# Direct3D 11教程6：光照

原文地址：<http://msdn.microsoft.com/en-us/library/ff729723.aspx>。

## 概览

在前面的教程中，因为所有物体都是以相同的方式被光照亮，因此还不够真实。本教程将介绍简单的光照概念以及如何施加光照，这个技术将使用Lambert光照模型。

这个教程的会在前一个示例的基础上包含一个光源，光源链接在轨道运行的立方体上，光照的效果可以在中央立方体表面上看到。



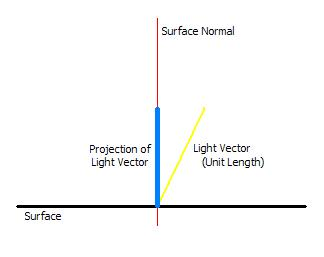
## 源代码

(SDK root)\Samples\C++\Direct3D11\Tutorials\Tutorial06

## 光照

在本教程中将介绍最简单的光照类型：**Lambert光照**。Lambert光照具有相同的光照强度而不考虑离开光源的距离。当光照射到表面时，反射光的强度根据光在表面上的入射角大小进行计算。当光线垂直射向表面，反射回所有光照，具有最大的光强。当入射角变大时，光的强度也会减弱。

要计算表面的光照强度，必须计算光照方向和表面法线之间的夹角。法线定义为垂直于平面的一个矢量，角度的计算只需使用一个简单的点乘即可，返回的结果是光照方向矢量在法线上的投影长度。入射角越大，投影长度越短，这样，我们就有一个正确的方法调整漫反射光照。



本教程使用的光源接近于单向光，表示光源的矢量决定了光线的方向。因为这是一个近似，所以物体的位置无关紧要，我们认为光照方向都是一样的。这种光源的一个例子就是太阳，在一个场景中，太阳光总是被认为是同一方向的。

另一种类型的光源是点光源，从光源中心发出光线；还有一种是聚光灯，它发出的光线具有方向性但又不是均匀照射所有物体。

## 初始化光照

在本教程中，我们使用两个光源。一个静止放置在立方体的上后方，另一个绕着它旋转。

因为光照是由shader进行计算的，所以必须首先声明变量然后将它们与technique绑定。在本例中，我们只需光照方向和颜色值。第一个光源为灰色不移动，第二个是一个沿轨道绕行的红色光源。

// 设置光源参数

XMFLOAT4 vLightDirs[2] =

{

XMFLOAT4( -0.577f, 0.577f, -0.577f, 1.0f ),

XMFLOAT4( 0.0f, 0.0f, -1.0f, 1.0f ),

};

XMFLOAT4 vLightColors[2] =

{

XMFLOAT4( 0.5f, 0.5f, 0.5f, 1.0f ),

XMFLOAT4( 0.5f, 0.0f, 0.0f, 1.0f )

};

第二个光源还像上一个教程中的立方体那样旋转，施加的矩阵会改变光照方向，让光照总是朝向中心。注意，**XMVec3Transform**方法用来在矢量上乘以矩阵。在前面的教程中，我们只是将变换矩阵乘进世界矩阵中，然后传递到shader用于变换；但本例中基于简化的考虑，我们在CPU中对光源进行世界变换。

// 使第2个正方体绕原点旋转

// Rotate the second light around the origin

XMMATRIX mRotate = XMMatrixRotationY( -2.0f \* t );

XMVECTOR vLightDir = XMLoadFloat4( &vLightDirs[1] );

vLightDir = XMVector3Transform( vLightDir, mRotate );

XMStoreFloat4( &vLightDirs[1], vLightDir );

光照方向和颜色都传递到shader中，对应的变量被调用并设置，参数被传递。

//

// 更新矩阵和光源变量

//

ConstantBuffer cb1;

cb1.mWorld = XMMatrixTranspose( g\_World );

cb1.mView = XMMatrixTranspose( g\_View );

cb1.mProjection = XMMatrixTranspose( g\_Projection );

cb1.vLightDir[0] = vLightDirs[0];

cb1.vLightDir[1] = vLightDirs[1];

cb1.vLightColor[0] = vLightColors[0];

cb1.vLightColor[1] = vLightColors[1];

cb1.vOutputColor = XMFLOAT4(0, 0, 0, 0);

g\_pImmediateContext->UpdateSubresource( g\_pConstantBuffer, 0, NULL, &cb1, 0, 0 );

