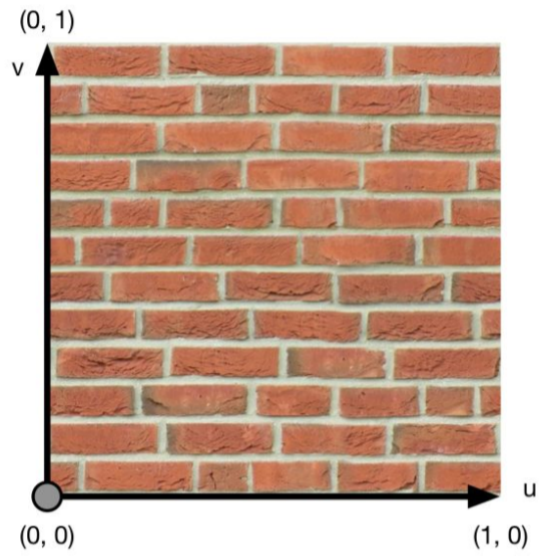
Lecture7 基础纹理

- 使用**纹理映射 texture mapping** 技术，可以把一张图黏在模型表面，逐**纹素 texel** 控制模型颜色
- 美术人员建模时，通常会在建模软件中把**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再纹理中对应的 2D 坐标，通常这个坐标使用**二维变量 (u, v)** 表示，u 为横向坐标，v 为纵向坐标，因此，纹理映射坐标也被称为 **UV 坐标**
- 注意，OpenGL 和 DirectX 在二维纹理空间中的**坐标存在差异**
  - OpenGL：原点在左下角
  - DirectX：原点在左上角

