值
106                 bump.z = sqrt(1.0 - saturate(dot(bump.xy, bump.xy)));
107                 //把法线从切线空间转换到世界空间
108                 bump = normalize(half3(dot(i.TtoW0.xyz, bump),
    dot(i.TtoW1.xyz, bump), dot(i.TtoW2.xyz, bump)));
109
110
111                 fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;
112                 fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;
113
114                 fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(bump,
    lightDir));
115
116                 fixed3 halfDir = normalize(lightDir + viewDir);
117                 fixed3 specular = _LightColor0.rgb * _Specular.rgb *
    pow(max(0, dot(bump, halfDir)), _Gloss);
118
119                 return fixed4(ambient + diffuse + specular, 1.0);
```

```
            }

            ENDCG
        }
    }

    // 用系统内置的高光shader作为兜底
    Fallback "Specular"
}
```

上面的代码中，我们计算了世界空间下的顶点切线、副法线和法线的向量表示，并把它们按列拜访得到从切线空间到世界空间的变换矩阵。我们把矩阵的每一行分别存储在TtoW0、TtoW1、TtoW2中，并把世界空间下的顶点坐标xyz分量分别存储在了这些变量的w分量中，以便充分利用插值寄存器的存储空间。

首先从TtoW0、TtoW1和TtoW2的w分量中构建世界空间下的坐标。然后，使用内置的 `UnityWorldSpaceLightDir` 和 `UnityWorldSpaceViewDir` 函数得到世界空间下的光照和视角方向。接着，我们使用内置的 `UnpackNormal` 函数对法线纹理进行采样和解码（需要把法线纹理的格式标识成Normal Map），并使用_BumpScale对其进行缩放。最后，我们使用TtoW0、TtoW1和TtoW2存储的变换矩阵把法线变换到世界空间下。这是通过使用点乘操作来实现矩阵的每一行和法线相乘来得到的。

从视觉表现上，在切线空间下和在世界空间下计算光照几乎没有任何差别，在不需要使用Cubemap进行环境映射的情况下，内置的Unity Shader使用的是切线空间来进行法线映射和光照计算。

Unity中的法线纹理类型

上面我们提到了当把法线纹理的纹理类型标识为Normal Map时，可以使用Unity的你值函数 `UnpackNormal` 来得到正确的法线方向。

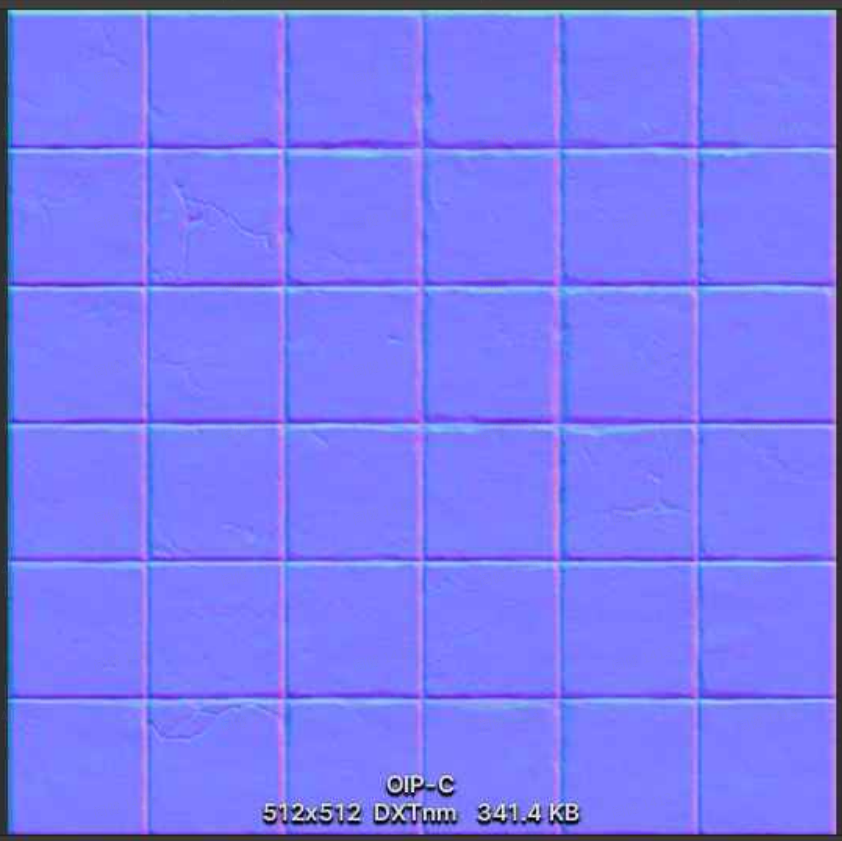
Frame Debug | Inspector

OIP-C (Texture 2D) Import Settings

Open

Texture Type	Normal map
Texture Shape	2D
Create from Grayscale	☐
Flip Green Channel	☐
▼ Advanced	
Non-Power of 2	ToNearest
Read/Write	☐
Virtual Texture Only	☐
Generate Mipmaps	☑
Use Mipmap Limits	☑
Mipmap Limit Group	None (Use Global Mipmap Limit)
Mip Streaming	☐
Mipmap Filtering	Box
Preserve Coverage	☐
Replicate Border	☐
Fadeout to Gray	☐
Swizzle	R G B A
Wrap Mode	Repeat
Filter Mode	Bilinear
Aniso Level	1

OIP-C



AssetBundle None None

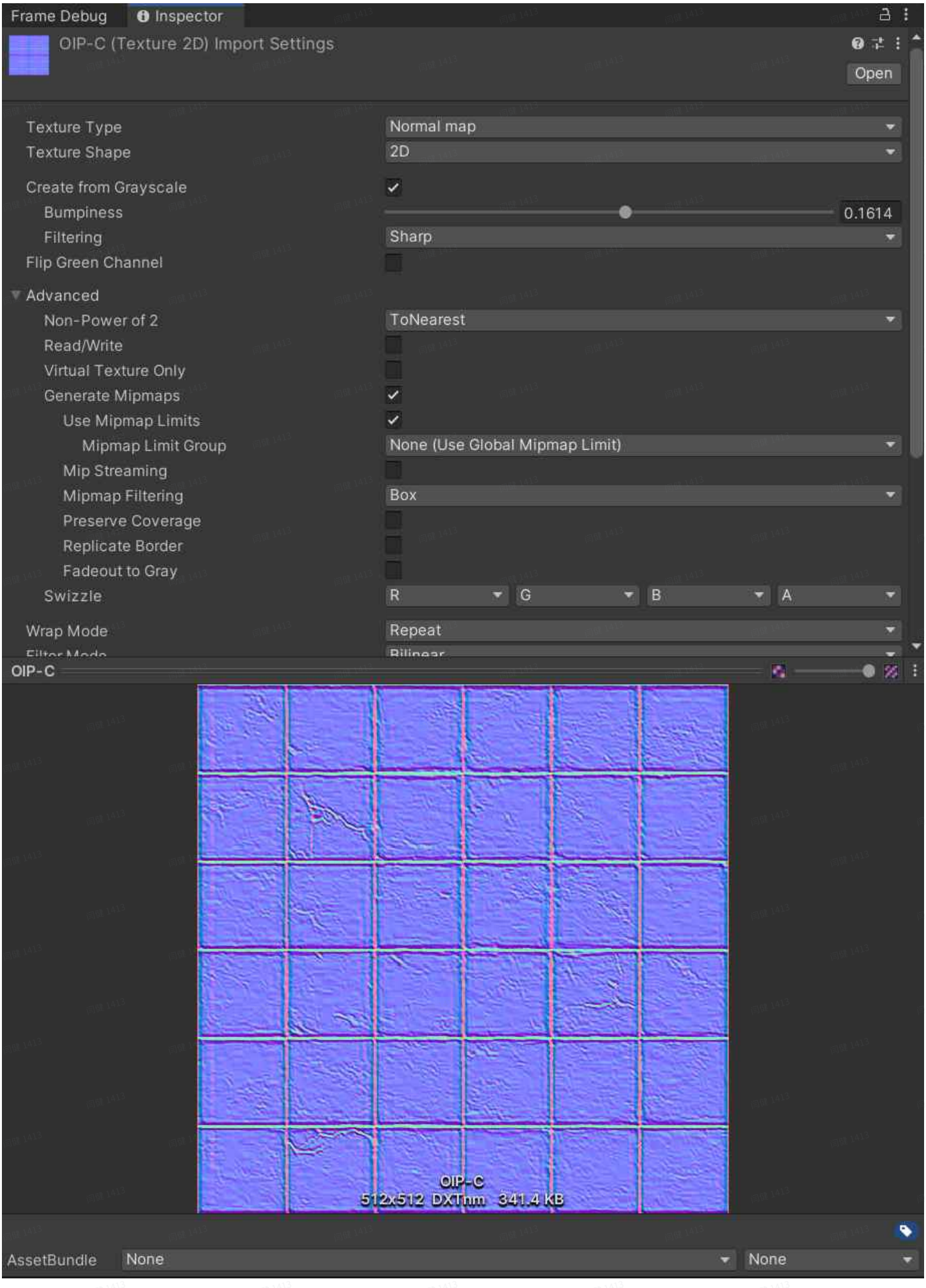
当我们需要使用那些包含了法线映射的内置的Unity Shader时，必须把使用的法线纹理按上面的方式标识成Normal Map才能得到正确结果。因为这些Unity Shader都使用了内置的UnpackNormal函数来采样法线方向。