## 在像素着色器中绘制光照

设置了所有数据我们就可以计算每一个像素的lambert光照因子了，使用的是上面提到过的点乘规则。

有了点乘结果后，就可以将这个结果乘以光照颜色计算光照的影响了。这个值被传递到saturate方法，这个方法将结果限定在[0, 1]区间。最后，两个光源的效果会相加获得最终的像素颜色。

物体表面材质并没有考虑在光照计算中，表面的最终颜色只包含光照颜色。

//

// Pixel Shader

//

float4 PS( PS\_INPUT input) : SV\_Target

{

float4 finalColor = 0;

//对两个光源进行NdotL光照计算

for(int i=0; i<2; i++)

{

finalColor += saturate( dot( (float3)vLightDir[i],input.Norm) \* vLightColor[i] );

}

return finalColor;

}

经过像素着色器的处理，像素的颜色就被光照调制，你可以在中央立方体表面看到光照效果。因为本例中同一表面的法线方向相同，所以同一表面的光照强度相同。漫反射（**Diffuse**）光照是一个非常简单容易的光照模型，你可以使用更复杂的光照模型实现更加真实的效果。

## 完整代码

#include <windows.h>

#include <d3d11.h>

#include <d3dcompiler.h>

#include <directxmath.h>

#include "resource.h"

using namespace DirectX;

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

// 顶点结构

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

struct SimpleVertex

{

XMFLOAT3 Pos;

XMFLOAT3 Normal;

};

struct ConstantBuffer

{

XMMATRIX mWorld;

XMMATRIX mView;

XMMATRIX mProjection;

XMFLOAT4 vLightDir[2];

XMFLOAT4 vLightColor[2];

XMFLOAT4 vOutputColor;

};

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

// 全局变量

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

HINSTANCE g\_hInst = NULL;

HWND g\_hWnd = NULL;

D3D\_DRIVER\_TYPE g\_driverType = D3D\_DRIVER\_TYPE\_NULL;

D3D\_FEATURE\_LEVEL g\_featureLevel = D3D\_FEATURE\_LEVEL\_11\_0;

ID3D11Device\* g\_pd3dDevice = NULL;

ID3D11DeviceContext\* g\_pImmediateContext = NULL;

IDXGISwapChain\* g\_pSwapChain = NULL;

ID3D11RenderTargetView\* g\_pRenderTargetView = NULL;

ID3D11Texture2D\* g\_pDepthStencil = NULL;

ID3D11DepthStencilView\* g\_pDepthStencilView = NULL;

ID3D11VertexShader\* g\_pVertexShader = NULL;

ID3D11PixelShader\* g\_pPixelShader = NULL;

ID3D11PixelShader\* g\_pPixelShaderSolid = NULL;

ID3D11InputLayout\* g\_pVertexLayout = NULL;

ID3D11Buffer\* g\_pVertexBuffer = NULL;

ID3D11Buffer\* g\_pIndexBuffer = NULL;

ID3D11Buffer\* g\_pConstantBuffer = NULL;

XMMATRIX g\_World;

XMMATRIX g\_View;

XMMATRIX g\_Projection;

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

// 函数声明

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

HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow );

HRESULT InitDevice();

void CleanupDevice();

LRESULT CALLBACK WndProc( HWND, UINT, WPARAM, LPARAM );

void Render();

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

// Entry point to the program. Initializes everything and goes into a message processing

// loop. Idle time is used to render the scene.