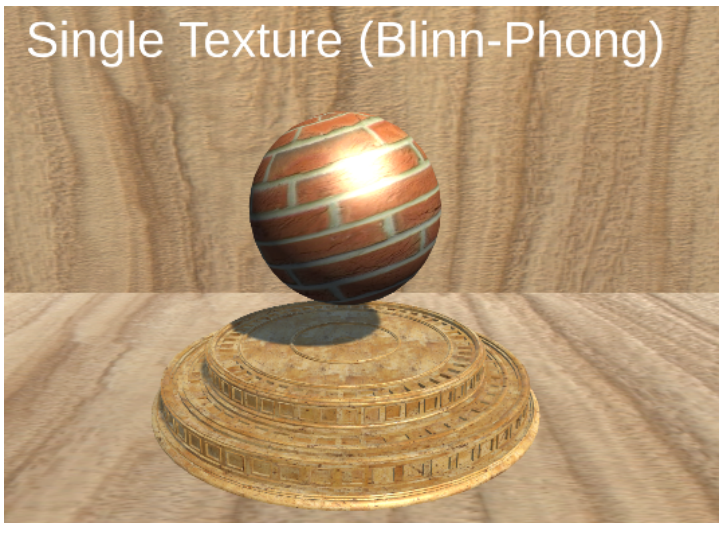


- 本章着重讲述纹理采样的原理， Shader 并不能实际应用在项目中（直接使用会缺少阴影、光照衰减等）

1. 单张纹理

单张纹理 Shader

我们使用 Blinn-Phong 模型计算光照，加入单张纹理的计算



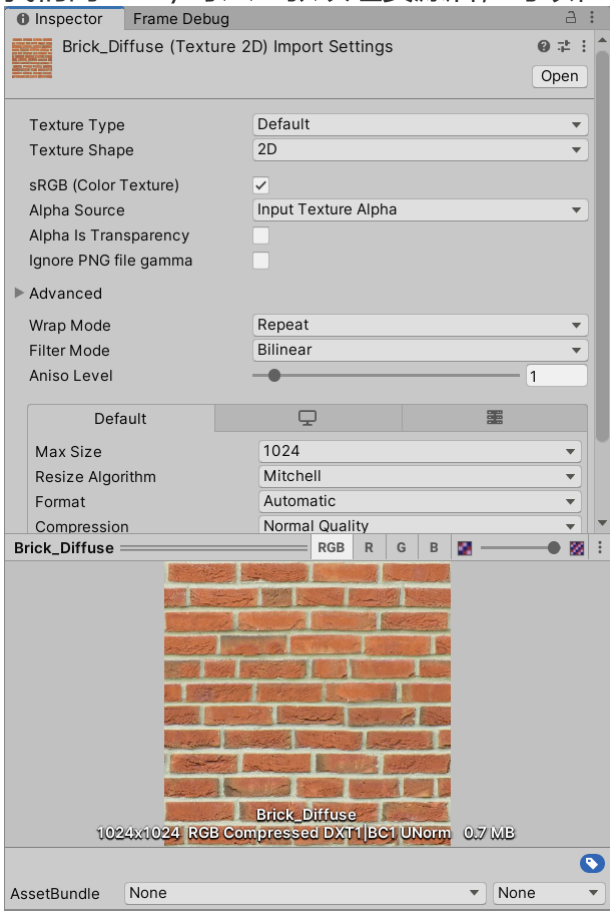

```
Shader "Custom/Assignment/Chapter7/1_SingleTexture"
{
    Properties
    {
        _Diffuse ("Color Tint", Color) = (1, 1, 1, 1)
        _MainTex ("Main Tex", 2D) = "white" {}
        _Specular ("Specular", Color) = (1, 1, 1, 1)
        _Gloss ("Gloss", Range(8.0, 256)) = 20
    }
    SubShader
    {
        Pass
        {
            Tags
            {
                "LightMode"="ForwardBase"
            }

            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
```

```
#include "Lighting.cginc"

fixed4 _Diffuse;
sampler2D _MainTex;
// 在Unity中，我们需要使用纹理名_ST的方式来声明某个纹理的属性
// ST是缩放和平移的缩写
// _MainTex_ST.xy：缩放，_MainTex_ST.zw：偏移值
float4 _MainTex_ST;
fixed4 _Specular;
float _Gloss;

struct a2v
{
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float4 texcoord : TEXCOORD0; // 存储第一组纹理坐标
};

struct v2f
{
    float4 pos : SV_POSITION;
    float3 worldNormal : TEXCOORD0;
    float3 worldPos : TEXCOORD1;
    float2 uv : TEXCOORD2; // 存储纹理坐标的变量，片元着色器使用其进行纹理采样
};

v2f vert(a2v v)
{
    v2f o;
    o.pos = UnityObjectToClipPos(v.vertex);
    o.worldNormal = UnityObjectToWorldNormal(v.normal);
    o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;

    // 使用纹理属性_MainTex_ST对纹理坐标进行变换，得到最终的纹理坐标，使用缩放属性xy进行缩放，再使用偏移属性zw对结果进行偏移
    o.uv = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
    // Unity内提供一个内置宏TRANSFORM_TEX来帮我们计算上述过程
    // 在UnityCG.cginc中定义，它接受两个参数，第一个参数是顶点纹理坐标，第二个参数是纹理名
    //          o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);

    return o;
}

fixed4 frag(v2f i) : SV_Target
{
    fixed3 worldNormal = normalize(i.worldNormal);
    fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));

    // 使用CG的tex2D函数对纹理进行采样
    // 第一个参数是被采样的纹理，第二个参数是一个float2类型的纹理坐标
    // 计算返回纹素值
    // 采样结果x漫反射颜色属性作为材质的反射率albedo
    fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Diffuse.rgb;

    // 计算环境光
    fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

    // 计算漫反射光照结果
    fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(worldNormal, worldLightDir));
```

```
                // 计算高光反射
                fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
                fixed3 halfDir = normalize(worldLightDir + viewDir);
                fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

                // 返回计算结果
                return fixed4(ambient + diffuse + specular, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Specular"
}
```

