# 6.8 Effects

effect框架是一组用于管理着色器程序和渲染状态的工具代码。例如，你可能会使用不同的effect绘制水、云、金属物体和动画角色。每个effect至少要由一个顶点着色器、一个像素着色器和渲染状态组成。

在Direct3D 11中，effects框架已从D3DX库中移除，你必须包含一个单独的头文件（**d3dx11Effect.h**），链接一个单独的库文件（**D3DX11Effects.lib**用于release生成，而**D3DX11EffectsD.lib**用于debug生成）。

而且，在Direct3D 11中提供了effect库的完整源代码（DirectX SDK\Samples\C++\Effects11）。因此，你可以根据需要修改effect框架。本书中，我们只是使用、并不会修改effect框架。要使用这个库，首先需要生成Effects11项目的Release和Debug模式，用于获得**D3DX11Effects.lib**和**D3DX11EffectsD.lib**文件，除非effect框架进行了更新（例如，新版本的DirectX SDK可能会更新这些文件，这时就需要重新生成.lib文件），这个步骤只需进行一次。**d3dx11Effect.h**头文件可在DirectX SDK\Samples\C++\Effects11\Inc文件夹中找到。在示例代码中，我们将**d3dx11Effect.h**，**D3DX11EffectsD.lib**和**D3DX11Effects.lib**文件都放在Common文件夹中，这样所有的项目文件都能共享这些文件。

## 6.8.1 Effect文件

我们已经讨论了顶点着色器、像素着色器，并对几何着色器、曲面细分着色器进行了简要概述。我们还讨论了常量缓冲，它可以用于存储由着色器访问的“全局”变量。这些代码通常保存在一个effect文件（.fx）中，它是一个纯文本文件中（就像是C++代码保存在.h和.cpp文件中一样）。除了着色器和常量缓冲之外，每个effect文件至少还要包含一个technique，而每个technique至少要包含一个pass。

**1．technique11**：一个technique由一个或多个pass组成，用于创建一个渲染技术。每个pass实现一种不同的几何体渲染方式，按照某些方式将多个pass的渲染结果混合在一起就可以得到我们最终想要的渲染结果。例如，在地形渲染中我们将使用多通道纹理映射技术（multi-pass texturing technique）。注意，多通道技术通常会占用大量的系统资源，因为每个pass都要对几何体进行一次渲染；不过，要实现某些渲染效果，我们必须使用多通道技术。

**2．pass：**一个pass由一个顶点着色器、一个可选的几何着色器、一个像素着色器和一些渲染状态组成。这些部分定义了pass的几何体渲染方式。像素着色器也是可选的（很罕见）。例如，若我们只想绘制深度缓冲，不想绘制后台缓冲，在这种情况下我们就不需要像素着色器计算像素的颜色。

**注意**：techniques也可以组合在一起成为effect组。如果你没有显式地定义一个effect组，那么编译器会创建一个匿名effect组，把所有technique包含在effect文件中。本书中，我们不显式地定义effect组。

下面是本章演示程序使用的effect文件：

cbuffer cbPerObject

{

float4x4 gWorldViewProj;

};

struct VertexIn

{

float3 PosL : POSITION;

float4 Color : COLOR;

};

struct VertexOut

{

float4 PosH : SV\_POSITION;

float4 Color : COLOR;

};

VertexOut VS(VertexIn vin)

{

VertexOut vout;

// 转换到齐次剪裁空间

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

// 将顶点颜色直接传递到像素着色器

vout.Color = vin.Color;

return vout;

}

float4 PS(VertexOut pin) : SV\_Target

{

return pin.Color;

}

technique11 ColorTech

{

pass P0

{

SetVertexShader( CompileShader( vs\_5\_0, VS() ) );

SetPixelShader( CompileShader( ps\_5\_0, PS() ) );

}

}

**注意**：点和向量可以在许多不同的空间中描述（例如，局部空间、世界空间、观察空间、齐次裁剪空间）。当阅读代码时，有时很难看出点和向量的坐标系是相对于哪个坐标系的。所以，我们经常使用下面的后缀来表示空间：L（局部空间）、W（世界空间）、V（观察空间）、H（齐次裁剪空间）。下面是一些例子：

float3 iPosL; // local space

float3 gEyePosW; // world space

float3 normalV; // view space

float4 posH; // homogeneous clip space

前面提到，pass可以包含渲染状态。也就是，状态块可以直接在effect文件中创建和指定。当effect需要特定的渲染状态时，这种方式非常实用；但是，当某些effect需要在运行过程中改变渲染状态时，我们更倾向于在应用程序层执行状态设定，因为这样进行状态切换更方便一些。下面的代码示范了如何在一个effect文件中创建和指定光栅化状态块。