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

int WINAPI wWinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPWSTR lpCmdLine, int nCmdShow )

{

UNREFERENCED\_PARAMETER( hPrevInstance );

UNREFERENCED\_PARAMETER( lpCmdLine );

if( FAILED( InitWindow( hInstance, nCmdShow ) ) )

return 0;

if( FAILED( InitDevice() ) )

{

CleanupDevice();

return 0;

}

// Main message loop

MSG msg = {0};

while( WM\_QUIT != msg.message )

{

if( PeekMessage( &msg, NULL, 0, 0, PM\_REMOVE ) )

{

TranslateMessage( &msg );

DispatchMessage( &msg );

}

else

{

Render();

}

}

CleanupDevice();

return ( int )msg.wParam;

}

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

// 注册和创建窗口

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

HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )

{

// Register class

WNDCLASSEX wcex;

wcex.cbSize = sizeof( WNDCLASSEX );

wcex.style = CS\_HREDRAW | CS\_VREDRAW;

wcex.lpfnWndProc = WndProc;

wcex.cbClsExtra = 0;

wcex.cbWndExtra = 0;

wcex.hInstance = hInstance;

wcex.hIcon = LoadIcon( hInstance, ( LPCTSTR )IDI\_TUTORIAL1 );

wcex.hCursor = LoadCursor( NULL, IDC\_ARROW );

wcex.hbrBackground = ( HBRUSH )( COLOR\_WINDOW + 1 );

wcex.lpszMenuName = NULL;

wcex.lpszClassName = L"TutorialWindowClass";

wcex.hIconSm = LoadIcon( wcex.hInstance, ( LPCTSTR )IDI\_TUTORIAL1 );

if( !RegisterClassEx( &wcex ) )

return E\_FAIL;

// Create window

g\_hInst = hInstance;

RECT rc = { 0, 0, 640, 480 };

AdjustWindowRect( &rc, WS\_OVERLAPPEDWINDOW, FALSE );

g\_hWnd = CreateWindow( L"TutorialWindowClass", L"Direct3D 11 Tutorial 6", WS\_OVERLAPPEDWINDOW,

CW\_USEDEFAULT, CW\_USEDEFAULT, rc.right - rc.left, rc.bottom - rc.top, NULL, NULL, hInstance,

NULL );

if( !g\_hWnd )

return E\_FAIL;

ShowWindow( g\_hWnd, nCmdShow );

return S\_OK;

}

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

// Helper for compiling shaders with D3DCompile

//

// With VS 11, we could load up prebuilt .cso files instead...

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

HRESULT CompileShaderFromFile( WCHAR\* szFileName, LPCSTR szEntryPoint, LPCSTR szShaderModel, ID3DBlob\*\* ppBlobOut )

{

HRESULT hr = S\_OK;

DWORD dwShaderFlags = D3DCOMPILE\_ENABLE\_STRICTNESS;

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

// Set the D3DCOMPILE\_DEBUG flag to embed debug information in the shaders.

// Setting this flag improves the shader debugging experience, but still allows

// the shaders to be optimized and to run exactly the way they will run in

// the release configuration of this program.

dwShaderFlags |= D3DCOMPILE\_DEBUG;

#endif

ID3DBlob\* pErrorBlob;

hr = D3DCompileFromFile( szFileName, NULL, NULL, szEntryPoint, szShaderModel,

dwShaderFlags, 0, ppBlobOut, &pErrorBlob );

if( FAILED(hr) )

{

if( pErrorBlob != NULL )

OutputDebugStringA( (char\*)pErrorBlob->GetBufferPointer() );

if( pErrorBlob ) pErrorBlob->Release();

return hr;

}

if( pErrorBlob ) pErrorBlob->Release();

return S\_OK;

}

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

// 创建Direct3D设备和交换链

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

HRESULT InitDevice()

{

HRESULT hr = S\_OK;

RECT rc;

GetClientRect( g\_hWnd, &rc );

UINT width = rc.right - rc.left;

UINT height = rc.bottom - rc.top;

UINT createDeviceFlags = 0;

#ifdef \_DEBUG

createDeviceFlags |= D3D11\_CREATE\_DEVICE\_DEBUG;

#endif

D3D\_DRIVER\_TYPE driverTypes[] =

{

D3D\_DRIVER\_TYPE\_HARDWARE,

D3D\_DRIVER\_TYPE\_WARP,

D3D\_DRIVER\_TYPE\_REFERENCE,

};

