# DirectX SDK 教程2：绘制一个三角形

原文地址：<http://msdn.microsoft.com/en-us/library/ff729719>。

## 概览

在前面的教程中，我们创建了一个最小化的Direct3D 11程序在平面上输出了单一的颜色。本教程中，我们会在屏幕上绘制一个三角形。我们会学习设置三角形数据结构的方法。

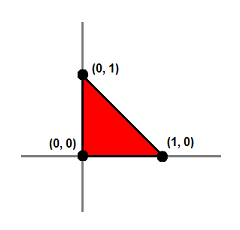


## 源代码

(SDK root)\Samples\C++\Direct3D11\Tutorials\Tutorial02

## 三角形的构成

三角形由三个点定义，这些点又叫**顶点（vertex）**。确定位置的三个顶点定义了一个唯一的三角形。要让GPU绘制一个三角形，我们必须告知三个顶点的位置。例如我们想绘制一个如图1所示的三角形，我们需要将顶点位置(0, 0)、(0, 1)和(1, 0)传递到GPU，这样GPU就拥有足够的信息绘制这个三角形了。



**图1. 一个三角形由三个顶点定义**

那么如何将顶点信息传递到GPU中？在Direct3D 11中，诸如位置之类的顶点信息是存储在一个缓存资源中的。存储顶点信息的缓存叫做**顶点缓存（vertex buffer）**。我们必须创建一个足够包含三个顶点大小的顶点缓存，并将顶点位置存储其中。在Direct3D 11中，程序必须以字节为单位设置缓存大小。我们知道缓存必须大到足够包含三个顶点，那么每个顶点需要多少字节呢？要回答这个问题，需要理解**顶点格式（vertex layout）**。

## 输入格式（Input Layout）

一个顶点具有位置信息，但往往不只包含位置信息，还可能包含法线、一个或以上的颜色、纹理坐标（用于纹理映射）等信息。顶点格式定义了这些信息属性在内存中时如何放置的：每个属性使用何种数据类型，每个属性的大小，在内存中的顺序。因为属性通常使用不同的类型，类似于C结构中的字段，所以顶点通常使用结构来表示，顶点的大小可以方便地从结构的大小获取。

本教程中，我们只使用顶点的位置信息。所以，我们定义的顶点结构中只使用一个XMFLOAT3类型的字段。这个类型表示一个由三个浮点数作为分量的矢量，这是3D中表示位置的典型数据结构。

struct SimpleVertex

{

XMFLOAT3 Pos; // Position

};

现在我们有了表示顶点的结构体，这个结构负责将顶点信息储存到系统内存中。但是，当我们将顶点缓存传递到GPU的时候，我们传递的只是一块内存中的信息。GPU必须知道顶点格式才能从缓存中提取正确的属性，要实现这个目的，需要使用一个**输入格式（input layout）**。

在Direct3D 11中，输入结构是一个Direct3D对象，这个对象表示可以被GPU理解的顶点结构。每个顶点属性可以由D3D11\_INPUT\_ELEMENT\_DESC结构表示。应用程序定义一个数组，这个数组包含一个或多个D3D11\_INPUT\_ELEMENT\_DESC，然后使用这个数组创建输入格式对象，这个对象表示以整体的形式表示顶点。下面我们看一下D3D11\_INPUT\_ELEMENT\_DESC 字段的细节。