RasterizerState Wireframe

{

FillMode = Wireframe;

CullMode = Back;

FrontCounterClockwise = false;

// 我们没有设置的属性使用默认值

};

technique11 ColorTech

{

pass P0

{

SetVertexShader( CompileShader( vs\_5\_0, VS() ) );

SetPixelShader( CompileShader( ps\_5\_0, PS() ) );

SetRasterizerState(Wireframe);

}

}

可以看到，在光栅化状态对象中定义的常量与C++中的枚举成员基本相同，只是省去了前缀而已（例如，**D3D11\_FILL\_**和**D3D11\_CULL\_**）。

注意：由于effect通常保存在扩展名为.fx的文件中，所以在修改effect代码之后，不必重新编译C++源代码。

## 6.8.2 编译着色器

创建一个effect的第一步是编译定义在.fx文件中的着色器程序，可以由下面的D3DX方法完成：

HRESULT D3DX11CompileFromFile (

LPCTSTR pSrcFile ,

CONST D3D10\_SHADE R\_MACRO \*pDefines,

LPD3D10INCLUDE pInclude ,

LPCSTR pFunctionName ,

LPCSTR pProfile,

UINT Flags 1,

UINT Flags 2,

ID3DX11ThreadPump \*pPump ,

ID3D10Blob \*\*ppShader,

ID3D10Blob \*\*ppErrorMsgs,

HRESULT \*pHResult);

**1．pSrcFile**：.fx文件名，该文件包含了我们所要编译的效果源代码。

**2．pDefines**：高级选项，我们不使用；请参阅SDK文档。

**3．pInclude**：高级选项，我们不使用；请参阅SDK文档。

**4．pFunctionName**：着色器入口函数的名字。只用于单独编译着色器程序的情况。当使用effect框架时设置为null，这是因为在effect文件中已经定义了入口点。

**5．pProfile**：用于指定着色器版本的字符串。对于Direct3D 11来说，我们使用的着色器版本为5.0（“fx\_5\_0”）。

**6．Flags1**：用于指定着色器代码编译方式的标志值。SDK文档列出了很多标志值，但本书只使用其中的2个：

* **D3D10\_SHADER\_DEBUG**：以调试模式编译着色器。
* **D3D10\_SHADER\_SKIP\_OPTIMIZATION**：告诉编译器不做优化处理（用于进行调试）。

**7．Flags2**：高级选项，我们不使用；请参阅SDK文档。

**8．pPump**：指向线程泵的指针，多线程编程时使用，是高级选项，我们不使用；请参阅SDK文档。本书中这个值都设为null。

**9．ppShader**：返回一个指向**ID3D10Blob**数据对象的指针，这个数据对象保存了经过编译的代码。

**10．ppErrorMsgs**：返回一个指向**ID3D10Blob**数据对象的指针，这个数据对象存储了一个包含错误信息的字符串。

**11．pHResult**：在使用异步编译时，用于获得返回的错误代码。仅当使用pPump时才使用该参数；我们在本书中将该参数设为空值。

**注意**：1．除了可以编译在.fx文件内的着色器代码，这个方法也可以编译单独的着色器代码。有些程序不使用effect框架，它们会单独的定义和编译自己的着色器代码。

2．方法中的指向“D3D10”的引用并不是打印错误。因为D3D11编译器是建立在D3D10的编译器之上的，所以Direct3D 11开发组就没有修改某些标识的名称。

3．**ID3D10Blob**只是一个通用内存块，它有两个方法：

（a）LPVOID GetBufferPointer：返回指向数据的一个void\*，所以在使用时应该对它执行相应的类型转换（具有请参见下面的示例）。

（b）SIZE\_T GetBufferSize：返回缓冲的大小，以字节为单位。

编译完成后，我们就可以使用下面的方法创建一个effect（用**ID3DXEffect11**接口表示）：

HRESULT D3DX11CreateEffectFromMemory(

void \*pData,

SIZE\_T DataLength,

UINT FXFlags ,

ID3D11Device \*pDevice,

ID3DX11Effect \*\*ppEffect);