UINT numDriverTypes = ARRAYSIZE( driverTypes );

D3D\_FEATURE\_LEVEL featureLevels[] =

{

D3D\_FEATURE\_LEVEL\_11\_0,

D3D\_FEATURE\_LEVEL\_10\_1,

D3D\_FEATURE\_LEVEL\_10\_0,

};

UINT numFeatureLevels = ARRAYSIZE( featureLevels );

DXGI\_SWAP\_CHAIN\_DESC sd;

ZeroMemory( &sd, sizeof( sd ) );

sd.BufferCount = 1;

sd.BufferDesc.Width = width;

sd.BufferDesc.Height = height;

sd.BufferDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

sd.BufferDesc.RefreshRate.Numerator = 60;

sd.BufferDesc.RefreshRate.Denominator = 1;

sd.BufferUsage = DXGI\_USAGE\_RENDER\_TARGET\_OUTPUT;

sd.OutputWindow = g\_hWnd;

sd.SampleDesc.Count = 1;

sd.SampleDesc.Quality = 0;

sd.Windowed = TRUE;

for( UINT driverTypeIndex = 0; driverTypeIndex < numDriverTypes; driverTypeIndex++ )

{

g\_driverType = driverTypes[driverTypeIndex];

hr = D3D11CreateDeviceAndSwapChain( NULL, g\_driverType, NULL, createDeviceFlags, featureLevels, numFeatureLevels,

D3D11\_SDK\_VERSION, &sd, &g\_pSwapChain, &g\_pd3dDevice, &g\_featureLevel, &g\_pImmediateContext );

if( SUCCEEDED( hr ) )

break;

}

if( FAILED( hr ) )

return hr;

// Create a render target view

ID3D11Texture2D\* pBackBuffer = NULL;

hr = g\_pSwapChain->GetBuffer( 0, \_\_uuidof( ID3D11Texture2D ), ( LPVOID\* )&pBackBuffer );

if( FAILED( hr ) )

return hr;

hr = g\_pd3dDevice->CreateRenderTargetView( pBackBuffer, NULL, &g\_pRenderTargetView );

pBackBuffer->Release();

if( FAILED( hr ) )

return hr;

// Create depth stencil texture

D3D11\_TEXTURE2D\_DESC descDepth;

ZeroMemory( &descDepth, sizeof(descDepth) );

descDepth.Width = width;

descDepth.Height = height;

descDepth.MipLevels = 1;

descDepth.ArraySize = 1;

descDepth.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;

descDepth.SampleDesc.Count = 1;

descDepth.SampleDesc.Quality = 0;

descDepth.Usage = D3D11\_USAGE\_DEFAULT;

descDepth.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;

descDepth.CPUAccessFlags = 0;

descDepth.MiscFlags = 0;

hr = g\_pd3dDevice->CreateTexture2D( &descDepth, NULL, &g\_pDepthStencil );

if( FAILED( hr ) )

return hr;

// Create the depth stencil view

D3D11\_DEPTH\_STENCIL\_VIEW\_DESC descDSV;

ZeroMemory( &descDSV, sizeof(descDSV) );

descDSV.Format = descDepth.Format;

descDSV.ViewDimension = D3D11\_DSV\_DIMENSION\_TEXTURE2D;

descDSV.Texture2D.MipSlice = 0;

hr = g\_pd3dDevice->CreateDepthStencilView( g\_pDepthStencil, &descDSV, &g\_pDepthStencilView );

if( FAILED( hr ) )

return hr;

g\_pImmediateContext->OMSetRenderTargets( 1, &g\_pRenderTargetView, g\_pDepthStencilView );

// Setup the viewport

D3D11\_VIEWPORT vp;

vp.Width = (FLOAT)width;

vp.Height = (FLOAT)height;

vp.MinDepth = 0.0f;

vp.MaxDepth = 1.0f;

vp.TopLeftX = 0;

vp.TopLeftY = 0;

g\_pImmediateContext->RSSetViewports( 1, &vp );