|  |  |
| --- | --- |
| SemanticName | Semantic（语义）名称是一个字符串，包含一个表示这个元素特性或目的（或语义）单词。这个单词可以是任何一个C标示符，我们可以任意选择。例如，表示顶点位置的语义名称POSITION就很好。语义名称不是大小写敏感的。 |
| SemanticIndex | Semantic索引补充说明语义名称。一个顶点的统一特性可能有多个属性。例如，一个顶点可能有2组纹理坐标和2组颜色。你可以使用语义名称加数字的形式，例如“COLOR0”和“COLOR1”，你也可以使用语义索引0和1代替，这样两个元素可以共享相同的语义名称“COLOR”。 |
| Format | Format定义了用于这个元素的数据类型。例如，DXGI\_FORMAT\_R32G32B32\_FLOAT格式有3个32位浮点数，让元素的长度为12个字节。DXGI\_FORMAT\_R16G16B16A16\_UINT格式有4个16位无符号整数，元素长度为8个字节。 |
| InputSlot | 前面已经提到过，一个Direct3D 11程序通过使用顶点缓存将顶点数据传送到GPU中。在Direct3D 11中，多个顶点缓存的数据可以同时传递到GPU，最多可以达到16个。每个顶点缓存绑定一个input slot 数字，范围从0到15。InputSlot字段告诉GPU从哪个顶点缓存中获取这个元素。 |
| AlignedByteOffset | 顶点是存储在顶点缓存中的，只是一块内存。字段AlignedByteOffset告知GPU从内存中哪个位置开始获取数据。 |
| InputSlotClass | 这个字段通常只包含D3D11\_INPUT\_PER\_VERTEX\_DATA。当程序使用instancing时，可以将输入结构的InputSlotClass设置为D3D11\_INPUT\_PER\_INSTANCE\_DATA处理包含instance数据的顶点缓存。Instancing是一个高级的Direct3D内容，不会在这里讨论。在本教程中，我们只使用D3D11\_INPUT\_PER\_VERTEX\_DATA。 |
| InstanceDataStepRate | 这个字段用于instancing。因为我们不使用instancing，所以这个字段必须设置为0表示不使用。 |

现在我们可以定义D3D11\_INPUT\_ELEMENT\_DESC数组并创建input layout了：

// 定义input layout

D3D11\_INPUT\_ELEMENT\_DESC layout[] =

{

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

};

UINT numElements = ARRAYSIZE (layout);

### 顶点格式（Vertex Layout）

在下一个教程中，我们会解释technique对象和其对应的shader。但现在，我们只需关注创建Direct3D 11顶点格式对象。但是，我们将要学习的technique和shader与顶点格式是紧密配合的。理由是：创建一个顶点格式对象需要顶点着色器的**输入签名（input signature）**。我们首先调用technique的GetPassByIndex()方法获取一个effect pass对象，这个对象代表technique的第一个第一个pass。然后，我们调用这个pass对象的GetDesc()方法获取pass的描述结构，在这个结构中有一个叫做pIAInputSignature的字段指向代表顶点着色器的输入签名的二进制数据。一旦我们获取了这个数据，就可以调用ID3D11Device::CreateInputLayout()创建一个顶点格式对象，调用ID3D11DeviceContext::IASetInputLayout()将它设置为激活的顶点格式。代码如下：

// 创建input layout

if( FAILED( g\_pd3dDevice->CreateInputLayout( layout, numElements, pVSBlob->GetBufferPointer(),

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

return FALSE;

// Set the input layout

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

### 创建顶点缓存

在初始化过程中还需要做的另一件事是创建顶点缓存保存顶点数据。要在Direct3D 11中创建顶点，我们需要两个结构，D3D11\_BUFFER\_DESC和D3D11\_SUBRESOURCE\_DATA，然后调用ID3D11Device::CreateBuffer()方法。D3D11\_BUFFER\_DESC表示要创建的顶点缓存对象，D3D11\_SUBRESOURCE\_DATA表示在创建过程中复制到顶点缓存中的实际数据。顶点缓存的创建和初始化只需进行一次，以后就无需再次初始化了。复制到顶点缓存中的数据是顶点，即一个拥有3个SimpleVertex结构的数组。创建了顶点缓存之后，我们可以调用ID3D11DeviceContext::IASetVertexBuffers()方法将它绑定到设备。完整代码如下：

// 创建顶点缓存

SimpleVertex vertices[] =

{

XMFLOAT3( 0.0f, 0.5f, 0.5f ),

XMFLOAT3( 0.5f, -0.5f, 0.5f ),

XMFLOAT3( -0.5f, -0.5f, 0.5f ),

};

D3D11\_BUFFER\_DESC bd;

ZeroMemory( &bd, sizeof(bd) );

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof( SimpleVertex ) \* 3;