简单来说，Unity会根据不同平台对纹理进行压缩（例如使用DXT5nm或者BC5格式的压缩格式，具体区别可以看我们在UnityCG.cginc中的内置函数 `UnpackNormal` 解析。）

例如，在某些平台上，使用了DXT5nm的压缩格式，因此需要针对这种格式，对法线进行解码。在DXT5nm格式的法线纹理中，纹素的a通道（w分量）对应了发现的x分量，g通道对应了法线的y分量，而纹理的r和b通道则会被舍弃，法线的z分量可以由xy分量推导而得。

这是因为，按我们之前的处理方式，法线纹理被当成一个和普通纹理无异的图，但实际上，它只有两个通道是真正必不可少的，因为第三个通道的值可以用另外两个推导出来（法线是单位向量，并且切线空间下的法线方向的z分量始终为正）。通过这种压缩方法，就可以减少法线纹理占用的内存空间。

下面的 `Create from Grayscale` 勾选框就是从高度图中生成法线纹理。高度图本身记录的是相对高度，是一张灰度图，白色表示相对更高，灰色表示相对更低。当我们把一张高度图导入Unity后，除了需要把它的纹理类型设置成Normal map外，还需要勾选 `Create from Grayscale` ，就可以得到图示效果：



然后就可以把它和切线空间下的法线纹理同等对待了。

此外，当勾选了`Create from Grayscale`后，还多出了两个选项——`Bumpiness`和`Filtering`：

1. `Bumpiness`用于控制凹凸程度；
2. `Filtering`用于决定我们是用那种方式来计算凹凸程度；
    a. `Smooth`会使得生成后的法线纹理比较平滑；
    b. `Sharp`会使用Sobel滤波（一种边缘检测时使用的滤波器）来生成发现。Sobel滤波实现非常简单，我们只需要在一个3x3的滤波器中计算x和y方向上的倒数，然后从中得到法线即可。
        i. 具体是：对于高度图中的每个像素，我们考虑它与水平方向和竖直方向上的像素差，把它们的差当成该点对应的法线在x和y方向上的位移，然后使用之前提到的映射函数存储成到法线纹理的r和g分量即可。

渐变纹理

纹理可以用于存储任何表面属性，例如使用渐变纹理来控制漫反射光照的结果。这种技术在《军团要塞2》中流行起来，它也是由Value公司提出来的，他们使用这种技术来渲染游戏中具有插画风格的角色。

这种技术最早有Gooch等人在1998年发表的的论文中，提出了一种冷到暖色调的着色技术，来得到插画风格的渲染效果，现在很多卡通风格的Shader都使用了这种技术。