// Compile the vertex shader

ID3DBlob\* pVSBlob = NULL;

hr = CompileShaderFromFile( L"Tutorial06.fx", "VS", "vs\_4\_0", &pVSBlob );

if( FAILED( hr ) )

{

MessageBox( NULL,

L"The FX file cannot be compiled. Please run this executable from the directory that contains the FX file.", L"Error", MB\_OK );

return hr;

}

// Create the vertex shader

hr = g\_pd3dDevice->CreateVertexShader( pVSBlob->GetBufferPointer(), pVSBlob->GetBufferSize(), NULL, &g\_pVertexShader );

if( FAILED( hr ) )

{

pVSBlob->Release();

return hr;

}

// Define the input layout

D3D11\_INPUT\_ELEMENT\_DESC layout[] =

{

{ "POSITION", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 0, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

{ "NORMAL", 0, DXGI\_FORMAT\_R32G32B32\_FLOAT, 0, 12, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

};

UINT numElements = ARRAYSIZE( layout );

// Create the input layout

hr = g\_pd3dDevice->CreateInputLayout( layout, numElements, pVSBlob->GetBufferPointer(),

pVSBlob->GetBufferSize(), &g\_pVertexLayout );

pVSBlob->Release();

if( FAILED( hr ) )

return hr;

// Set the input layout

g\_pImmediateContext->IASetInputLayout( g\_pVertexLayout );

// Compile the pixel shader

ID3DBlob\* pPSBlob = NULL;

hr = CompileShaderFromFile( L"Tutorial06.fx", "PS", "ps\_4\_0", &pPSBlob );

if( FAILED( hr ) )

{

MessageBox( NULL,

L"The FX file cannot be compiled. Please run this executable from the directory that contains the FX file.", L"Error", MB\_OK );

return hr;

}

// Create the pixel shader

hr = g\_pd3dDevice->CreatePixelShader( pPSBlob->GetBufferPointer(), pPSBlob->GetBufferSize(), NULL, &g\_pPixelShader );

pPSBlob->Release();

if( FAILED( hr ) )

return hr;

// Compile the pixel shader

pPSBlob = NULL;

hr = CompileShaderFromFile( L"Tutorial06.fx", "PSSolid", "ps\_4\_0", &pPSBlob );

if( FAILED( hr ) )

{

MessageBox( NULL,

L"The FX file cannot be compiled. Please run this executable from the directory that contains the FX file.", L"Error", MB\_OK );

return hr;

}

// Create the pixel shader

hr = g\_pd3dDevice->CreatePixelShader( pPSBlob->GetBufferPointer(), pPSBlob->GetBufferSize(), NULL, &g\_pPixelShaderSolid );

pPSBlob->Release();

if( FAILED( hr ) )

return hr;

// Create vertex buffer

SimpleVertex vertices[] =

{

{ XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT3( 0.0f, -1.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT3( 0.0f, -1.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT3( 0.0f, -1.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT3( 0.0f, -1.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT3( -1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT3( -1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT3( -1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT3( -1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT3( 1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT3( 1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT3( 1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT3( 1.0f, 0.0f, 0.0f ) },

{ XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT3( 0.0f, 0.0f, -1.0f ) },

{ XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT3( 0.0f, 0.0f, -1.0f ) },

{ XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT3( 0.0f, 0.0f, -1.0f ) },

{ XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT3( 0.0f, 0.0f, -1.0f ) },

{ XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT3( 0.0f, 0.0f, 1.0f ) },

{ XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT3( 0.0f, 0.0f, 1.0f ) },

{ XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT3( 0.0f, 0.0f, 1.0f ) },

{ XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT3( 0.0f, 0.0f, 1.0f ) },

};

D3D11\_BUFFER\_DESC bd;

ZeroMemory( &bd, sizeof(bd) );

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof( SimpleVertex ) \* 24;

bd.BindFlags = D3D11\_BIND\_VERTEX\_BUFFER;

bd.CPUAccessFlags = 0;

D3D11\_SUBRESOURCE\_DATA InitData;

ZeroMemory( &InitData, sizeof(InitData) );