**1．pData**：指向编译好的effect数据的指针。

**2．DataLength**：effect数据的长度，以字节为单位。

**3．FXFlags**：Effect标识必须与定义在**D3DX11CompileFromFile**方法中的**Flags2**匹配。

**4．pDevice**：指向Direct3D 11设备的指针。

**5．ppEffect**：指向创建好的effect的指针。

下面的代码演示了如何编译并创建一个effect：

DWORD shaderFlags = 0;

#ifdefined (DEBUG)||defined(\_DEBUG)

shaderFlags |= D3D10\_SHADER\_DEBUG;

shaderFlags |= D3D10\_SHADER\_SKIP\_OPTIMIZATION ;

#endif

ID3D10Blob \* compiledShader = 0;

ID3D10Blob \* compilationMsgs = 0;

HRESULT hr = D3DX11CompileFromFile(L"color.fx", 0,

0, 0, "fx\_5\_0", shaderFlags,

0, 0, &compiledShader, &compilationMsgs, 0);

// compilationMsgs包含错误或警告的信息

if(compilationMsgs ! = 0)

{

MessageBoxA(0, (char\*)compilationMsgs->GetBufferPointer(), 0, 0);

ReleaseCOM(compilationMsgs);

}

// 就算没有compilationMsgs，也需要确保没有其他错误

if(FAILED(hr))

{

DXTrace(\_\_FILE\_\_,(DWORD)\_\_LINE\_\_,hr,L"D3DX11Compile FromFile",true);

}

ID3DX11Effect\* mFX;

HR(D3DX11CreateEffectFromMemory(

compiledShader->Ge tBufferPointer(),

compiledShader->Ge tBufferSize(),

0, md3dDevice, &mFX));

// 编译完成释放资源

ReleaseCOM(compiledShader);

**注意**：创建Direct3D资源代价昂贵，尽量在初始化阶段完成，即创建输入布局、缓冲、渲染状态对象和effect应该总在初始化阶段完成。

## 6.8.3 在C++应用程序中与Effect进行交互

C++应用程序代码通常要与effect进行交互；尤其是C++应用程序经常要更新常量缓冲中的变量。例如，在一个effect文件中，我们有如下常量缓冲定义：

cbuffer cbPerObject

{

float4x4 gWVP;

float4 gColor;

float gSize;

int gIndex;

bool gOptionOn;

};

通过**ID3D11Effect**接口，我们可以获得指向常量缓冲变量的指针：

ID3D11EffectMatrixVariable\* fxWVPVar;

ID3D11EffectVectorVariable\* fxColorVar;

ID3D11EffectScalarVariable\* fxSizeVar;

ID3D11EffectScalarVariable\* fxIndexVar;

ID3D11EffectScalarVariable\* fxOptionOnVar;

fxWVPVar = mFX->GetVariableByName("gWVP")->AsMatrix();

fxColorVar = mFX->GetVariableByName("gColor")->AsVector();

fxSizeVar = mFX->GetVariableByName("gSize")->AsScalar();

fxIndexVar = mFX->GetVariableByName("gIndex")->AsScalar();

fxOptionOnVar = mFX->GetVariableByName("gOptionOn")->AsScalar();

**ID3D11Effect::GetVariableByName**方法返回一个**ID3D11EffectVariable**指针。它是一种通用effect变量类型；要获得指向特定类型变量的指针（例如，矩阵、向量、标量），你必须使用相应的As-方法（例如，**AsMatrix**、**AsVector**、**AsScalar**）。

一旦我们获得变量指针，我们就可以通过C++接口来更新它们了。下面是一些例子：

fxWVPVar->SetMatrix( (float\*)&M ); // assume M is of type XMMATRIX

fxColorVar->SetFloatVector( (float\*)&v ); // assume v is of type XMVECTOR

fxSizeVar->>SetFloat( 5.0f );

fxIndexVar->SetInt( 77 );

fxOptionOnVar->SetBool( true );

注意，这些语句修改的只是effect对象在系统内存中的一个副本，它并没有传送到GPU内存中。所以在执行绘图操作时，我们必须使用**Apply**方法更新GPU内存（参见6.8.4节）。这样做的原因是为了提高效率，避免频繁地更新GPU内存。如果每修改一个变量就要更新一次GPU内存，那么效率会很低。

注意：effect变量不一定要被类型化。例如，可以有如下代码：

ID3D11EffectVariable\* mfxEyePosVar;

mfxEyePosVar = mFX->GetVariableByName("gEyePosW");

