# 7.12 实现

## 7.12.1 光照结构体

在**LightHelper.h**文件中，我们定义了下面的结构体来表示平行光、点光或聚光灯。

struct DirectionalLight

{

 DirectionalLight() { ZeroMemory(this, sizeof(this)); }

 XMFLOAT4 Ambient;

 XMFLOAT4 Diffuse;

 XMFLOAT4 Specular;

 XMFLOAT3 Direction;

 float Pad; // 占位最后一个float，这样我们就可以设置光源数组了。

};

struct PointLight

{

 PointLight() { ZeroMemory(this, sizeof(this)); }

 XMFLOAT4 Ambient;

 XMFLOAT4 Diffuse;

 XMFLOAT4 Specular;

 // 打包到4D矢量: (Position, Range)

 XMFLOAT3 Position;

 float Range;

 // 打包到4D矢量: (A0, A1, A2, Pad)

 XMFLOAT3 Att;

 float Pad; // 占位最后一个float，，这样我们就可以设置光源数组了。

};

struct SpotLight

{

 SpotLight() { ZeroMemory(this, sizeof(this)); }

 XMFLOAT4 Ambient;

 XMFLOAT4 Diffuse;

 XMFLOAT4 Specular;

 // 打包到4D矢量: (Position, Range)

 XMFLOAT3 Position;

 float Range;

 // 打包到4D矢量: (Direction, Spot)

 XMFLOAT3 Direction;

 float Spot;

 // 打包到4D矢量: (Att, Pad)

 XMFLOAT3 Att;

 float Pad; // 占位最后一个float，，这样我们就可以设置光源数组了。

};

**1．Ambient**：由光源发射的环境光的数量。

**2．Diffuse**：由光源发射的漫反射光的数量。

**3．Specular**：由光源发射的高光的数量。

**4．Direction**：灯光方向。

**5．Position**：灯光位置。

**6．Range**：光照范围（离开光源的距离大于这个值的点不会被照亮）。

**7．Attenuation**：按照（*a*0、*a*1和*a*2）的顺序存储3个衰减常量。衰减常量只用于点和聚光灯，用于控制光强随距离衰减的程度。

**8．Spot**：该指数用于控制聚光灯的圆锥体区域大小；这个值只用于聚光灯。

我们会在下一节中讨论“pad”变量的必要性和“packing”格式。

定义在LightHelper.fx文件中的结构体镜像了上面的结构体：

struct DirectionalLight

{

 float4 Ambient;

 float4 Diffuse;

 float4 Specular;

 float3 Direction;

 float pad;

};

struct PointLight

{

 float4 Ambient;

 float4 Diffuse;

 float4 Specular;

 float3 Position;

 float Range;

 float3 Att;

 float pad;

};

struct SpotLight

{

 float4 Ambient;

 float4 Diffuse;

 float4 Specular;

 float3 Position;

 float Range;

 float3 Direction;

 float Spot;

 float3 Att;

 float pad;

};

## 7.12.2结构Packing

定义了HLSL结构体后，我们就可以像以下代码一样初始化常量缓冲：

cbuffer cbPerFrame

{

 DirectionalLight gDirLight;

 PointLight gPointLight;

 SpotLight gSpotLight;

 float3 gEyePosW;

} ;

在应用程序中初始化对应的灯光结构体，我们想在一次调用中将灯光实例设置到effect变量中，而不是单独地设置每个成员变量。可以用以下函数将一个结构实例设置到一个effect变量实例上：

ID3DX11EffectVariable::SetRawValue(void \*pData,

 UINT Offset,UINT Count);

// Example call:

DirectionalLight mDirLight;

mfxDirLight->SetRawValue(&mDirLight, 0, sizeof(mDirLight));

但是，由于这个函数只是简单地复制原始字节，因此如果你不仔细的话会导致很难找到错误，这里所说的要仔细对待的东西是指C++的打包（packing）规则与HLSL不同。

在HLSL中，当元素打包到要给4D矢量中时会发生结构填充（padding），根据HLSL的规则，一个元素无法拆分到两个4D矢量中。考虑下面的例子：

//HLSL

struct S

{

 float3 Pos;

 float3 Dir;

} ;

如果将这个数据打包在4D矢量中，你可能以为会是这样：

vector1:(Pos.x,Pos.y,Pos.z,Dir.x)

vector2:(Dir.y,Dir.z,empty,empty)

但是，将元素dir打包在两个不同的4D矢量中违反了HLSL的规则，是不被允许的——一个元素不可以跨过一个4D矢量的范围，它只能这样被打包：

vector1:(Pos.x,Pos.y,Pos.z,empty)

vector2:(Dir.x,Dir.y,Dir.z,empty)

现在假设我们对应的C++结构如下所示：

// C++

struct S

{

 XMFLOAT3 Pos;

 XMFLOAT3 Dir;

};

如果我们不注意打包规则，只是盲目地调用并复制数据，我们就会得到错误的结果：

vector1:(Pos.x,Pos.y,Pos.z,Dir.x)

vector2:(Dir.y,Dir.z,empty,empty)

所以我们必须基于HLSL打包规则定义C++结构，才能让元素正确地复制到HLSL结构中。我们要使用一个“pad”变量解决上述问题。让我们看几个HLSL打包的例子。

例如有下列一个结构：

struct S

{

 float3 v;

 float s;

 float2 p;

 float3 q;

} ;

这个结构会有空隙，数据会被打包到3个4D矢量中：

vector1:(v.x,v.y,v.z,s)

vector2:(p.x,p.y,empty,empty)

vector3:(q.x,q.y,q.z,empty)

