# Direct3D 11教程7：纹理映射和常量缓存

原文地址：<http://msdn.microsoft.com/en-us/library/ff729724.aspx>。

## 概览

在前面的教程中我们介绍了光照，现在我们将在立方体上添加纹理。我们还会介绍常量缓存的概念，以及如何通过最小化带宽的方式利用常量缓存加速处理过程。

这个教程介绍如何在立方体上施加纹理。



## 源代码

(SDK root)\Samples\C++\Direct3D11\Tutorials\Tutorial07

## 纹理映射

纹理映射表示将一张2D图像映射到3D几何体上。我们可以把它想象成包装一个礼物盒，即将一张包装纸覆盖在一个盒子上。要做到这点，我们需要指定几何体表面上的点是如何对应2D图像的。

诀窍是将正确地将纹理对齐到模型的坐标上。对于复杂的模型，很难手动确定纹理的坐标，因此，3D建模工具通常会输出带有纹理坐标信息的模型。因为本示例是一个立方体，所以很容易确定匹配纹理的坐标。纹理坐标是在顶点中定义的，然后会在表面的单个像素间进行插值。

### 从纹理创建一个Shader Resource

纹理是一张从文件获取的2D图像，用于创建一个shader资源视图（**shader-resource view**），因此它可以从shader中读取。

hr = D3DX11CreateShaderResourceViewFromFile( g\_pd3dDevice, L"seafloor.dds", NULL, NULL,

 &g\_pTextureRV, NULL );

我们还需要创建一个采样状态控制shader如何处理过滤、层级渐进纹理和纹理寻址、本教程中我们使用最简单的采样器，开启了线性过滤，包装寻址模式。我们使用**ID3D11Device::CreateSamplerState()**方法创建采样状态。

// 创建采样状态

D3D11\_SAMPLER\_DESC sampDesc;

ZeroMemory( &sampDesc, sizeof(sampDesc) );

sampDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

sampDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

sampDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

sampDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

sampDesc.ComparisonFunc = D3D11\_COMPARISON\_NEVER;

sampDesc.MinLOD = 0;

sampDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

hr = g\_pd3dDevice->CreateSamplerState( &sampDesc, &g\_pSamplerLinear );

### 定义坐标

我们必须首先定义立方体每个顶点的纹理坐标才能将图像映射到立方体上。因为图像尺寸不定，因此使用的坐标系统被调整到[0, 1]区间。纹理的左上角对应(0,0)，右下角对应(1,1)。

本例中，我们让整个纹理覆盖在立方体的每个面上。你也可以将纹理展开到所有六个面，虽然这样做的话坐标的定义会变得难一些，纹理可能会变形。

首先，我们需要更新结构体使之包含纹理坐标。

struct SimpleVertex

{

 XMFLOAT3 Pos;

 XMFLOAT2 Tex;

};

然后，更新输入布局使之也包含纹理坐标信息。

// Define the input layout

D3D11\_INPUT\_ELEMENT\_DESC layout[] =

{

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

 { "TEXCOORD", 0, DXGI\_FORMAT\_R32G32\_FLOAT, 0, 12, D3D11\_INPUT\_PER\_VERTEX\_DATA, 0 },

};

因为输入布局发生了变化，对应的顶点着色器也需进行调整。

struct VS\_INPUT

{

 float4 Pos : POSITION;

 float2 Tex : TEXCOORD;

};

最后，就可以在顶点中包含纹理坐标了。第二个参数类型为XMFLOAT2，包含了纹理坐标。立方体的每个顶点对应纹理的一个角，这样就创建了一个非常简单的映射，每个顶点的纹理坐标为(0,0)或(0,1)或(1,0)或(1,1)。

// 创建顶点缓冲

SimpleVertex vertices[] =

{

 { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

};

当采样纹理时，我们需要使用材质颜色对纹理颜色进行调制。

### 将纹理绑定为Shader资源

纹理和采样状态是类似于常量缓冲的对象，在它们可以被shader使用前，需要使用**ID3D11DeviceContext::PSSetSamplers()**和**ID3D11DeviceContext::PSSetShaderResources()**方法进行设置。

g\_pImmediateContext->PSSetShaderResources( 0, 1, &g\_pTextureRV );

g\_pImmediateContext->PSSetSamplers( 0, 1, &g\_pSamplerLinear );