...

mfxEyePosVar->SetRawValue(&mEyePos, 0, sizeof(XMFLOAT3));

这种方式可以用来设置任意大小的变量（例如，普通结构体）。注意，**ID3D11EffectVectorVariable**接口使用4D向量。如果你希望使用3D向量的话，那应该像上面那样使用**ID3D11EffectVariable**接口。

除了常量缓冲变量之外，我们还需要获得指向technique对象的指针。实现方法如下：

ID3D11EffectTechnique\* mTech;

mTech = mFX->GetTechniqueByName("ColorTech");

该方法只包含一个用于指定technique名称的字符串参数。

## 6.8.4 使用effect绘图

要使用technique来绘制几何体，我们只需要确保对常量缓冲中的变量进行实时更新。然后，使用循环语句来遍历technique 中的每个pass，使用pass来绘制几何体：

// 设置常量缓冲

XMMATRIX world = XMLoadFloat4x4(&mWorld);

XMMATRIX view = XMLoadFloat4x4(&mView);

XMMATRIX proj = XMLoadFloat4x4(&mProj);

XMMATRIX worldViewProj = world\*view\*proj;

mfxWorldViewProj->SetMatrix(reinterpret\_cast<float\*>(&worldViewPr

oj));

D3DX11\_TECHNIQUE\_DESC techDesc;

mTech->GetDesc(&techDesc);

for(UINT p = 0;p < techDesc.Passes;++p )

{

mTech->GetPassByIndex(p)->Apply(0,md3dImmediateContext);

// 绘制几何体

md3dImmediateContext->DrawIndexed(36, 0, 0);

}

当使用pass来绘制几何体时，Direct3D会启用在pass中指定的着色器和渲染状态。**ID3D11EffectTechnique::GetPassByIndex**方法返回一个指定索引的pass对象的**ID3D11EffectPass**接口指针。**Apply**方法更新存储在GPU内存中的常量缓冲、将着色器程序绑定到管线、并启用在pass中指定的各种渲染状态。在当前版本的Direct3D 11中，**ID3D11EffectPass::Apply**方法的第一个参数还未使用，应设置为0；第二个参数指向pass使用的设备上下文的指针。

如果你需要在绘图调用之间改变常量缓冲中的变量值，那你必须在绘制几何体之前调用**Apply**方法：

for(UINT i = 0; i < techDesc.Passes; ++i)

{

ID3D11EffectPass\* pass = mTech->GetPassByIndex(i);

//设置地面几何体的WVP组合矩阵

worldViewProj = mLandWorld\*mView\*mProj;

mfxWorldViewProj->SetMatrix(reinterpret\_cast<float\*>(& worldViewProj);

pass->Apply(0, md3dImmediateContext);

mLand.draw();

// 设置水波几何体的WVP组合矩阵

worldViewProj = mWavesWorld\*mView\*mProj;

mfxWorldViewProj->SetMatrix(reinterpret\_cast<float\*>(& worldViewProj);

pass->Apply(0 ,md3dImmediateContext);

mWaves.draw();