我们可以将标量放在第1个矢量的第4个分量中。但是，我们无法在第2个矢量中匹配q的全部分量，所以需要为它分配单独的矢量。

最后一个例子，结构：

struct S

{

 float2 u;

 float2 v;

 float a0;

 float a1;

 float a2;

} ;

会这样被打包：

vector1:(u.x,u.y,v.x,v.y)

vector2:(a0,a1,a2,empty)

**注意**：数组的处理方法有所不同。SDK文档中这样说明：“数组中的每个元素都存储在一个4个分量的矢量中。”所以，若你有一个float2的数组：

float2 TexOffsets[8];

你可能会认为如上面的例子所说，两个float2会包装在一个float4中。但是，数组是例外，上面的代码等同于：

float4 TexOffsets[8];

所以，你需要在C++中定义一个8个XMFLOAT4的数组，而不是8个XMFLOAT2的数组，这样才能正常运行。我们实际上只需要一个float2数组，所以每个元素浪费了两个float的空间。SDK文档指出你可以使用转换和额外的地址计算指令提高内存的使用效率：

float4 array[4];

static float2 aggressivePackArray[8] = (float2[8])array;

## 7.12.3 实现平行光

下面的HLSL函数根据给出的材质、平行光源、表面法线和由表面指向观察点的矢量，输出光照后的表面颜色。

void ComputeDirectionalLight(Material mat, DirectionalLight L,

 float3 normal, float3 toEye,

 out float4 ambient,

 out float4 diffuse,

 out float4 spec)

{

 // 初始化输出的变量

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

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

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

 // 光照矢量与光线的传播方向相反

 float3 lightVec = -L.Direction;

 // 添加环境光

 ambient = mat.Ambient \* L.Ambient;

 // 添加漫反射和镜面光

 float diffuseFactor = dot(lightVec, normal);

 // Flatten避免动态分支

 [flatten]

 if( diffuseFactor > 0.0f )

 {

 float3 v = reflect(-lightVec, normal);

 float specFactor = pow(max(dot(v, toEye), 0.0f), mat.Specular.w);

 diffuse = diffuseFactor \* mat.Diffuse \* L.Diffuse;

 spec = specFactor \* mat.Specular \* L.Specular;

 }

}

其中，使用的HLSL内置函数有：**dot**、**reflect**、**pow**和**max**，它们分别用来计算向量点积、向量反射、乘方和取最大值。读者可以在附录B中找到大部分HLSL内置函数的描述，以及有关HLSL语法的快速入门。另外还有一件事情需要注意，那就是当两个向量使用\*运算符相乘时，实际执行的是分量乘法。

**注意**：PC上的HLSL函数总是内联的，调用函数或传递参数不会有性能损失。

## 7.12.4 实现点光

下面的HLSL函数根据给出的材质、点光源、表面位置、表面法线和表面点到观察点的的矢量信息，输出光照后的表面颜色。

void ComputePointLight(Material mat, PointLight L, float3 pos, float3 normal, float3 toEye,

 out float4 ambient, out float4 diffuse, out float4 spec)

{

 // 初始化输出变量

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

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

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

 // 表面指向光源的矢量

 float3 lightVec = L.Position - pos;

 // 表面离光源的距离

 float d = length(lightVec);

 // Range test.

 if( d > L.Range )

 return;

 // Normalize the light vector.

 lightVec /= d;

 // Ambient term.

 ambient = mat.Ambient \* L.Ambient;

 // Add diffuse and specular term, provided the surface is in

 // the line of site of the light.

 float diffuseFactor = dot(lightVec, normal);

 // Flatten to avoid dynamic branching.

 [flatten]

 if( diffuseFactor > 0.0f )

 {

 float3 v = reflect(-lightVec, normal);

 float specFactor = pow(max(dot(v, toEye), 0.0f), mat.Specular.w);

 diffuse = diffuseFactor \* mat.Diffuse \* L.Diffuse;

 spec = specFactor \* mat.Specular \* L.Specular;

 }

 // 衰减

 float att = 1.0f / dot(L.Att, float3(1.0f, d, d\*d));

 diffuse \*= att;

 spec \*= att;

}

## 7.12.5 实现聚光灯

下面的HLSL函数根据给出的材质、聚光灯、表面位置、表面法线、表面点指向观察点位置的矢量信息，输出光照后的表面颜色：

void ComputeSpotLight(Material mat, SpotLight L, float3 pos, float3 normal, float3 toEye,

 out float4 ambient, out float4 diffuse, out float4 spec)

{

 // 初始化输出变量.

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

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

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

 // 从表面指向光源的光照矢量

 float3 lightVec = L.Position - pos;

 // 表面离开光源的距离

 float d = length(lightVec);

 // Range test.

 if( d > L.Range )

 return;

 // 规范化光照矢量

 lightVec /= d;

 // 计算环境光

 ambient = mat.Ambient \* L.Ambient;

 // 计算漫反射和镜面光，provided the surface is in

 // the line of site of the light.

 float diffuseFactor = dot(lightVec, normal);

 // Flatten避免动态分支

 [flatten]

 if( diffuseFactor > 0.0f )

 {

 float3 v = reflect(-lightVec, normal);

 float specFactor = pow(max(dot(v, toEye), 0.0f), mat.Specular.w);

 diffuse = diffuseFactor \* mat.Diffuse \* L.Diffuse;

 spec = specFactor \* mat.Specular \* L.Specular;

 }

 // Scale by spotlight factor and attenuate.

 float spot = pow(max(dot(-lightVec, L.Direction), 0.0f), L.Spot);

 // Scale by spotlight factor and attenuate.

 float att = spot / dot(L.Att, float3(1.0f, d, d\*d));

 ambient \*= spot;

 diffuse \*= att;

 spec \*= att;

}