现在我们就做好了在shader中使用纹理的准备了。

### 施加纹理(fx)

要将纹理映射到几何体上，我们需要在像素着色器中调用一个纹理查询方法。这个叫做**Sample**的方法会执行2D纹理的查询，然后返回一个采样过的颜色。像素着色器调用这个方法并将返回的结果颜色乘以网格的颜色(或材质颜色)，最后输出最终的颜色。

* txDiffuse用于储存从传入的纹理，这个纹理绑定在资源视图g\_pTextureRV中。
* samLinear在下面介绍，它是用于纹理查询的采样器的详细说明。
* input.Tex是纹理的坐标。

// Pixel Shader

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

{

 return txDiffuse.Sample( samLinear, input.Tex ) \* vMeshColor;

}

别忘了通过顶点着色器将纹理坐标传递过来，如果你忘了此步，在像素着色器中就会丢失数据。本例中，我们只是将输入的坐标直接输出，硬件会处理余下的工作。

// Vertex Shader

PS\_INPUT VS( VS\_INPUT input )

{

 PS\_INPUT output = (PS\_INPUT)0;

 output.Pos = mul( input.Pos, World );

 output.Pos = mul( output.Pos, View );

 output.Pos = mul( output.Pos, Projection );

 output.Tex = input.Tex;

 return output;

}

## 常量缓存（Constant buffer）

在Direct3D 11中，一个应用程序可以使用常量缓存设置shader常量(shader变量)。声明常量缓存的方法类似于C语言的结构体。常量缓存将shader常量组合在一起并同时提交，与对每个常量单独调用相比，这样做减少了更新时所需的带宽。

上一个教程中我们将所有shader常量放置在一个常量缓冲中。但更有效率地使用常量缓存的方法是将shader变量基于它们的更新频率创建在常量缓存中。这样做可以让应用程序将更新shader常量所需的带宽减到最小。例如，本教程中将常量分为三组：一个用于每帧都改变的变量，一组用于只在窗口大小改变时才改变的变量，一组只设置一次然后就不再改变。

下面的常量缓存是定义在.fx文件中的。

cbuffer cbNeverChanges

{

 matrix View;

};

cbuffer cbChangeOnResize

{

 matrix Projection;

};

cbuffer cbChangesEveryFrame

{

 matrix World;

 float4 vMeshColor;

};

要使用这些常量缓冲，需要为每个缓冲创建一个ID3D11Effect对象。然后调用**ID3D11DeviceContext::UpdateSubresource()**方法更新对应的常量缓冲。

//

// Update variables that change once per frame

//

CBChangesEveryFrame cb;

cb.mWorld = XMMatrixTranspose( g\_World );

cb.vMeshColor = g\_vMeshColor;

g\_pImmediateContext->UpdateSubresource( g\_pCBChangesEveryFrame, 0, NULL, &cb, 0, 0 );

## 完整代码

#include <windows.h>

#include <d3d11.h>

#include <directxmath.h>

#include "DDSTextureLoader.h"

#include "resource.h"

#include <d3dcompiler.h>

using namespace DirectX;

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

// Structures

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

struct SimpleVertex

{

 XMFLOAT3 Pos;

 XMFLOAT2 Tex;

};

struct CBNeverChanges

{

 XMMATRIX mView;

};

struct CBChangeOnResize

{

 XMMATRIX mProjection;

};

struct CBChangesEveryFrame

{

 XMMATRIX mWorld;

 XMFLOAT4 vMeshColor;

};

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

// Global Variables

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

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;

ID3D11InputLayout\* g\_pVertexLayout = NULL;

ID3D11Buffer\* g\_pVertexBuffer = NULL;

ID3D11Buffer\* g\_pIndexBuffer = NULL;

ID3D11Buffer\* g\_pCBNeverChanges = NULL;

ID3D11Buffer\* g\_pCBChangeOnResize = NULL;

ID3D11Buffer\* g\_pCBChangesEveryFrame = NULL;

ID3D11ShaderResourceView\* g\_pTextureRV = NULL;

ID3D11SamplerState\* g\_pSamplerLinear = NULL;

XMMATRIX g\_World;

XMMATRIX g\_View;

XMMATRIX g\_Projection;

XMFLOAT4 g\_vMeshColor( 0.7f, 0.7f, 0.7f, 1.0f );

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

// Forward declarations

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

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;