}

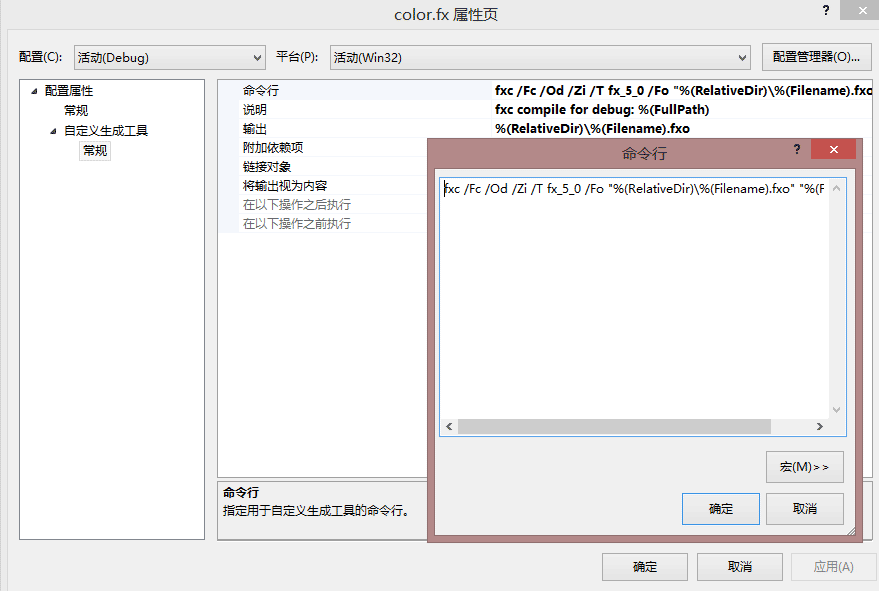
## 6.8.5 在生成期间编译effect

我们已经介绍了如何在运行时通过**D3DX11CompileFromFile**方法编译一个effect。但这样做会带来一个小小的不便：如果你的effect文件有一个编译错误，直到程序运行时你才会发现这个错误。我们还可以使用DirectX SDK自带的*fxc*工具（位于DirectX SDK\Utilities\bin\x86）离线编译你的effect。而且，你还可以修改你的VC++项目，将调用fxc编译effect的过程作为生成过程的一部分。步骤如下：

**1．**确保路径DirectX SDK\Utilities\bin\x86位于你的项目的VC++目录的“可执行文件目录（Executable Directories）”之下。

**2．**在项目中添加effect文件。

**3．**在解决方案资源管理器中右击每个effect文件选择**属性**，添加自定义生成工具（见图6.6）：

****

**图6.6 在项目中添加自定义生成工具**

调试模式：

**fxc /Fc /Od /Zi /T fx\_5\_0 /Fo " % (RelativeDir)\% (Filename).fxo " " % (FuIIPath) "**

发布模式：

**fxc /T fx\_5\_0 /Fo " o/o (RelativeDir)\% (Filename).fxo” " % (FuIIPath) "**

你可以在SDK文档中找到fxc完整的编译参数说明。在调试模式中我们使用了以下三个参数，“/Fc /Od /Zi”分别对应输出汇编指令，禁用优化，开启调试信息。

现在当生成项目时，就会在每个effect上调用*fxc*并生成它的编译版本，以后缀为.fxo的文件的形式保存。而且，如果有来自于*fxc*的编译警告或错误，会在调试输出窗口显示相关信息。例如，如果在color.fx文件中打错了一个变量的名称：

// 应该是gWorldViewProj而不是worldViewProj!

vout.PosH = mul(float4(vin.Pos, l.Of), worldViewProj);

在调试输出窗口就会显示从这个错误引发的一系列错误的信息（第一条错误信息是修正的关键）：

error X3004: undeclared identifier 'worldViewProj'

error X3013: 'mul': intrinsic function does not take 2 parameters

error X3013: Possible intrinsic functions are:

error X3013: mul(float, float)…

在编译阶段获取错误信息要比运行时获取方便得多。现在我们在生成过程中编译effect文件（.fxo），再也不需要在运行时进行这个操作了（即，我们无须再调用**D3DX11CompileFromFile**方法了）。但是，我们仍需要从.fxo文件中加载编译过的shader，并将它们传递给**D3DX11CreateEffectFromMemory**方法。这个工作可以通过使用C++的文件输入功能实现：

std::ifstream fin("fx/color.fxo",std::ios::binary);

fin.seekg(0, std::ios\_base::end);

int size = (int)fin.tellg();

fin.seekg(0, std::ios\_base::beg);

std::vector<char> compiledShader(size);

fin.read(&compiledShader[0],size);

fin.close();

HR(D3DX11CreateEffectFromMemory(&compiledShader[0],size,

0, md3dDevice, &mFX));

除了在颜色立方体演示程序中我们在运行时编译了shader之外，本书的其他示例都是在生成过程中编译了所有shader。

## 6.8.6 将effect框架作为“着色器生成器”

在本节的一开始我们提到过一个effect可以包含多个technique。那为什么我们要使用多个technique呢？下面我们用阴影绘制为例子解释一下这个问题，但不会讨论实现阴影的细节内容。显然，阴影质量越高，要求的资源就越多。为了支持用户不同等级的显卡，我们可能会提供低、中、高不同质量的阴影技术。因此，即使只有一个阴影效果，我们也会使用多个technique去实现它。我们的阴影effect文件如下所示：

// 省略了常量缓冲，顶点结构等代码...