```
shader "Unity Shaders Book/C_7/C_7RampTexture"
{
    Properties
    {
        _Color("Color Tint", Color) = (1, 1, 1, 1)
        _RampTex("Ramp Tex", 2D) = "white" {}
        _Specular("Specular", Color) = (1, 1, 1, 1)
        _Gloss("Gloss", Range(8.0, 256)) = 20
    }

    SubShader
    {
        Pass
        {
            // 指明当前Pass的光照模式
            Tags { "LightMode" = "ForwardBase"}

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

```

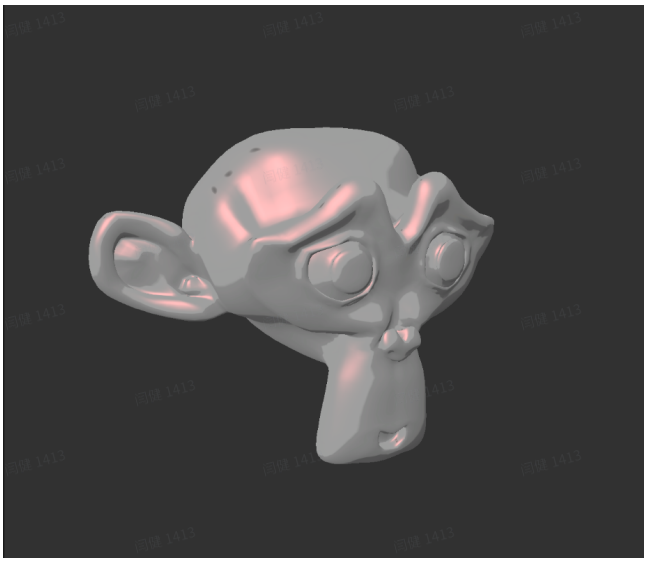
```
23              // 为了使用光照相关的内置变量（如：_LightColor0 光照颜色）
24              #include "Lighting.cginc"
25
26              fixed4 _Color;
27              sampler2D _RampTex;
28              float4 _RampTex_ST;
29              fixed4 _Specular;
30              float _Gloss;
31
32              struct a2v
33              {
34                  float4 vertex : POSITION;
35                  float3 normal : NORMAL;
36                  float4 texcoord : TEXCOORD0;
37              };
38              struct v2f
39              {
40                  float4 pos : SV_POSITION;
41                  float3 worldNormal : TEXCOORD0;
42                  float3 worldPos : TEXCOORD1;
43                  float2 uv : TEXCOORD2;
44              };
45
46              v2f vert(a2v v)
47              {
48                  v2f o;
49
50                  // 将顶点坐标由模型空间转到裁剪空间
51                  o.pos = UnityObjectToClipPos(v.vertex);
52                  o.worldNormal = UnityObjectToWorldNormal(v.normal);
53                  o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
54                  o.uv = TRANSFORM_TEX(v.texcoord, _RampTex);
55
56                  return o;
57              }
58
59              float4 frag(v2f i) : SV_Target
60              {
61                  float3 worldNormal = normalize(i.worldNormal);
62                  float3 worldLightDir =
    normalize(UnityWorldSpaceLightDir(i.worldPos));
63                  //根据半兰伯特光照模型，通过法线方向和光照方向的点积
64                  //做一次0.5倍的缩放以及一个0.5大小的偏移
65                  //来计算半兰伯特部分halfLambert
66                  //得到的值范围被映射到[0,1]之间
67                  fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;
68                  //根据halfLambert构建一个纹理坐标
```

