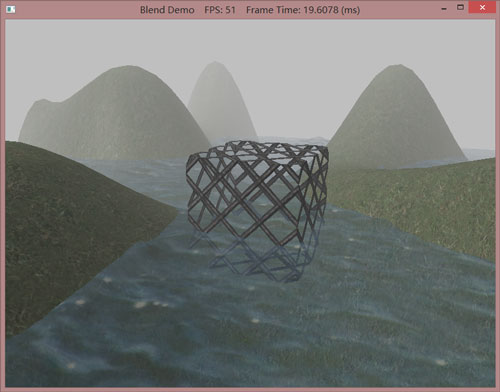
# 9.8 雾

当我们在游戏中模拟某些类型的天气状况时，可能会用到雾效（参见图9.11）。雾除了本身所具有的用途外，还具有一些附加效用。例如，雾可以用来掩盖渲染过程中出现的不自然的人工痕迹，避免蹿出问题的发生。蹿出（popping）是指由于摄像机的移动，使原本在远平面后面的物体突然进入平截头体内，从不可见变为可见；这看上去就像是突然“蹿”到场景里面一样。通过在一定距离内加入雾效，可以掩盖这一问题。注意，即使你的场景是在晴朗的白天，你也可以在较远的地方加入一些淡淡的雾气。因为就算是晴天，远处的物体（比如山岳）也会看上去有些模糊，就像是有一层薄薄的雾笼罩在上面一样，当深度增加时，物体的对比度会逐渐减小。我们可以用雾来模拟这种大气透视现象。

****

**图9.11 雾演示程序的屏幕截图。**

我们用如下方法来实现雾效：我们为雾指定一个颜色、一个相对于摄像机的起始位置和一个范围（即，该范围从雾的起始位置开始到完全遮隐任何物体为止）。那么，三角形表面点的颜色等于点的照颜色与雾颜色的加权平均值：

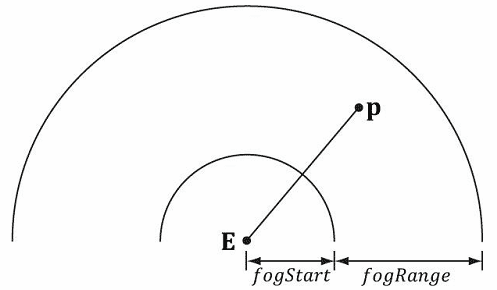
*foggedColor* = *litColor* + *s* (*fogColor* − *litColor*) = (1 − *s* ) ∙ *litColor* + *s* ∙ *fogColor*

参数*s*的取值范围是从0到1，它是一个以表面点和观察点之间的距离为自变量的函数。随着表面点和观察点之间的距离增大，雾在表面点颜色中所占的比例会越来越大。该参数的定义如下：



其中，dist(**p,E**)是表面点**p**和观察点**E**之间的距离。**saturate**函数将自变量限定在[0,1]区间内：



****

**图9.12 从观察点到表面点的距离，以及*fogStart*和*fogRange*参数。**

图9.13是距离函数*s*的曲线图。我们可以看到，当dist(**p,E**)≤*fogStart*时，*s* = 0且雾化颜色为：

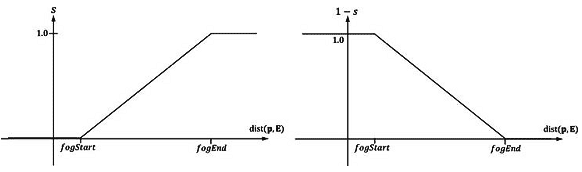
*foggedColor* = *litColor*

换句话说，当顶点与观察点之间的距离小于*fogStart*时，雾不会影响顶点颜色。从*fogStart*这个名字就可以猜到：只有当顶点与观察点之间的距离超过了*fogStart*这个分界线时，雾才会开始影响顶点颜色。