VertexOut VS(Vertexln vin) {/\* Omit implementation details \*/}

float4 LowQualityPS(VertexOut pin) : SV\_Target

{

/\* Do work common to all quality levels \*/

/\* Do low quality specific stuff \*/

/\* Do more work common to all quality levels \*/

}

float4 MediumQualityPS(VertexOut pin) : SV\_Target

{

/\* Do work common to all quality levels \*/

/\* Do medium quality specific stuff \*/

/\* Do more work common to all quality levels \*/

}

float4 HighQualityPS(VertexOut pin) : SV\_Target

{

/\* Do work common to all quality levels \*/

/\* Do high quality specific stuff \*/

/\* Do more work common to all quality levels \*/

}

technique11 ShadowsLow

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixeIShader(CompileShader(ps\_5\_0, LowQualityPS()));

}

}

technique11 ShadowsMedium

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixeIShader(CompileShader(ps\_5\_0, MediumQualityPS()));

}

}

technique11 ShadowsHigh

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixeIShader(CompileShader(ps\_5\_0, HighQualityPS ()));

}

}

C++应用程序会侦测玩家的显卡等级，选择最合适的technique进行渲染。

**注意**：前面的代码假设三个不同的阴影technique只在像素着色器中有所区别，所有的technique共享相同的顶点着色器。但是，每个technique都有不同的顶点着色器也是有可能的。

前面的实现中还有一个问题：即使像素着色器的代码是不同的，但是还是有一些通用的代码是重复的。建议使用条件分支语句解决这个问题。在shader中使用动态分支语句代价不菲，所以只在必要时才使用它们。其实我们真正想要的是一个条件编译，它可以生成不同的shader代码，但又不使用分支指令。幸运的是，effect框架提供了一个方法可以解决这个问题。下面是具体实现：

// 省略常量缓冲，顶点结构等...

VertexOut VS(VertexIn vin) {/\* 省略代码细节 \*/}

#define LowQuality 0

#define MediumQuality 1

#define HighQuality 2

float4 PS(VertexOut pin, uniform int gQuality) : SV\_Target

{

/\* Do work common to all quality levels \*/

if(gQuality == LowQuality)

{

/\* Do low quality specific stuff \*/

}

elseif(gQuality == MediumQuality)

{

/\* Do medium quality specific stuff \*/

}

else

{

/\* Do high quality specific stuff \*/

}

/\* Do more work common to all quality levels \*/

}

technique11 ShadowsLow

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixeIShader(CompileShader(ps\_5\_0, PS(LowQuality)));

}

}

technique11 ShadowsMedium

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixeIShader(CompileShader(ps\_5\_0, PS(MediumQuality)));

}

}

techniquell ShadowsHigh

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixeIShader(CompileShader(ps\_5\_0, PS(HighQuality)));

}

}

我们在像素着色器中添加了一个额外的uniform参数，用来表示阴影质量等级。这个参数值是不同的，但对每个像素来说却是不变的，but is instead uniform/constant。Moreover，we do not change it at runtime either，like we change constant buffer variables。我们是在编译时设置这些参数的，而且这些值在编译时就是已知的，所以effect框架会基于这个值生成不同的shader变量。这样，我们不用复制代码（effect框架帮我们在编译时复制了这些代码）就可以生成低、中、高三种不同阴影质量的shader代码，而且没有用到条件分支语句。

下面的两个例子是使用shader生成器的常见情景：

**1．是否需要纹理**？有个应用程序需要在一些物体上施加纹理，而另一些物体不使用纹理。一个解决方法是创建两个像素着色器，一个提供纹理而另一个不提供。或者我们也可以使用shader生成技巧创建两个像素着色器，然后在C++程序中选择期望的technique。

float4 PS(VertexOut pin, uniform bool gApplyTexture) : SV\_Target

{

/\* Do common work \*/

if(gApplyTexture)

{

/\* Apply texture \*/

}

/\* Do more common work \*/

}

technique11 BasicTech

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixelShader(CompileShader(ps\_5\_0, PS(false)));

}

}

technique11 TextureTech

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0,VS()));

SetPixelShader(CompileShader(ps\_5\_0,PS(true)));

}

}

**2．使用多少个光源？**一个游戏关卡可能会支持1至4个光源。光源越多，光照计算就越慢。我们可以基于光源数量设计不同的顶点着色器，或者也可以使用shader生成技巧创建四个顶点着色器，然后在C++程序中根据当前激活的光源数量选择期望的technique：

VertexOut VS(VertexOut pin, uniform int gLightCount)