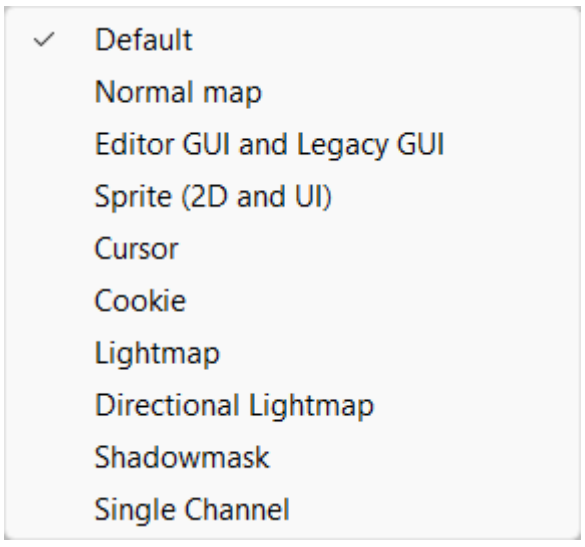
纹理的属性

- 在渲染流水线中，纹理映射的实现很复杂
- 我们向 Unity 导入一张纹理资源后，可以在它的材质面板上调整属性



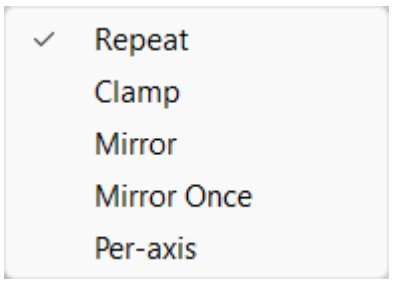
Texture Type

我们为导入的纹理选择合适的类型，以让 Unity 知道我们的意图，为 Unity Shader 传递正确的纹理，并在一些情况下让 Unity 对纹理进行优化

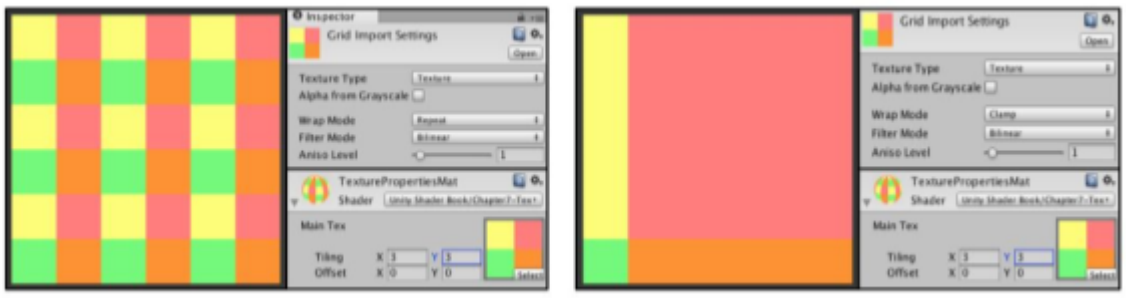


Wrap Mode

决定了当纹理坐标超过了 [0, 1] 范围后将会被如何平铺



- Repeat：当纹理坐标超过了 1，它的整数部分将会被舍弃，直接使用小数部分采样，纹理不断重复
- Clamp：如果纹理坐标大于 1，则会截取到 1，如果小于 0，就会截取到 0

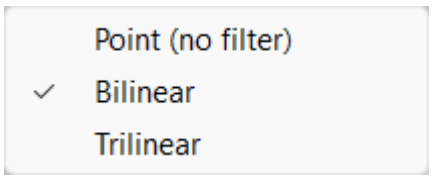


想要纹理得到这样的效果，必须要在 Unity Shader 中对纹理进行相应的变换

```
o.uv = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
// Or
o.uv = TRANSFORM_TEX(v.texcoord, _MainTex);
```

Filter Mode

决定了当纹理由于变换而产生拉伸的时候将采用哪种滤波模式



- 它们得到图片的滤波效果依次提升，但是耗费的性能也依次增大

Filter Mode: Point　Filter Mode: Bilinear　Filter Mode: Trilinear

多级渐远纹理 Mipmapping

- 纹理缩小的过程常用多级渐远纹理技术，它提前使用滤波处理来得到很多更小的图像，形成一个图像金字塔，每一层都是对上一层降采样的效果
- 这样实时运行时，当物体远离摄像机时，可以选择较小的纹理
- 缺点是通常会多占用 33% 的内存空间，这是一个空间换时间的方法
- 在新版 Unity 中，default 模式下会自动生成 mipmap

纹理尺寸和模式

- 通常，导入的纹理的长宽大小最好为 **2 的幂**，否则纹理往往会占用更大的空间，GPU 读取该纹理的速度也会下降
- Format 决定了Unity 内部使用哪种格式来存储纹理

- **纹理格式精度越高，占用内存空间越大，但得到的效果越好**，对于一些不需要使用很高精度的纹理，我们应该尽量采用压缩模式

2. 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，为模型提供更多细节，这种方法**不是真的改变模型的顶点位置**，而是让**模型看起来好像是凹凸不平**

有两种方法来进行凹凸映射

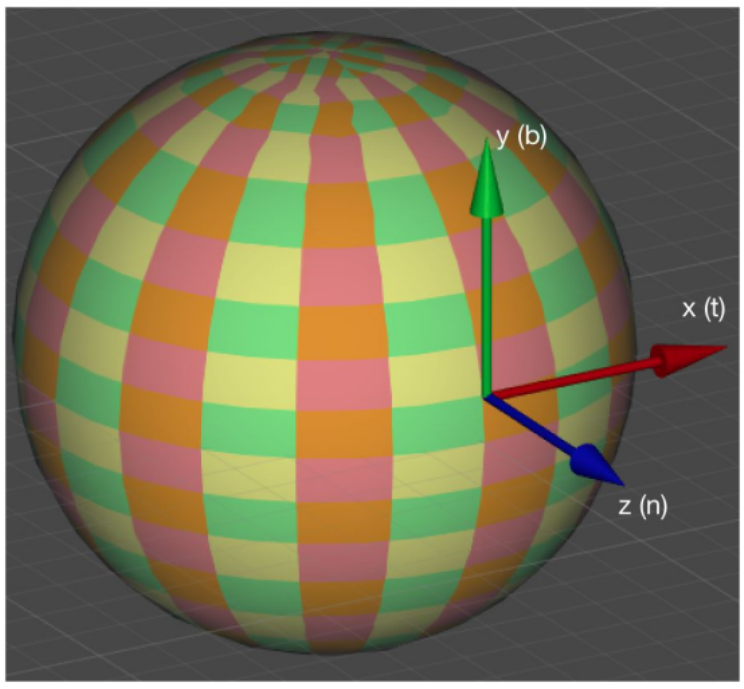
高度纹理 height map

- 模拟表面位移，来得到一个被修改后的法线值
- 使用一张高度图来实现凹凸映射，高度图中存储的是**强度值 intensity**，表示模型局部表面的**海拔高度**
- 颜色越浅表明该位置越向外凸起，越深表明该位置越向里凹