InitData.pSysMem = vertices;

hr = g\_pd3dDevice->CreateBuffer( &bd, &InitData, &g\_pVertexBuffer );

if( FAILED( hr ) )

return hr;

// Set vertex buffer

UINT stride = sizeof( SimpleVertex );

UINT offset = 0;

g\_pImmediateContext->IASetVertexBuffers( 0, 1, &g\_pVertexBuffer, &stride, &offset );

// Create index buffer

WORD indices[] =

{

3,1,0,

2,1,3,

6,4,5,

7,4,6,

11,9,8,

10,9,11,

14,12,13,

15,12,14,

19,17,16,

18,17,19,

22,20,21,

23,20,22

};

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof( WORD ) \* 36; // 36 vertices needed for 12 triangles in a triangle list

bd.BindFlags = D3D11\_BIND\_INDEX\_BUFFER;

bd.CPUAccessFlags = 0;

InitData.pSysMem = indices;

hr = g\_pd3dDevice->CreateBuffer( &bd, &InitData, &g\_pIndexBuffer );

if( FAILED( hr ) )

return hr;

// Set index buffer

g\_pImmediateContext->IASetIndexBuffer( g\_pIndexBuffer, DXGI\_FORMAT\_R16\_UINT, 0 );

// Set primitive topology

g\_pImmediateContext->IASetPrimitiveTopology( D3D11\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST );

// Create the constant buffer

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof(ConstantBuffer);

bd.BindFlags = D3D11\_BIND\_CONSTANT\_BUFFER;

bd.CPUAccessFlags = 0;

hr = g\_pd3dDevice->CreateBuffer( &bd, NULL, &g\_pConstantBuffer );

if( FAILED( hr ) )

return hr;

// Initialize the world matrices

g\_World = XMMatrixIdentity();

// Initialize the view matrix

XMVECTOR Eye = XMVectorSet( 0.0f, 4.0f, -10.0f, 0.0f );

XMVECTOR At = XMVectorSet( 0.0f, 1.0f, 0.0f, 0.0f );

XMVECTOR Up = XMVectorSet( 0.0f, 1.0f, 0.0f, 0.0f );

g\_View = XMMatrixLookAtLH( Eye, At, Up );

// Initialize the projection matrix

g\_Projection = XMMatrixPerspectiveFovLH( XM\_PIDIV4, width / (FLOAT)height, 0.01f, 100.0f );

return S\_OK;

}

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

// Clean up the objects we've created

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

void CleanupDevice()

{

if( g\_pImmediateContext ) g\_pImmediateContext->ClearState();

if( g\_pConstantBuffer ) g\_pConstantBuffer->Release();

if( g\_pVertexBuffer ) g\_pVertexBuffer->Release();

if( g\_pIndexBuffer ) g\_pIndexBuffer->Release();

if( g\_pVertexLayout ) g\_pVertexLayout->Release();

if( g\_pVertexShader ) g\_pVertexShader->Release();

if( g\_pPixelShaderSolid ) g\_pPixelShaderSolid->Release();

if( g\_pPixelShader ) g\_pPixelShader->Release();

if( g\_pDepthStencil ) g\_pDepthStencil->Release();

if( g\_pDepthStencilView ) g\_pDepthStencilView->Release();

if( g\_pRenderTargetView ) g\_pRenderTargetView->Release();

if( g\_pSwapChain ) g\_pSwapChain->Release();

if( g\_pImmediateContext ) g\_pImmediateContext->Release();

if( g\_pd3dDevice ) g\_pd3dDevice->Release();

}

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

// Called every time the application receives a message

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

LRESULT CALLBACK WndProc( HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam )

{

PAINTSTRUCT ps;

HDC hdc;

switch( message )

{

case WM\_PAINT:

hdc = BeginPaint( hWnd, &ps );

EndPaint( hWnd, &ps );

break;

case WM\_DESTROY:

PostQuitMessage( 0 );

break;

default:

return DefWindowProc( hWnd, message, wParam, lParam );

}

return 0;

}

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

// Render a frame

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

void Render()