{

/\* Do common work \*/

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

{

/\* do lighting work \*/

}

/\* Do more common work \*/

}

technique11 Light1

{

P0

{

SetVertexShader(CompileShader(vs\_5\_0, VS(1)));

SetPixeIShader(CompileShader(ps\_5\_0, PS()));

}

}

technique11 Light2

{

P0

{

SetVertexShader(CompileShader(vs\_5\_0, VS(2)));

SetPixeIShader(CompileShader(ps\_5\_0, PS()));

}

}

technique11 Light3

{

P0

{

SetVertexShader(CompileShader(vs\_5\_0, VS(3)));

SetPixeIShader(CompileShader(ps\_5\_0, PS()));

}

}

technique11 Light4

{

P0

{

SetVertexShader(CompileShader(vs\_5\_0, VS(4)));

SetPixeIShader(CompileShader(ps\_5\_0, PS()));

}

}

参数也可以不止一个。要将阴影质量，纹理和多个光源组合在一起，我们可以使用以下的顶点和像素着色器：

VertexOut VS(VertexOut pin, uniform int gLightCount)

{}

float4 PS(VertexOut pin,uniform int gQuality,uniform bool gApplyTexture) : SV\_Target

{}

要创建一个使用低质量阴影，两个光源，不使用纹理的technique，我们可以这样写代码：

technique11 LowShadowsTwoLightsNoTextures

{

pass P0

{

SetVertexShader(CompileShader(vs\_5\_0, VS(2)));

SetPixeIShader(CompileShader(ps\_5\_0,PS(LowQuality,false)));

}

}

## 6.8.7 对应的汇编代码

In case you do not bother to ever look at the assembly output of one of your effect files, we show what one looks like in this section.

我们不会解释汇编代码，但是，如果你以前学过汇编，你可能会认得代码中的mov指令，and

maybe could guess thatdp4 does a 4D dot product. Even without understanding the assembly, the listing gives some useful

information. It clearly identifies the input and output signatures, and gives the approximate mstruction count, which is one useful metric

to understand how expensive/complex your shader is. Moreover, we see that multiple versions of our shaders really were generated based on the compile time parameters with no branching instructions. The effect file we use is the same as the one shown in ~6.8.1,except that we add a simple uniform bool parameter to generate two techniques:

float4 PS(VertexOut pin, uniform bool gUseColor) : SV\_Target

{

if(gUs e Color)

{

return pin.Color;

else

{

return float4(0, 0, 0, 1); techniquell ColorTech

{

pass PO

{

SetVe rtexShade r(Compile Shade r(vs\_5\_0, VSO》;

SetPixeIShader(CompileShader(ps\_5\_0, PS(true》);

techniquell NoColorTech

{

pass PO

{

SetVertexShader(CompileShade r(vs\_5\_0, VS(》);

SetPixeIShader(CompileShader(ps\_5\_0, PS(false》);

//

// FX Version: fx\_5\_0

//

11 1 local buffer(s)

//

cbuffer cbPerObject

{

float4x4 gWorldViewProj; // Offset: O, size: 64

//

11 1 groups(s)

//

fxgroup

{

//

11 2 technique(s)

//

techniquell ColorTech

{

pass PO

{

VertexShader = asm {

//

// Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111

//

//

// Buffer Definitions:

//

// cbuffer cbPerObject

// {

//

// float4x4 gWorldViewProj; // Offset: O Size: 64

//

// }

//

//

// Resource Bindings:

//

// Name Type Format Dim Slot Elements

// -----.

// cbPerObject cbuffer NA NA 0 1

// //

//

// Input signature:

//

// Name Index Mask Register SysValue Format Used

// -----

// POSlrflON 0 xyz O NONE float xyz

// COLOR O xyzw l NONE float xyzw

//

//

// Output signature:

//

// Name Index Mask Register SysValue Format Used

// -----

// SV\_POSITION O xyzw O POS float xyzw

// COLOR O xyzw l NONE float xyzw

//

vs\_5\_0

dcl\_globaIFlags refactoringAllowed

dcl\_constantbuffe r cb0 [4], imme diatelnde xe d

dcljnput vO.xyz

dcl\_input vl.xyzw

dcl\_output\_siv oO.xyzw, position

dcl\_output ol.xyzw

dcl\_temps 1

mov rO.xyz, vO.xyzx

mov rO.w, l(l.000000)

dp4 00.x, rO.xyzw, cbO[O].xyzw

dp4 00.y, rO.xyzw, cbO[l].xyzw

dp4 00.z, rO.xyzw, cb0[2].xyzw

dp4 00.w, rO.xyzw, cb0[3].xyzw

mov ol.xyzw, vl.xyzw

ret

// Approximately 8 instruction slots used

};