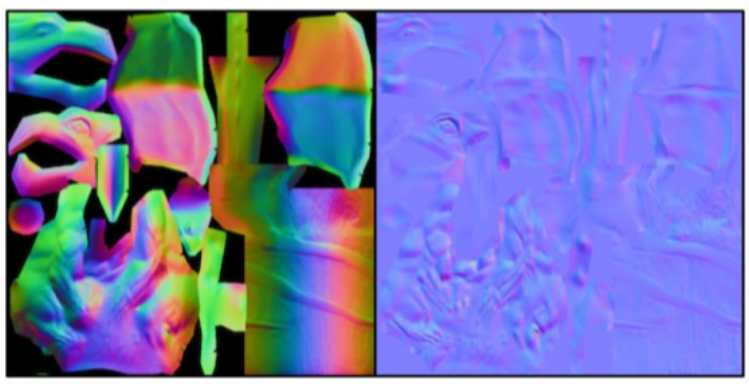
法线纹理 normal map

- 直接存储表面法线，这种方法也被称为**法线映射 normal mapping**，由于法线方向的分量范围在 [-1, 1]，而像素分量分为在 [0, 1]，因此我们做一个映射

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



- 在实际制作中，我们通常采用模型的**切线空间 tangent space** 来存储法线，对于模型的每个顶点，都有属于自己的切线空间，这个空间的原点就是该定点本身
  - z 轴：法线方向
  - x 轴：切线方向
  - y 轴：法线叉积得到，也被称为副切线 bitangent



- 左：模型空间下法线纹理
  - 颜色各异，因为法线方向各异
- 右：切线空间下法线纹理
  - 颜色多为淡蓝色，这种法线纹理其实存储了每个点在格子切线空间下的扰动情况
  - 如果一个点的法线方向不变，那么它的切线空间中，新的法线方向就是 z 轴方向 (0, 0, 1)，经过映射后存储在了纹理对应的 RGB (0.5, 0.5, 1) 浅蓝色上，这个颜色就是法线纹理中大部分蓝色，表明大部分顶点的和模型本身的法线一样，不需要改变

切线空间下法线纹理的优点

- **自由度高**：模型空间下法线纹理运用到其他模型上会崩溃，切线空间的法线纹理仅记载相对信息，即使将该纹理运用到了不同的网格上，也可以得到相对合理的结果
- **可进行 UV 动画**：可以移动纹理的 UV 坐标来实现凹凸移动的效果，这种动画在水或者火山熔岩这类物体上会经常用到
- **可重用**：我们可以用一张法线纹理运用到所有的 6 个面上
- **可压缩**：法线纹理中的法线 Z 方向总是正方向，可以仅存储 XY 方向，而推导得到 Z 方向

Shader 实践

在切线空间下计算 Shader


```
Shader "Custom/Assignment/Chapter7/2_NormalMapTangent"
{
    Properties
    {
        _Diffuse ("Color Tint", Color) = (1, 1, 1, 1)
        _MainTex ("Main Tex", 2D) = "white" {}
        _BumpMap ("Normal Map", 2D) = "bump" {} // 法线纹理，"bump"是Unity内置的法线纹理，当没有提供任何法线纹理时，"bump"就对应了模型自带的法线信息
        _BumpScale ("Bump Scale", Float) = 1.0 // 控制凹凸程度，当它为0时，就意味着法线纹理不会对光照产生任何影响
        _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" // 使用Unity内的一些内置变量

            fixed4 _Diffuse;
            sampler2D _MainTex;
            float4 _MainTex_ST;
            sampler2D _BumpMap; // 凹凸贴图纹理
            float4 _BumpMap_ST; // 凹凸贴图纹理的属性
            float _BumpScale;
            fixed4 _Specular;
            float _Gloss;

            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 tangent : TANGENT; // Unity把顶点的切线方向填充到tangent变量中，需要注意的是，与normal不同，tangent的类型是float4而非float3，这是因为我们需要用tanget.w分量来决定切线空间中的第三个坐标轴，副切线的方向性
                float4 texcoord : TEXCOORD0;
            };

            struct v2f
```

```
            {
                float4 pos : SV_POSITION;
                float4 uv : TEXCOORD0;
                float3 lightDir : TEXCOORD1; // 切线空间的光线方向
                float3 viewDir : TEXCOORD2; // 切线空间的视角方向
            };

            v2f vert(a2v v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                // MainTex和BumpMap使用同一组纹理坐标，是为了减少插值寄存器的使用数目，我们往往只计算和存储一个纹理坐标即可
                o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw; // xy存储_MainTex的纹理坐标
                o.uv.zw = v.texcoord.xy * _BumpMap_ST.xy + _BumpMap_ST.zw; // zw存储_BumpMap的纹理坐标

                // 内置宏，计算得到rotation变换矩阵，自带rotation变量
                TANGENT_SPACE_ROTATION;

                // 光线方向和视角方向的空间变换
                o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
                o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;
                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                fixed3 tangentLightDir = normalize(i.lightDir);
                fixed3 tangentViewDir = normalize(i.viewDir);

                fixed4 packedNoamal = tex2D(_BumpMap, i.uv.zw); // 使用tex2D对法线纹理进行采样
                fixed3 tangentNormal = UnpackNormal(packedNoamal); // 法线纹理存储到是把法线经过映射后得到的像素值，因此需要反映射回来，使用Unity当中的内置函数UnpackNormal来得到正确的法线方向
                tangentNormal.xy *= _BumpScale;
                tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy, tangentNormal.xy)));

                fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Diffuse.rgb;

                fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

                fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal, tangentLightDir));

                fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);
                fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(tangentNormal, halfDir)), _Gloss);

                return fixed4(ambient + diffuse + specular, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Specular"
}
```