设*fogEnd* = *fogStart* + *fogRange*，当dist(**p**,**E**)≥*fogEnd*时，*s* = 1且雾化颜色为：

*foggedColor* = *fogColor*

换句话说，当表面点与观察点之间的距离大于等于*fogEnd*时，雾将完全取代表面点本身的光照颜色。



**图9.13 （上图）距离函数*s*（雾色权重）的曲线图。（下图）距离函数 1 − *s*（光照颜色权重）的曲线图。当*s*增大时，1 − *s*会减小相同的量。**

我们可以看到，当*fogStart* < dist(**p**,**E**) < *fogEnd*时，随着dist(**p**,**E**)从*fogStart*增加到*fogEnd*，*s*会线性地从0增加到1。这说明当距离增加时，雾色所占的比重会越来越大，而表面点的光照颜色所占的比重会越来越小。这很容易理解，因为距离越远，被雾色笼罩的表面点就越多。

下面的着色器代码示范了雾的实现方法。我们在顶点级别上计算距离和插值参数，然后进行插值，在像素级别上完成照颜色的计算。

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;

SamplerState samAnisotropic

{

Filter = ANISOTROPIC;

MaxAnisotropy = 4;

AddressU = WRAP;

AddressV = WRAP;

};

struct VertexIn

{

float3 PosL : POSITION;

float3 NormalL : NORMAL;

float2 Tex : TEXCOORD;

};

struct VertexOut

{

float4 PosH : SV\_POSITION;

float3 PosW : POSITION;

float3 NormalW : NORMAL;

float2 Tex : TEXCOORD;

};

VertexOut VS(VertexIn vin)

{

VertexOut vout;

// 转换到世界空间

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

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

// 转换到齐次剪裁空间

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) : SV\_Target

{

// 插值后的法线需要重新归一化

pin.NormalW = normalize(pin.NormalW);

// toEye矢量用于光照计算

float3 toEye = gEyePosW - pin.PosW;

// 保存观察点到表面的距离

float distToEye = length(toEye);

// 规范化

toEye /= distToEye;

// Default to multiplicative identity.

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

if(gUseTexure)

{

// 采样纹理

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

if(gAlphaClip)

{

// 如果纹理的alpha<0.1，则丢弃像素。

// 注意，我们应该尽可能早地进行这个测试，这样我们就可以及早退出

// shader，忽略其余shader代码。

clip(texColor.a - 0.1f);

}

}

//

// 光照

//

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], pin.NormalW, toEye,

A, D, S);

ambient += A;

diffuse += D;

spec += S;

}

// Modulate with late add.

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

}

//

// 雾化

//

if( gFogEnabled )

{

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

// 混合雾颜色和光照颜色

litColor = lerp(litColor, gFogColor, fogLerp);

}

// 从漫反射材质和纹理中提取alpha

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

return litColor;

}

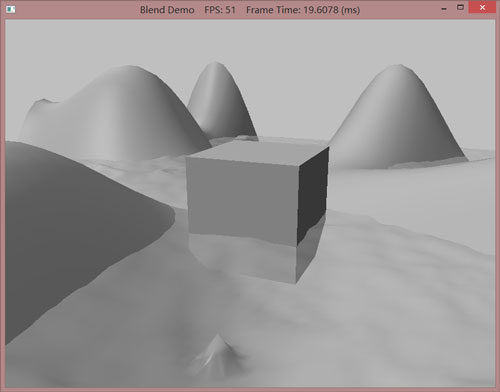
**注意**：在雾效计算中，我们使用了**distToEye**，这个值还用来归一化**toEye**矢量，下面的代码也可以用来归一化toEye矢量，但不够优化：

float3 toEye = normalize(gEyePosW - pin.PosW);

float distToEye = distance(gEyePosW, pin.PosW);

上述代码必须计算两次**toEye**矢量的长度，一次在**normalize**函数中，一次在**distance**函数中。

**注意**：“Blend”演示程序支持三种渲染模式，可以通过按‘1’，‘2’，‘3’键进行切换。第一个模式只绘制了光照（见图9.14）。光照前面已经讨论过了，但看一下没有纹理的场景还是有用处的。第二个模式绘制了光照和纹理（图9.10）。第三个模式绘制了光照、纹理和雾效（图9.11）。这些渲染模式的切换是为了说明我们是如何在effect框架中通过uniform参数生成不同效果。读者也可以看一下Basic.fx的汇编代码，也可以比较一下三种模式下的帧频。

****

**图9.14 只开启光照时的“Blend”示例程序截图**