PixeIShader = asm {

//

// Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111

//

//

//

// Input signature:

//

// Name Index Mask Register SysValue Format Used

// -----

// SV\_POSIUON O xyzw o POS float

// COLOR O xyzw l NONE float xyzw

//

//

// Output signature:

//

// Name Index Mask Register SysValue Format Used

// -----

// SV\_Target O xyzw O TARGET float xyzw

//

ps\_5\_0

dcl\_globaIFlags refactoringAllowed

dcl\_input\_ps linear vl.xyzw

dcl\_output oO.xyzw

mov oO.xyzw, vl.xyzw

ret // Approximately 2 instruction slots used

};

techniquell NoColorTech

{

pass PO

{

VertexShader = asm {

//

// Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111

//

//

// Buffer Definitions:

//

// cbuffer cbPerObject

// {

//

// float4x4 gWorldViewProj; // Offset: O Size: 64

//

// }

//

//

// Resource Bindings:

//

// Name Type Format Dim Slot Elements

// -----.

// cbPerObject cbuffer NA NA 0 1

//

//

//

// Input signature:

//

// Name Index Mask Register SysValue Format Used

// -----.

// POSlrflON O xyz O NONE float xyz

// COLOR O xyzw l NONE float xyzw

//

//

// Output signature:

//

// Name Index Mask Register SysValue Format Used

// -----.

// SV\_POSITION O xyzw O POS float xyzw

// COLOR 0 xyzw l NONE float xyzw

//

vs\_5\_0

dcl\_globaIFlags refactoringAllowed

dcl\_constantbuffe r cb0 [4], imme diatelnde xe d

dcljnput vO.xyz

dcl\_input vl.xyzw

dcl\_output\_siv oO.xyzw, position

dcl\_output ol.xyzw

dcl\_temps 1

mov rO.xyz, vO.xyzx

mov rO.w, 1(1.000000)

dp4 00.x, rO.xyzw, cbO[O].xyzw

dp4 00.y, rO.xyzw, cbO[l].xyzw

dp4 00.z, rO.xyzw, cb0[2].xyzw

dp4 00.w, rO.xyzw, cb0[3].xyzw

mov ol.xyzw, vl.xyzw}

}

}

ret

// Approximately 8 instruction slots used

};

PixeIShader = asm {

Generated by Microsoft (R) HLSL Shader Compiler 9.29.952.3111

Input signature:

Name Index Mask Register SysValue Format Used

SV\_POSITION O xyzw O POS float

COLOR O xyzw l NONE float

Output signature:

Name Index Mask Register SysValue Format Used

SV\_Target O xyzw O TARGET float xyzw

ps\_5\_0

dcl\_globaIFlags refactoringAllowed

dcl\_output oO.xyzw

mov oO.xyzw, 1(0,0,0,1.000000)

ret

// Approximately 2 instruction slots used

6.9 BOX DEMO

At last, we have covered enough material to present a simple demo, which renders a colored box. This example essentially puts

everythmg we have discussed in this chapter up to now into a single program. The reader should study the code and refer back to the

previous sections of this chapter until every line is understood. Note that the program uses the "color.fx" effect, as written in ~6.8.1.

，／击击出士击击出士击出去击击出去击击出士击击出士击击击击击出去击击出去击击出士击击出士击出去击击出去击击古士击击出士击击出去击出去击击出去击

// BoxDemo.cpp by Frank Luna (C) 2011 All Rights Reserved.

//

／／Demonstrates rendering a colored box.

||

/／Controls：

／／Hold the left mouse button down and move the mouse to rotate.

／／Hold the right mouse button down to zoom in and out.

||

，／击出去击击出去击击古去士击古去士出去击击出去击击出去士击古去士击古去击出去击击出去击击古去士击古去士击击击击出去击击出去士击古去士击古去击

#include "d3dApp.h"

#include "d3dxllEffect.h"

#include "MathHelper.h"

struct Vertex

XMFLOArl3 Pos;

XMFLOAT4 Color;

∥∥∥∥∥∥∥∥∥∥∥∥∥∥∥∥∥∥∥

暂未整理。