在世界空间下计算 Shader

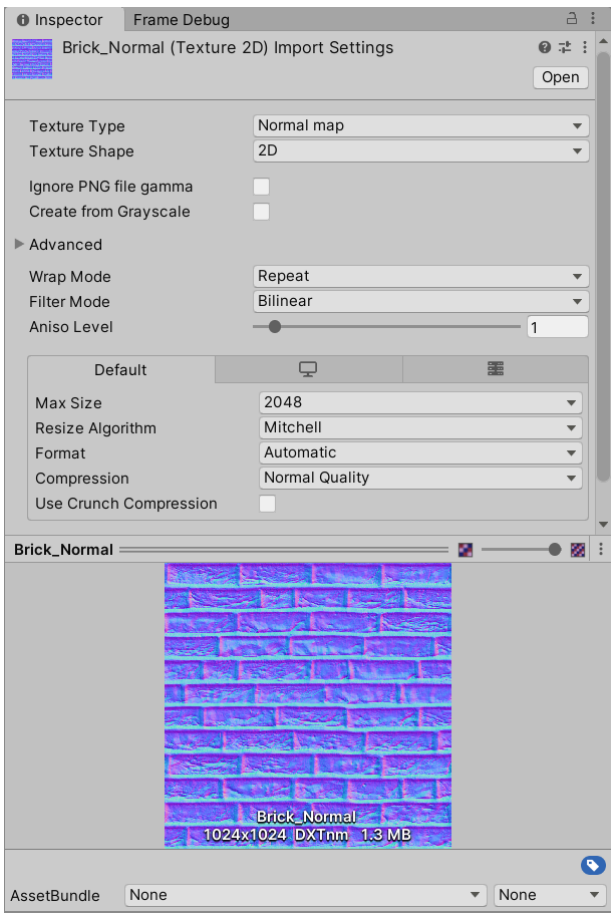
- 我们需要在片元着色器下把法线方向从切线空间变换到世界空间下
- 在顶点着色器中计算切线空间到世界空间的变换矩阵，并把它传递给片元着色器
- 变换矩阵的计算可以由顶点的切线、副切线和法线在世界空间下表示来得到
- 最后我们只需要在片元着色器下把法线纹理中的法线方向从切线空间辩护拿到世界空间下即可
- 这种方法需要更多的计算，但是在需要使用 Cubemap 进行环境映射等情况下，需要使用这种方法

```
Shader "Custom/Assignment/Chapter7/3_NormalMapWorld"
{
    Properties
    {
        _Diffuse ("Color Tint", Color) = (1, 1, 1, 1)
        _MainTex ("Main Tex", 2D) = "white" {}
        _BumpMap ("Normal Map", 2D) = "bump" {}
        _BumpScale ("Bump 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 _Diffuse;
            sampler2D _MainTex;
            float4 _MainTex_ST;
            sampler2D _BumpMap;
            float4 _BumpMap_ST;
            float _BumpScale;
            fixed4 _Specular;
            float _Gloss;

            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 tangent : TANGENT; // this is a float4 variable
                float4 texcoord : TEXCOORD0;
            };

            // 在顶点着色器中计算从切线空间到世界空间的变换矩阵
            // 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，我们可以把它按行拆成多个变量进行存储，所以TtoW0, TtoW1, 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
            // 方向向量的变换实际上只需要3x3的矩阵，为了充分利用插值寄存器的空间，我们把世界空间下的顶点位置存储到w分量中
            struct v2f
            {
                float4 pos : SV_POSITION;
                float4 uv : TEXCOORD0;
                float4 TtoW0 : TEXCOORD1;
                float4 TtoW1 : TEXCOORD2;
```

```
                float4 TtoW2 : TEXCOORD3;
            };

            v2f vert(a2v v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
                o.uv.zw = v.texcoord.xy * _BumpMap_ST.xy + _BumpMap_ST.zw;

                float3 worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;
                fixed3 worldNormal = UnityObjectToWorldNormal(v.normal);
                fixed3 worldTangent = UnityObjectToWorldDir(v.tangent.xyz);
                fixed3 worldBinormal = cross(worldNormal, worldTangent) * v.tangent.w;

                // Compute the matrix that transform directions from tangent space to
world space
                // put the world position in w component for optimization
                o.TtoW0 = float4(worldTangent.x, worldBinormal.x, worldNormal.x,
worldPos.x);
                o.TtoW1 = float4(worldTangent.y, worldBinormal.y, worldNormal.y,
worldPos.y);
                o.TtoW2 = float4(worldTangent.z, worldBinormal.z, worldNormal.z,
worldPos.z);
                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                float3 worldPos = float3(i.TtoW0.w, i.TtoW1.w, i.TtoW2.w);
                fixed3 lightDir = normalize(UnityWorldSpaceLightDir(worldPos));
                fixed3 viewDir = normalize(UnityWorldSpaceViewDir(worldPos));

                fixed3 bump = UnpackNormal(tex2D(_BumpMap, i.uv.zw));
                bump.xy *= _BumpScale;
                bump.z = sqrt(1.0 - saturate(dot(bump.xy, bump.xy)));
                bump = normalize(
                    half3(
                        dot(i.TtoW0.xyz, bump), dot(i.TtoW1.xyz, bump), dot(i.TtoW2.xyz,
bump)
                    ));

                // Use the texture to sample the diffuse color
                fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Diffuse.rgb;

                fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

                fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(bump, lightDir));

                fixed3 halfDir = normalize(lightDir + viewDir);
                fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0, dot(bump,
halfDir)), _Gloss);

                return fixed4(ambient + diffuse + specular, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Specular"
}
```