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

bd.CPUAccessFlags = 0;

bd.MiscFlags = 0;

D3D11\_SUBRESOURCE\_DATA InitData;

ZeroMemory( &InitData, sizeof(InitData) );

InitData.pSysMem = vertices;

if( FAILED( g\_pd3dDevice->CreateBuffer( &bd, &InitData, &g\_pVertexBuffer ) ) )

return FALSE;

// Set vertex buffer

UINT stride = sizeof( SimpleVertex );

UINT offset = 0;

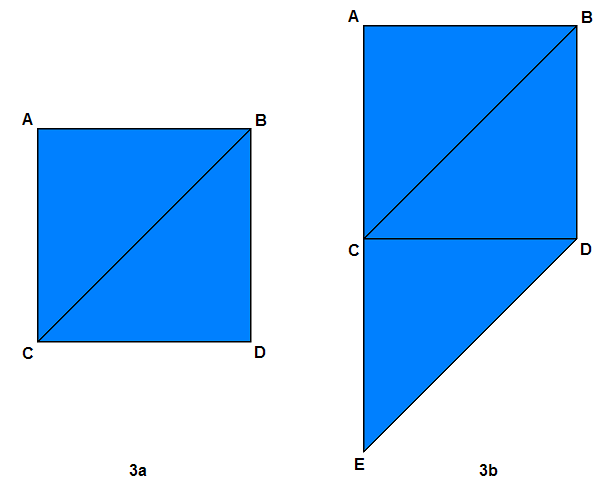
g\_pImmediateContext->IASetVertexBuffers( 0, 1, &g\_pVertexBuffer, &stride, &offset );

### 图元拓扑结构

图元拓扑结构（Primitive topology）表示GPU如何使用三个顶点绘制一个三角形。要绘制一个单独的三角形，需要发送三个顶点给GPU，所以，顶点缓存中有三个顶点。但如果我们要绘制两个三角形呢？一个方法是发送六个顶点，前3个顶点绘制第一个三角形，后3个顶点绘制第二个三角形。这种拓扑结构成为三角形列表（triangle list），三角形列表容易理解，但如果要绘制的三角形共享顶点，往往效率不高。例如，图3a显示了一个由两个三角形组成的正方形：A B C和C B D。（习惯上，三角形是按顺时针方向的顶点顺序定义的。）如果我们使用三角形列表向GPU发送这两个三角形，顶点缓存如下列形式：

A B C C B D

你可以注意到B和C在顶点缓存中出现了两次，这是因为它们是被两个三角形共享的。



**图3a是一个由两个三角形组成的正方形；图3b是一个由三个三角形组成的梯形**

但如果在绘制第二个三角形时，我们不是从顶点缓存中获取三个顶点，而是使用前一个三角形的2个顶点从顶点缓存中只获取一个顶点，就可以减小顶点缓存的大小。Direct3D支持这个方法，成为三角形带（triangle strip）。当绘制三角形带时，第一个三角形是由顶点缓存中的前三个顶点定义的，下一个三角形由前一个三角形的最后两个顶点加顶点缓存中的下一个顶点定义。以图3a中的正方形为例，使用三角形带，顶点缓存的结构如下：

A B C D

前三个顶点，A B C，定义了第一个三角形。第二个三角形由B和C，第一个三角形的最后两个顶点，加上D定义。通过使用三角形带，顶点缓存的大小由6个顶点变为4个。同理，图3b中的三个三角形，使用三角形列表时顶点缓存的结构如下：

A B C C B D C D E

使用三角形带时顶点缓存的大小显著减小：

A B C D E

在三角形带的例子中，第二个三角形定义为B C D，这三个顶点并不是顺时针方向，这时使用三角形带的正常情况。为了正常显示，GPU会自动交换前一个三角形的后两个顶点的顺序。这种交换只处理第2、4、6、8……个三角形，这样就能保证每个三角形都已正确的环绕方向定义（这种情况中时顺时针方向）。除了三角形列表和三角形带，Direct3D 10 还支持其他类型的图元拓扑结构，这个示例中我们不会讨论它们。