```
69                  //并使用这个纹理坐标对渐变纹理_RampTex进行采样
70                  //由于_RampTex实际是一个一维纹理(纵轴方向颜色不变)
71                  //因此纹理坐标的u和v方向都是halfLambert
72                  //采样得到的颜色和材质颜色_Color相乘，得到最终的漫反射颜色
73                  fixed halfLambert = 0.5 * dot(worldNormal, worldLightDir) +
    0.5;
74                  fixed3 diffuseColor = tex2D(_RampTex, fixed2(halfLambert,
    halfLambert)).rgb * _Color.rgb;
75
76                  fixed3 diffuse = _LightColor0.rgb * diffuseColor;
77                  //应用Blinn-Phong光照模型计算高光反射
78                  fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
79                  fixed3 halfDir = normalize(worldLightDir + viewDir);
80                  fixed3 specular = _LightColor0.rgb * _Specular.rgb *
    pow(max(0, dot(worldNormal, halfDir)), _Gloss);
81                  return fixed4(ambient + diffuse + specular, 1.0);
82              }
83
84              ENDCG
85          }
86      }
87
88      // 用系统内置的高光shader作为兜底
89      Fallback "Specular"
90  }
```

使用三种不同的渐变纹理的模型效果如图：



需要注意的是，我们需要把渐变纹理的 `Wrap Mode` 设置为Clamp模式，以防止对纹理进行采样时，由于浮点数精度造成的问题。




前者是Repeat模式，后者是Clamp模式，可以看出还是有明显区别的，左图的高光区域有一些黑点，这是由浮点数精度造成的。当我们使用fixed2(halfLambert, halfLambert)对渐变纹理进行采样时，虽然理论上halfLambert的值在[0,1]之间，但可能会有1.000000001这样的值出现；如果是Repeat模式，此时就会舍弃整数部分，只保留小数部分，由于这个值很接近于0，所以就会是黑色。反之，如果是Clamp模式，会把浮点数夹到1的范围内，就可以解决这种问题。

遮罩纹理

遮罩纹理(mask texture)允许我们保护某些区域，使他们免于某些修改。例如之前的高光反射是应用到模型表面的全部区域，所有的像素都使用同样大小的高光强度和指数。如果希望模型表面某些区域的反光强烈一些，某些区域弱一些，为了得到更细腻的效果，我们就可以使用一张纹理遮罩来控制光照。

另一种常见的应用是在制作地形材质时，需要混合多长图片，例如表现草地的纹理、表现石子的纹理、表现裸露土地的纹理等，使用遮罩纹理可以控制如何混合这些纹理。

使用遮罩纹理的流程一般是：通过采样得到遮罩纹理的纹素值，然后使用其中某个（或某几个）通道的值（例如texel.r）来与某种表面属性进行相乘，这样，当通道的值为0时，可以保护表面不受该属性的影响。总之，使用遮罩纹理可以让美术人员像素级别地控制表面的各种性质。

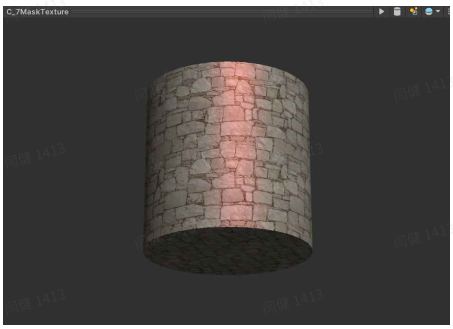
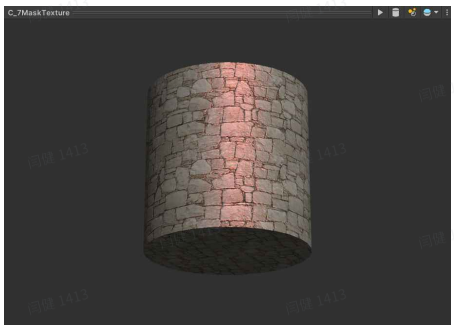
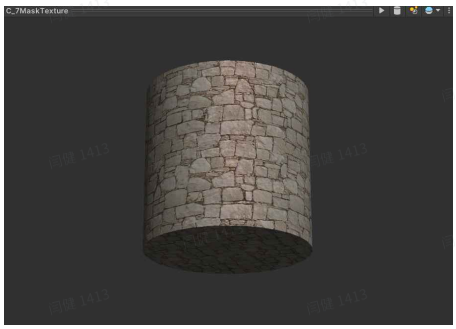
实践

```
shader "Unity Shaders Book/C_7/C_7MaskTexture"
{
    Properties
    {
        _Color ("Color Tint", Color) = (1, 1, 1, 1)
        // 主纹理
        _MainTex ("Main Tex", 2D) = "white" {}
        // 法线纹理
        _BumpMap ("Normal Map", 2D) = "bump" {}
```