Unity 中的法线纹理类型

当把法线纹理的 Texture Type 标识成 Normal Map 的时候，可以使用 Unity 的内置函数 UnpackNormal 来得到正确的法线方向，当我们使用那些包含法线映射的内置 Unity Shader 时，必须标识



UnityCG.cginc 中 UmpackNormal 的实现

```
inline fixed3 UnpackNormalDXT5nm (fixed4 packednormal)
{
    fixed3 normal;
    normal.xy = packednormal.wy * 2 - 1;
    normal.z = sqrt(1 - saturate(dot(normal.xy, normal.xy)));
    return normal;
}

// Unpack normal as DXT5nm (1, y, 1, x) or BC5 (x, y, 0, 1)
// Note neutral texture like "bump" is (0, 0, 1, 1) to work with both plain RGB normal and DXT5nm/BC5
fixed3 UnpackNormalmapRGorAG(fixed4 packednormal)
{
    // This do the trick
   packednormal.x *= packednormal.w;

    fixed3 normal;
    normal.xy = packednormal.xy * 2 - 1;
    normal.z = sqrt(1 - saturate(dot(normal.xy, normal.xy)));
    return normal;
}
```

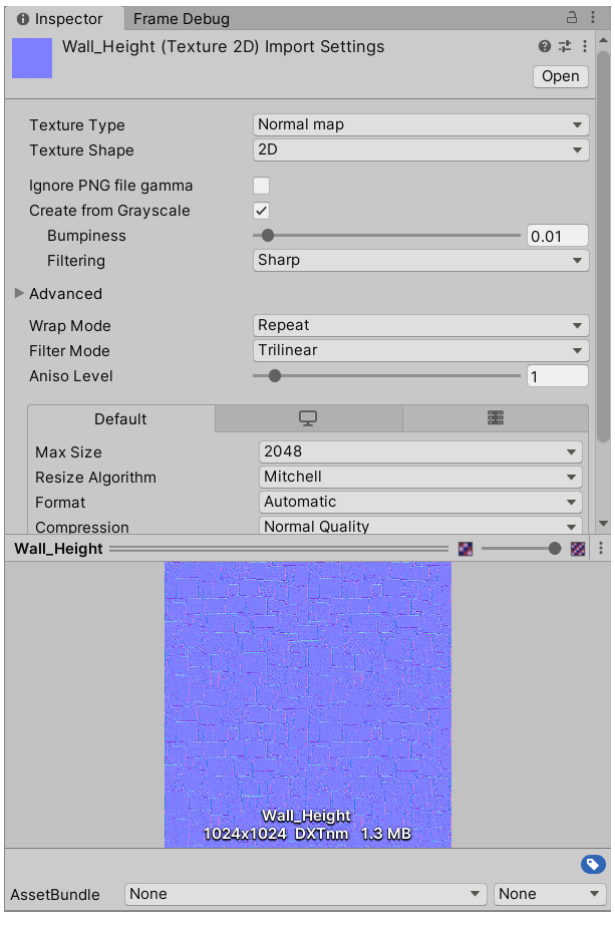
DXT5nm 格式的法线纹理中，纹素的 a 分量对应法线的 x 分量，g 通道对应法线的 y 分量，而纹理的 r 和 b 通道会被舍弃，法线的 z 分量可以由 xy 分量推导获得

事实上，法线的第三个通道的值可以由另外两个推导出来（切线空间下 z 分量始终为正），使用这种压缩方式可以减少法线的内存空间

Create From Grayscale

Inspector 面板中还有一个 Create from Grayscale 的复选框，它用于从**高度图**中生成法线纹理

当我们把一张高度图导入 Unity 后，除了需要设置 Normal map 类型外，还需要勾选这个



勾选后，会多出两个选项

- Bumpiness：凹凸程度
- Filtering：使用哪种方式来计算凹凸程度
  - Smooth：生成的法线纹理较平滑
  - Sharp：使用 Sobel 滤波来生成法线

3. 渐变纹理 Ramp Texture

> A Non-Photorealistic Lighting Model For Automatic Technical Illustration

- 一开始我们使用纹理是为了定义一个物体的颜色，后来人们发现，**纹理其实可以存储任何表面属性**
- 一种常见的用法是使用渐变纹理来控制**漫反射光照的结果**
- 之前我们是使用表面法线与光照方向的点积与材质的反射率相乘来得到漫反射光照，但我们可以更灵活的控制效果
- 很多卡通渲染风格都使用了这种技术

