# Direct3D 11教程5：3D变换

原文地址：<http://msdn.microsoft.com/en-us/library/ff729722.aspx>。

## 概览

在前面的教程中，我们将一个立方体从模型空间变换到了屏幕空间并进行了绘制。在本教程中，我们将拓展变换的概念，展示一个使用这些变换实现的简单动画。

这个教程的输出结果是一个物体绕着另一个物体旋转，用来展示变换以及如何组合这些变换达到所需的效果。后面的教程会建立在这个教程之上介绍新的概念。



## 源代码

(SDK root)\Samples\C++\Direct3D11\Tutorials\Tutorial05

## 变换

在3D图形学中，变换常用在顶点和矢量上，变换还用于将它们从一个空间变换到另一个。变换的操作是通过矩阵乘法实现的。典型的有三种基本变换：平移，旋转和缩放。此外，还有投影变换用于从视空间变换到投影空间。XNA数学库包含许多可以便利地构建矩阵的API，这些矩阵可以用于平移、旋转、缩放，世界-视空间变换，视-投影空间变换等。应用程序可以使用这些矩阵对顶点进行变换。对矩阵有个基本的理解是必须的，下面我们来看几个例子。

## 平移

平移变换表示在空间中移动一段距离。在3D中，用于平移的矩阵有以下的形式：



其中矢量(a, b, c)定义了移动的方向和大小。例如，要将一个顶点沿X轴移动-5个单位（负X轴方向），我们可以将这个顶点乘以以下矩阵：



如果我们将它施加在一个位于原点的立方体上，结果就是盒子沿负X轴移动5个单位，如图1所示。



**图1. 平移效果**

在3D中，空间通常是由一个原点和从原点出发的三条轴定义的。在计算机图形学中通常使用好几个空间：对象空间，世界空间，视空间，投影空间和屏幕空间。



**图2. 定义在对象空间中的立方体**

## 旋转

旋转表示绕一条过原点的轴转动顶点，这些轴就是X，Y和Z轴。2D中的一个例子就是将矢量[1 0]逆时针旋转90度，结果是矢量[0 1]。下面的矩阵用来绕Y轴顺时针旋转θ度：



图3显示了将一个立方体绕Y轴旋转45度的效果。



**图3. 绕Y轴旋转的效果**

## 缩放

缩放表示沿轴方向放大或缩小矢量的分量大小。例如，一个矢量可以沿所有方向放大，也可以单独沿X轴方向缩小。通常使用的缩放矩阵如下：



其中p，q和r表示沿X，Y和Z方向的缩放因子。下图表示沿X轴放大2倍，沿Y轴缩小0.5倍的效果。



**图4. 缩放效果**

## 多重变换

要在一个矢量上施加多重变换，只需简单地首先乘以第一个变换矩阵，然后将结果乘以第二个变换矩阵，依此类推。因为矢量和矩阵乘法符合结合律，我们可以首先将所有矩阵相乘，然后用矢量乘以结果矩阵也能获得相同的结果。下图展示了在一个立方体上组合旋转和平移变换的效果。



**图5. 旋转和平移的效果**

## 创建环绕轨道

本教程中，我们会对两个立方体进行变换。第一个立方体会在原位置旋转，而第二个立方体绕第一个旋转，同时绕自身轴旋转。这两个立方体都有各自的世界变换矩阵，这些变换在每一帧都会重新施加。

XNA库中有一些方法可以帮助你创建旋转、平移和缩放矩阵。

* 绕X、Y和Z的旋转可以使用**XMMatrixRotationX**，**XMMatrixRotationY**和**XMMatrixRotationZ**方法实现，它们创建了绕轴旋转的矩阵。绕其他轴的复杂旋转可以通过以上基本旋转的乘积实现。
* 平移使用的是**XMMatrixTranslation**方法，这个方法可以创建一个平移矩阵，移动的信息保存在传递的参数中。
* 缩放使用的是**XMMatrixScaling**方法。这个方法只会沿着3根基本轴缩放，如果想要沿任意轴缩放，那么需要乘以一个正确的旋转矩阵才能达到效果。

第一个立方体原地旋转，作为圆轨道的中心。这个立方体有一个绕Y轴的旋转，这是通过调用**XMMatrixRotationY**方法实现，。它每帧旋转一定的数量。因为需要立方体持续地旋转，旋转矩阵的参数是递增的。

// 1st Cube: Rotate around the origin

g\_World = XMMatrixRotationY(t );

第二个立方体绕第一个旋转。为了介绍多重变换，添加了一个缩放因子和绕自身轴的旋转。公式就在代码的下面(在注释中)。首先这个立方体缩小到30%大小，然后绕自身轴旋转(本例中为Z轴)。要模拟轨道的运动，需要先平移出原点然后绕Y轴旋转。要达到效果，可以使用4个独立变换的矩阵（mScale，mSpin，mTranslate，mOrbit），然后将它们相乘。

// 第2个立方体：绕原点旋转

XMMATRIX mSpin = XMMatrixRotationZ( -t );