```
        // 控制法线纹理影响程度的系数
        _BumpScale ("Bump Scale", Float) = 1.0
        // 高光反射遮罩纹理
        _SpecularMask ("Specular Mask", 2D) = "white" {}
        // 控制遮罩影响程度的系数
        _SpecularScale ("Specular Scale", Float) = 1.0
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }

    SubShader
    {
        Pass
        {
            Tags { "LightMode" = "ForwardBase" }

            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "Lighting.cginc"

            fixed4 _Color;
            sampler2D _MainTex;
            // _MainTex、_BumpMap和_SpecularMask共用同一套纹理属性变量_MainTex_ST
            // 意味着修改主纹理的平铺系数和偏移系数，会同时影响3个纹理的采样
            // 这样可以节省存储的纹理坐标数，减少差值寄存器的使用
            float4 _MainTex_ST;
            sampler2D _BumpMap;
            float _BumpScale;
            sampler2D _SpecularMask;
            float _SpecularScale;
            fixed4 _Specular;
            float _Gloss;

            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 tangent : TANGENT;
                float4 texcoord : TEXCOORD0;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
```

```
57                 float2 uv : TEXCOORD0;
58                 float3 lightDir : TEXCOORD1;
59                 float3 viewDir : TEXCOORD2;
60             };
61 
62             v2f vert (a2v v)
63             {
64                 v2f o;
65                 o.pos = UnityObjectToClipPos(v.vertex);
66 
67                 o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
68 
69                 // 使用宏，让后续可以调用rotation变量获取模型空间到
70                 // 切线空间的转换矩阵
71                 TANGENT_SPACE_ROTATION;
72                 // 获得切线空间的光照方向
73                 o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
74                 // 获得切线空间的视角方向
75                 o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;
76 
77                 return o;
78             }
79 
80             fixed4 frag(v2f i) : SV_Target
81             {
82                 fixed3 tangentLightDir = normalize(i.lightDir);
83                 fixed3 tangentViewDir = normalize(i.viewDir);
84 
85                 // 从法线纹理中采样获得切线空间的法线
86                 fixed3 tangentNormal = UnpackNormal(tex2D(_BumpMap, i.uv));
87                 // 乘以影响程度系数_BumpScale
88                 tangentNormal.xy *= _BumpScale;
89                 // 因为最终法线需要归一化，所以由勾股定理求出z分量
90                 tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy,
    tangentNormal.xy)));
91 
92                 fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Color.rgb;
93 
94                 fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;
95 
96                 fixed3 diffuse = _LightColor0.rgb * albedo * max(0,
    dot(tangentNormal, tangentLightDir));
97 
98                 fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
99                 // 对遮罩纹理进行采样，因为本案例中遮罩纹理的每个纹素的rgb分量
100                 // 都是一样的，表明了该点对应高光反射强度，
101                 // 所以仅适用r分量来计算掩码值
```

```
                fixed specularMask = tex2D(_SpecularMask, i.uv).r * _SpecularScale;
                // 掩码值与最终求得的高光反射相乘，用以控制高光反射的强度
                fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss) * specularMask;

                return fixed4(ambient + diffuse + specular, 1.0);
            }

            ENDCG
        }
    }

    Fallback "Specular"
}
```

上面代码中，为主纹理_MainTex、法线纹理_BumpMap和遮罩纹理_SpecularMask定义了它们共同使用的纹理属性变量_MainTex_ST，这意味着在材质面板中修改主纹理的平铺系数和偏移系数会同时影响3个纹理的采样。使用这种方式可以让我们节省需要存储的纹理坐标，如果我们为每一个纹理都使用一个单独的属性变量Texture_ST，那么随着使用的纹理数量增加，我们会迅速占满顶点着色器中可以使用的插值寄存器。

在计算高光反射时，我们首先对遮罩纹理_SpecularMask进行采样。由于本书使用的遮罩纹理中每个纹素的rgb分量其实都是一样的，表明了该点对应的高光反射强度，在这里我们选择使用r分量来计算。最终，用得到的值和_SpecularScale相乘，一起来控制高光反射的强度。

需要说明，这张遮罩纹理其实有很多空间被浪费了——它的rgb分量存储的都是同一个值。实际制作中，可以充分利用遮罩纹理的每一个颜色通道来存储不同的表面属性。