- 图1：从紫色到浅黄色调的渐变纹理
- 图2：与《军团要塞2》中使用类似的渐变纹理，从黑色向浅灰色靠拢，中部微微发红
- 图3：卡通渲染风格，纹理色调突变，没有平滑过渡，来模仿卡通中的阴影色块

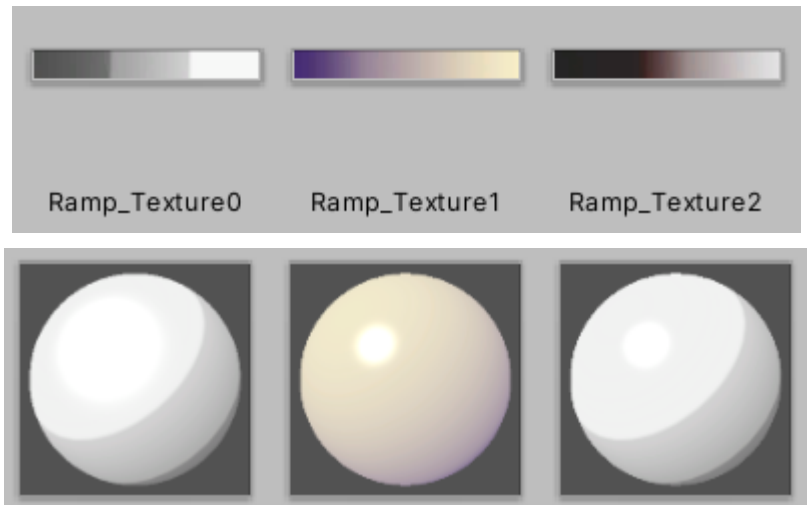
Shader 实践

```
Shader "Custom/Assignment/Chapter7/4_RampTexture"
{
    Properties
    {
        _Color ("Color Tint", Color) = (1, 1, 1, 1)
        _RampTex ("Ramp Tex", 2D) = "white" {} // 渐变纹理
        _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 _RampTex; // 渐变纹理
        float4 _RampTex_ST;
        fixed4 _Specular;
        float _Gloss;

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

        struct v2f
        {
            float4 pos : SV_POSITION;
            float3 worldNormal : TEXCOORD0;
            float3 worldPos : TEXCOORD1;
            float2 uv : TEXCOORD2;
        };

        v2f vert(a2v v)
        {
            v2f o;
            o.pos = UnityObjectToClipPos(v.vertex);

            o.worldNormal = UnityObjectToWorldNormal(v.normal);

            o.worldPos = mul(unity_ObjectToWorld, v.vertex).xyz;

            o.uv = TRANSFORM_TEX(v.texcoord, _RampTex);

            return o;
        }

        fixed4 frag(v2f i) : SV_Target
        {
            fixed3 worldNormal = normalize(i.worldNormal);
            fixed3 worldLightDir = normalize(UnityWorldSpaceLightDir(i.worldPos));

            fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz;

            // 使用半兰伯特模型。将halfLambert 范围映射到[0,1]之间
            // Use the texture to sample the diffuse color
            fixed halfLambert = 0.5 * dot(worldNormal, worldLightDir) + 0.5;
            // 使用halfLambert构建纹理坐标，并使用这个纹理坐标对渐变纹理_RampTex进行采样
            // _RampTex实际上就是一个一维纹理（纵轴方向上颜色不变），因此u,v我们都是用
halfLambert
            // 从渐变纹理采样得到的颜色和材质颜色_Color相乘，得到漫反射最终颜色
            fixed3 diffuseColor = tex2D(_RampTex, fixed2(halfLambert,
halfLambert)).rgb * _Color.rgb;

            fixed3 diffuse = _LightColor0.rgb * diffuseColor;
```

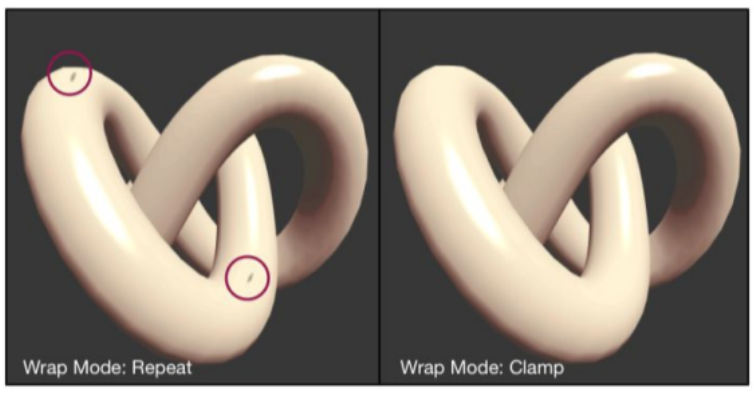
```
                fixed3 viewDir = normalize(UnityWorldSpaceViewDir(i.worldPos));
                fixed3 halfDir = normalize(worldLightDir + viewDir);
                fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(worldNormal, halfDir)), _Gloss);

                return fixed4(ambient + diffuse + specular, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Specular"
}
```