{

// Update our time

static float t = 0.0f;

if( g\_driverType == D3D\_DRIVER\_TYPE\_REFERENCE )

{

t += ( float )XM\_PI \* 0.0125f;

}

else

{

static DWORD dwTimeStart = 0;

DWORD dwTimeCur = GetTickCount();

if( dwTimeStart == 0 )

dwTimeStart = dwTimeCur;

t = ( dwTimeCur - dwTimeStart ) / 1000.0f;

}

// Rotate cube around the origin

g\_World = XMMatrixRotationY( t );

// 设置光源参数

XMFLOAT4 vLightDirs[2] =

{

XMFLOAT4( -0.577f, 0.577f, -0.577f, 1.0f ),

XMFLOAT4( 0.0f, 0.0f, -1.0f, 1.0f ),

};

XMFLOAT4 vLightColors[2] =

{

XMFLOAT4( 0.5f, 0.5f, 0.5f, 1.0f ),

XMFLOAT4( 0.5f, 0.0f, 0.0f, 1.0f )

};

// 使第2个光源绕原点旋转

XMMATRIX mRotate = XMMatrixRotationY( -2.0f \* t );

XMVECTOR vLightDir = XMLoadFloat4( &vLightDirs[1] );

vLightDir = XMVector3Transform( vLightDir, mRotate );

XMStoreFloat4( &vLightDirs[1], vLightDir );

//

// Clear the back buffer

//

float ClearColor[4] = { 0.0f, 0.125f, 0.3f, 1.0f }; // red, green, blue, alpha

g\_pImmediateContext->ClearRenderTargetView( g\_pRenderTargetView, ClearColor );

//

// Clear the depth buffer to 1.0 (max depth)

//

g\_pImmediateContext->ClearDepthStencilView( g\_pDepthStencilView, D3D11\_CLEAR\_DEPTH, 1.0f, 0 );

//

// 更新矩阵和光源变量

//

ConstantBuffer cb1;

cb1.mWorld = XMMatrixTranspose( g\_World );

cb1.mView = XMMatrixTranspose( g\_View );

cb1.mProjection = XMMatrixTranspose( g\_Projection );

cb1.vLightDir[0] = vLightDirs[0];

cb1.vLightDir[1] = vLightDirs[1];

cb1.vLightColor[0] = vLightColors[0];

cb1.vLightColor[1] = vLightColors[1];

cb1.vOutputColor = XMFLOAT4(0, 0, 0, 0);

g\_pImmediateContext->UpdateSubresource( g\_pConstantBuffer, 0, NULL, &cb1, 0, 0 );

//

// 绘制立方体

//

g\_pImmediateContext->VSSetShader( g\_pVertexShader, NULL, 0 );

g\_pImmediateContext->VSSetConstantBuffers( 0, 1, &g\_pConstantBuffer );

g\_pImmediateContext->PSSetShader( g\_pPixelShader, NULL, 0 );

g\_pImmediateContext->PSSetConstantBuffers( 0, 1, &g\_pConstantBuffer );

g\_pImmediateContext->DrawIndexed( 36, 0, 0 );

//

// 绘制两个光源

//

for( int m = 0; m < 2; m++ )

{

XMMATRIX mLight = XMMatrixTranslationFromVector( 5.0f \* XMLoadFloat4( &vLightDirs[m] ) );

XMMATRIX mLightScale = XMMatrixScaling( 0.2f, 0.2f, 0.2f );

mLight = mLightScale \* mLight;

// 更新世界矩阵to reflect the current light

cb1.mWorld = XMMatrixTranspose( mLight );

cb1.vOutputColor = vLightColors[m];

g\_pImmediateContext->UpdateSubresource( g\_pConstantBuffer, 0, NULL, &cb1, 0, 0 );

g\_pImmediateContext->PSSetShader( g\_pPixelShaderSolid, NULL, 0 );

g\_pImmediateContext->DrawIndexed( 36, 0, 0 );

}

//

// Present our back buffer to our front buffer

//

g\_pSwapChain->Present( 0, 0 );

}