}

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

// Register class and create window

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

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 7", 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;

}

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

// Create Direct3D device and swap chain

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

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"Tutorial07.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 },

 { "TEXCOORD", 0, DXGI\_FORMAT\_R32G32\_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"Tutorial07.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;

 // Create vertex buffer

 SimpleVertex vertices[] =

 {

 { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT2( 1.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT2( 0.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT2( 0.0f, 0.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT2( 1.0f, 0.0f ) },

 };

 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

 // Create vertex 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;

 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 buffers

 bd.Usage = D3D11\_USAGE\_DEFAULT;

 bd.ByteWidth = sizeof(CBNeverChanges);

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

 bd.CPUAccessFlags = 0;

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

 if( FAILED( hr ) )

 return hr;

 bd.ByteWidth = sizeof(CBChangeOnResize);

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

 if( FAILED( hr ) )

 return hr;

 bd.ByteWidth = sizeof(CBChangesEveryFrame);

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

 if( FAILED( hr ) )

 return hr;

 // Load the Texture

 hr = CreateDDSTextureFromFile( g\_pd3dDevice, L"seafloor.dds", NULL, &g\_pTextureRV );

 if( FAILED( hr ) )

 return hr;

 // Create the sample state

 D3D11\_SAMPLER\_DESC sampDesc;

 ZeroMemory( &sampDesc, sizeof(sampDesc) );

 sampDesc.Filter = D3D11\_FILTER\_MIN\_MAG\_MIP\_LINEAR;

 sampDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_WRAP;

 sampDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_WRAP;

 sampDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_WRAP;

 sampDesc.ComparisonFunc = D3D11\_COMPARISON\_NEVER;

 sampDesc.MinLOD = 0;

 sampDesc.MaxLOD = D3D11\_FLOAT32\_MAX;

 hr = g\_pd3dDevice->CreateSamplerState( &sampDesc, &g\_pSamplerLinear );

 if( FAILED( hr ) )

 return hr;

 // Initialize the world matrices

 g\_World = XMMatrixIdentity();

 // Initialize the view matrix

 XMVECTOR Eye = XMVectorSet( 0.0f, 3.0f, -6.0f, 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 );

 CBNeverChanges cbNeverChanges;

 cbNeverChanges.mView = XMMatrixTranspose( g\_View );

 g\_pImmediateContext->UpdateSubresource( g\_pCBNeverChanges, 0, NULL, &cbNeverChanges, 0, 0 );

 // Initialize the projection matrix

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

 CBChangeOnResize cbChangesOnResize;

 cbChangesOnResize.mProjection = XMMatrixTranspose( g\_Projection );

 g\_pImmediateContext->UpdateSubresource( g\_pCBChangeOnResize, 0, NULL, &cbChangesOnResize, 0, 0 );

 return S\_OK;

}

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

// Clean up the objects we've created

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

void CleanupDevice()

{

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

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

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

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

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

 if( g\_pCBChangesEveryFrame ) g\_pCBChangesEveryFrame->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\_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 );

 // Modify the color

 g\_vMeshColor.x = ( sinf( t \* 1.0f ) + 1.0f ) \* 0.5f;

 g\_vMeshColor.y = ( cosf( t \* 3.0f ) + 1.0f ) \* 0.5f;

 g\_vMeshColor.z = ( sinf( t \* 5.0f ) + 1.0f ) \* 0.5f;

 //

 // 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 );

 //

 // Update variables that change once per frame

 //

 CBChangesEveryFrame cb;

 cb.mWorld = XMMatrixTranspose( g\_World );

 cb.vMeshColor = g\_vMeshColor;

 g\_pImmediateContext->UpdateSubresource( g\_pCBChangesEveryFrame, 0, NULL, &cb, 0, 0 );

 //

 // Render the cube

 //

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

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

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

 g\_pImmediateContext->VSSetConstantBuffers( 2, 1, &g\_pCBChangesEveryFrame );

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

 g\_pImmediateContext->PSSetConstantBuffers( 2, 1, &g\_pCBChangesEveryFrame );

 g\_pImmediateContext->PSSetShaderResources( 0, 1, &g\_pTextureRV );

 g\_pImmediateContext->PSSetSamplers( 0, 1, &g\_pSamplerLinear );

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

 //

 // Present our back buffer to our front buffer

 //

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

}