# 18.6 着色器代码

我们将法线贴图映射的一般处理过程总结如下：

1．使用某个绘图软件或工具软件创建法线贴图，并保存为图像文件。当程序初始化时，从这些文件创建2D纹理。

2．为每个三角形计算切线向量**T**。在网格中，顶点**v**的切线向量等于共享该顶点的每个三角形的切线向量的平均值。（在我们的演示程序中，我们为简单几何体直接指定切线向量，为那些从3D建模软件导出的复杂三角形网格计算平均值。）

3．在顶点着色器中将顶点的法线向量和切线向量变换到世界空间，然后将结果输出到像素着色器。

4．使用插值后的切线向量和法线向量，为三角形表面上的每个像素点生成TBN基。使用TBN基，将从法线贴图采样得到的法线向量从切线空间变换到世界空间。最后将法线向量用于光照计算。

要实现法线法线贴图映射，我们需要在*lighthelper.fx*中添加以下函数：

//-------------------------------------------------------------------------------------

// 将法线贴图的采样值转换到世界空间.

//-------------------------------------------------------------------------------------

float3 NormalSampleToWorldSpace(float3 normalMapSample, float3 unitNormalW, float3 tangentW)

{

// 将每个分量从[0,1]解压到[-1,1].

float3 normalT = 2.0f\*normalMapSample - 1.0f;

// 创建TBN基.

float3 N = unitNormalW;

float3 T = normalize(tangentW - dot(tangentW, N)\*N);

float3 B = cross(N, T);

float3x3 TBN = float3x3(T, B, N);

// 从切线空间转换到世界空间.

float3 bumpedNormalW = mul(normalT, TBN);

return bumpedNormalW;

}

这个函数在像素着色器中的使用方法如下：

float3 normalMapSample = gNormalMap.Sample(samLinear, pin.Tex).rgb;

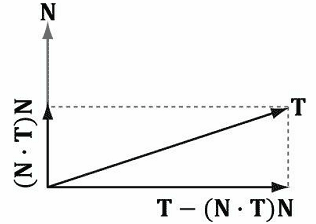
float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample, pin.NormalW, pin.TangentW);

这里有两行代码需要特别说明一下：

float3 N = unitNormalW;

float3 T = normalize(tangentW - dot(tangentW, N)\*N);

在插值之后，切线向量和法线向量可能不再相互垂直。这两行代码的用途是通过从**T**中减去偏向于**N**的部分（见图18.6），使T重新垂直于**N**。注意，我们假设**unitNormalW**已被规范化。

****

**图18.6 因为‖N‖=1，所以projN(T) = (T·N)N。向量T− projN(T)是T垂直于N的部分。**

在从法线贴图中获取了法线之后，就可以将它用在所有包含法线向量的计算中（即，光照、立方贴图映射）。我们将整个法线贴图映射的effect代码都列了出来。

#include "LightHelper.fx"

cbuffer cbPerFrame

{

DirectionalLight gDirLights[3];

float3 gEyePosW;

float gFogStart;

float gFogRange;

float4 gFogColor;

};

cbuffer cbPerObject

{

float4x4 gWorld;

float4x4 gWorldInvTranspose;

float4x4 gWorldViewProj;

float4x4 gTexTransform;

Material gMaterial;

};

// Nonnumeric values cannot be added to a cbuffer.

Texture2D gDiffuseMap;

Texture2D gNormalMap;

TextureCube gCubeMap;

SamplerState samLinear

{

Filter = MIN\_MAG\_MIP\_LINEAR;

AddressU = WRAP;

AddressV = WRAP;

};

struct VertexIn

{

float3 PosL : POSITION;

float3 NormalL : NORMAL;

float2 Tex : TEXCOORD;

float3 TangentL : TANGENT;

};

struct VertexOut

{

float4 PosH : SV\_POSITION;

float3 PosW : POSITION;

float3 NormalW : NORMAL;

float3 TangentW : TANGENT;

float2 Tex : TEXCOORD;

};

VertexOut VS(VertexIn vin)

{

VertexOut vout;

// Transform to world space space.

vout.PosW = mul(float4(vin.PosL, 1.0f), gWorld).xyz;

vout.NormalW = mul(vin.NormalL, (float3x3)gWorldInvTranspose);

vout.TangentW = mul(vin.TangentL, (float3x3)gWorld);

// Transform to homogeneous clip space.

vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj);

// Output vertex attributes for interpolation across triangle.

vout.Tex = mul(float4(vin.Tex, 0.0f, 1.0f), gTexTransform).xy;

return vout;

}

float4 PS(VertexOut pin,

uniform int gLightCount,

uniform bool gUseTexure,

uniform bool gAlphaClip,

uniform bool gFogEnabled,

uniform bool gReflectionEnabled) : SV\_Target

{

// Interpolating normal can unnormalize it, so normalize it.

pin.NormalW = normalize(pin.NormalW);

// The toEye vector is used in lighting.

float3 toEye = gEyePosW - pin.PosW;

// Cache the distance to the eye from this surface point.

float distToEye = length(toEye);

// Normalize.

toEye /= distToEye;

// Default to multiplicative identity.

float4 texColor = float4(1, 1, 1, 1);

if(gUseTexure)

{

// Sample texture.

texColor = gDiffuseMap.Sample( samLinear, pin.Tex );

if(gAlphaClip)

{

// Discard pixel if texture alpha < 0.1. Note that we do this

// test as soon as possible so that we can potentially exit the shader

// early, thereby skipping the rest of the shader code.

clip(texColor.a - 0.1f);

}

}

//

// 法线映射

//

float3 normalMapSample = gNormalMap.Sample(samLinear, pin.Tex).rgb;

float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample, pin.NormalW, pin.TangentW);

//

// Lighting.

//

float4 litColor = texColor;

if( gLightCount > 0 )

{

// Start with a sum of zero.

float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);

float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);

float4 spec = float4(0.0f, 0.0f, 0.0f, 0.0f);

// Sum the light contribution from each light source.

[unroll]

for(int i = 0; i < gLightCount; ++i)

{

float4 A, D, S;

ComputeDirectionalLight(gMaterial, gDirLights[i], bumpedNormalW, toEye,

A, D, S);

ambient += A;

diffuse += D;

spec += S;

}

litColor = texColor\*(ambient + diffuse) + spec;

if( gReflectionEnabled )

{

float3 incident = -toEye;

float3 reflectionVector = reflect(incident, bumpedNormalW);

float4 reflectionColor = gCubeMap.Sample(samLinear, reflectionVector);

litColor += gMaterial.Reflect\*reflectionColor;

}

}

//

// Fogging

//

if( gFogEnabled )

{

float fogLerp = saturate( (distToEye - gFogStart) / gFogRange );

// Blend the fog color and the lit color.

litColor = lerp(litColor, gFogColor, fogLerp);

}

// Common to take alpha from diffuse material and texture.

litColor.a = gMaterial.Diffuse.a \* texColor.a;

return litColor;

}