在本教程的代码中，我们只绘制一个三角形，所以采用哪种形式并无区别，但必须设置一个，所以使用的是三角形列表。

// 设置图元拓扑结构

g\_pImmediateContext->IASetPrimitiveTopology( D3D10\_PRIMITIVE\_TOPOLOGY\_TRIANGLELIST );

## 绘制三角形

最后是实际绘制三角形的代码。我们已经创建了顶点着色器和像素着色器，顶点着色器用来对顶点位置进行变换，像素着色器负责计算三角形上每个像素的颜色，具体细节会在下一个教程中介绍。要使用这两个shader，我们必须调用ID3D11DeviceContext::VSSetShader() and ID3D11DeviceContext::PSSetShader()。最后需要调用ID3D11DeviceContext::Draw()，它命令GPU使用当前的顶点缓存、顶点布局和图元拓扑进行绘制。Draw()方法的第一个参数是发送到GPU的顶点数量，第二个参数是第一个顶点的索引，因为我们绘制的是一个三角形并且从第一个顶点开始绘制，所以这里是3和0，完整代码如下：

// 绘制一个三角形

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

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

g\_pImmediateContext->Draw( 3, 0 );

## 完整代码

//

// Copyright (c) Microsoft Corporation. All rights reserved.

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

#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;

};

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

// 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;

ID3D11VertexShader\* g\_pVertexShader = NULL;

ID3D11PixelShader\* g\_pPixelShader = NULL;

ID3D11InputLayout\* g\_pVertexLayout = NULL;

ID3D11Buffer\* g\_pVertexBuffer = NULL;

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

// 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;

}

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

// 注册窗口类并创建窗口

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

HRESULT InitWindow( HINSTANCE hInstance, int nCmdShow )

{

// 注册窗口类

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;

// 创建窗口

g\_hInst = hInstance;

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

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

g\_hWnd = CreateWindow( L"TutorialWindowClass", L"Direct3D 11 Tutorial 3: Shaders",

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;

}

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

// 使用D3DCompile编译shader的辅助方法

//

// 如果使用的是VS 11，可以不使用这种方法，而是加载预生成的.cso文件代替...

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

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;

// 创建渲染目标视图

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;

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

// 设置视口

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 );

// 编译顶点着色器

ID3DBlob\* pVSBlob = NULL;

hr = CompileShaderFromFile( L"Tutorial03.fx", "VS", "vs\_5\_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;

}

// 创建顶点着色器

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

if( FAILED( hr ) )

{

pVSBlob->Release();

return hr;

}

// 定义输入布局描述

D3D11\_INPUT\_ELEMENT\_DESC layout[] =

{

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

};

UINT numElements = ARRAYSIZE( layout );

// 创建输入布局

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

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

pVSBlob->Release();

if( FAILED( hr ) )

return hr;

// 设置输入布局

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

// 编译像素着色器

ID3DBlob\* pPSBlob = NULL;

hr = CompileShaderFromFile( L"Tutorial03.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;

}

// 创建像素着色器

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

pPSBlob->Release();

if( FAILED( hr ) )

return hr;

// 创建顶点缓冲

SimpleVertex vertices[] =

{

XMFLOAT3( 0.0f, 0.5f, 0.5f ),

XMFLOAT3( 0.5f, -0.5f, 0.5f ),

XMFLOAT3( -0.5f, -0.5f, 0.5f ),

};

D3D11\_BUFFER\_DESC bd;

ZeroMemory( &bd, sizeof(bd) );

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof( SimpleVertex ) \* 3;

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;

// 设置顶点缓冲

UINT stride = sizeof( SimpleVertex );

UINT offset = 0;

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

// 设置图元拓扑

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

return S\_OK;

}

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

// 清理创建的对象

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

void CleanupDevice()

{

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

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

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

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

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

}

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

// 绘制帧

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

void Render()

{

// 清除后台缓冲

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

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

// 绘制一个三角形

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

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

g\_pImmediateContext->Draw( 3, 0 );

// 将后台缓冲中的内容呈现在前台缓冲（屏幕）中

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

}