渐变纹理贴图效果，以及使用它们的 Shader 情况如下



注意

我们需要把渐变纹理的 Wrap Mode 设置为 **Clamp** 模式，以防止对纹理采样时由于**浮点数精度造成的问题**



当我们使用 fixed(halfLambert, halfLambert) 对渐变纹理进行采样时，会存在 1.0001 这样的值出现，如果使用的是 Repeat 模式，此时就会舍弃整数部分，只保留小数 0.0001，对应渐变左图中出现高光区域反面有黑点的情况

4. 遮罩纹理 Mask Texture

- 它可以保护某些区域，使它们免于某些修改
  - 前面高光反射模型运用到模型表面所有地方都拥有同样大小的高光强度和高光指数，但是我们希望模型表面某些区域的反射更强烈，而某些区域弱一些，可以得到更加细腻的效果
  - 制作地形材质的时候需要混合多张图片，使用遮罩纹理可以控制如何混合这些纹理
- 遮罩纹理在很多商业游戏中都有应用
- 让美术人员更加精准（像素值级别）控制模型表面的各种属性

流程

- 采样得到遮罩纹理的纹素值
- 使用某个（或某几个通道值）与某种表面属性相乘
- 当通道的值为 0 时，就可以保护表面不受该属性影响

Shader 实践

```
Shader "Custom/Assignment/Chapter7/5_MaskTexture"
{
    Properties
    {
        _Diffuse ("Color Tint", Color) = (1, 1, 1, 1)
        _MainTex ("Main Tex", 2D) = "white" {}
        _BumpMap ("Normal Map", 2D) = "bump" {}
        _BumpScale ("Bump Scale", Float) = 1.0
        _SpecularMask("Specular Mask", 2D) = "white"{} // 高光反射遮罩纹理
        _SpecularScale("Specular Scale", Float) = 1.0 // 控制遮罩影响度的系数
        _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 _Diffuse;
            sampler2D _MainTex;
            float4 _MainTex_ST;
            sampler2D _BumpMap;
            float4 _BumpMap_ST;
            float _BumpScale;
            sampler2D _SpecularMask;
            float _SpecularScale;
            fixed4 _Specular;
            float _Gloss;

            struct a2v
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float4 tangent : TANGENT; // this is a float4 variable
                float4 texcoord : TEXCOORD0;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
                float4 uv : TEXCOORD0;
                float3 lightDir : TEXCOORD1;
                float3 viewDir : TEXCOORD2;
            };
```

```
            v2f vert(a2v v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                o.uv.xy = v.texcoord.xy * _MainTex_ST.xy + _MainTex_ST.zw;
                o.uv.zw = v.texcoord.xy * _BumpMap_ST.xy + _BumpMap_ST.zw;
                TANGENT_SPACE_ROTATION;
                o.lightDir = mul(rotation, ObjSpaceLightDir(v.vertex)).xyz;
                o.viewDir = mul(rotation, ObjSpaceViewDir(v.vertex)).xyz;
                return o;
            }

            fixed4 frag(v2f i) : SV_Target
            {
                fixed3 tangentLightDir = normalize(i.lightDir);
                fixed3 tangentViewDir = normalize(i.viewDir);

                fixed4 packedNoamal = tex2D(_BumpMap, i.uv.zw);
                fixed3 tangentNormal = UnpackNormal(packedNoamal);
                tangentNormal.xy *= _BumpScale;
                tangentNormal.z = sqrt(1.0 - saturate(dot(tangentNormal.xy,
tangentNormal.xy)));

                // Use the texture to sample the diffuse color
                fixed3 albedo = tex2D(_MainTex, i.uv).rgb * _Diffuse.rgb;

                fixed3 ambient = UNITY_LIGHTMODEL_AMBIENT.xyz * albedo;

                fixed3 diffuse = _LightColor0.rgb * albedo * max(0, dot(tangentNormal,
tangentLightDir));

                fixed3 halfDir = normalize(tangentLightDir + tangentViewDir);

                // 对遮罩纹理进行采样（本书使用的遮罩纹理中每个纹素的rgb分量是一样的），表明了该点对应
的高光反射强度
                fixed specularMask = tex2D(_SpecularMask, i.uv).r * _SpecularScale;
                fixed3 specular = _LightColor0.rgb * _Specular.rgb * pow(max(0,
dot(tangentNormal, halfDir)), _Gloss) * specularMask;

                return fixed4(ambient + diffuse + specular, 1.0);
            }
            ENDCG
        }
    }
    FallBack "Specular"
}
```

注意

> Dota2 的制作有详细的游戏人物模型、纹理以及制作手册可供学习

在这个实践案例中，有很多空间被浪费了，它的 rgb 分量存储的是同一个值，在实际的游戏制作中，往往会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

- 高光反射
- 边缘光照强度
- 高光反射的指数部分
- 自发光强度