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

XMMATRIX mTranslate = XMMatrixTranslation( -4.0f, 0.0f, 0.0f );

XMMATRIX mScale = XMMatrixScaling( 0.3f, 0.3f, 0.3f );

g\_World2 = mScale \* mSpin \* mTranslate \* mOrbit;

以上操作不满足交换律，变换的顺序非常重要。你可以改变一下变换的顺序观察导致的结果。

所有变换方法都会从参数中创建一个矩阵，旋转的值是递增的，这可以通过更新“time”变量做到。

// Update our time

t += XM\_PI \* 0.0125f;

在进行绘制前，technique需要为shader获取变量。本例中，世界矩阵、视矩阵和投影矩阵会被绑定到technique。注意两个立方体的世界矩阵不同，需要独立设置。

//

// 更新第一个立方体的变量

//

ConstantBuffer cb1;

cb1.mWorld = XMMatrixTranspose( g\_World1 );

cb1.mView = XMMatrixTranspose( g\_View );

cb1.mProjection = XMMatrixTranspose( g\_Projection );

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

//

// 绘制第一个立方体

//

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

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

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

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

//

// 更新第二个立方体的变量

//

ConstantBuffer cb2;

cb2.mWorld = XMMatrixTranspose( g\_World2 );

cb2.mView = XMMatrixTranspose( g\_View );

cb2.mProjection = XMMatrixTranspose( g\_Projection );

g\_pImmediateContext->UpdateSubresource( g\_pConstantBuffer, 0, NULL, &cb2, 0, 0 );

//

// 绘制第二个立方体

//

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

## 深度缓存

本教程还有一个重要的补充是深度缓存。没有它，当小物体绕到大物体背后时，仍会绘制在大物体前面。深度缓存让Direct3D保存了平面上每个像素的深度。在Direct3D 11中，深度缓存的默认行为是将绘制在屏幕上的每个像素的深度与存储在深度缓冲中的深度进行比较。如果正在进行绘制的像素的深度小于等于存储在深度缓存中的，则绘制这个像素，并将深度缓存中的深度值更新为新绘制的这个像素的深度值。相反，如果正在绘制的像素的深度值大于深度缓存中的，则该像素就不进行绘制，深度缓存中的值保持不变。

下面的代码创建了一个深度缓存(一个DepthStencil纹理)。还创建了一个深度缓存的DepthStencilView，这样Direct3D 11就知道将它作为一个Depth Stencil纹理。

// 创建深度缓冲纹理

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;

// 创建深度模板视图

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;

为了使用这个新创建的深度模版缓存，必须将它绑定到设备。这是通过将深度缓存视图作为**OMSetRenderTargets**方法的第三个参数实现的。

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

因为处理的是渲染目标，我们还需要在绘制前清除深度缓存。这可以确保前一帧的深度值不会错误地抛弃当前帧的像素。下面的代码将深度缓存全部设置为最大值1.0。

//

// 将深度缓冲重置为1.0 (最大深度值)

//

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

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

// Structures

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

struct SimpleVertex

{

 XMFLOAT3 Pos;

 XMFLOAT4 Color;

};

struct ConstantBuffer

{

 XMMATRIX mWorld;

 XMMATRIX mView;

 XMMATRIX mProjection;

};

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

// 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\_pConstantBuffer = NULL;

XMMATRIX g\_World1;

XMMATRIX g\_World2;

XMMATRIX g\_View;

XMMATRIX g\_Projection;

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

// 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 5", 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;

 // 创建深度缓冲纹理

 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;

 // 创建深度模板视图

 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"Tutorial05.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 },

 { "COLOR", 0, DXGI\_FORMAT\_R32G32B32A32\_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"Tutorial05.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;

 // 创建顶点缓冲Create vertex buffer

 SimpleVertex vertices[] =

 {

 { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT4( 0.0f, 0.0f, 1.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT4( 0.0f, 1.0f, 0.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT4( 0.0f, 1.0f, 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT4( 1.0f, 0.0f, 0.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, -1.0f ), XMFLOAT4( 1.0f, 0.0f, 1.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, -1.0f ), XMFLOAT4( 1.0f, 1.0f, 0.0f, 1.0f ) },

 { XMFLOAT3( 1.0f, -1.0f, 1.0f ), XMFLOAT4( 1.0f, 1.0f, 1.0f, 1.0f ) },

 { XMFLOAT3( -1.0f, -1.0f, 1.0f ), XMFLOAT4( 0.0f, 0.0f, 0.0f, 1.0f ) },

 };

 D3D11\_BUFFER\_DESC bd;

 ZeroMemory( &bd, sizeof(bd) );

 bd.Usage = D3D11\_USAGE\_DEFAULT;

 bd.ByteWidth = sizeof( SimpleVertex ) \* 8;

 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;

 // 绑定顶点缓冲

 UINT stride = sizeof( SimpleVertex );

 UINT offset = 0;

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

 // 创建索引缓冲

 WORD indices[] =

 {

 3,1,0,

 2,1,3,

 0,5,4,

 1,5,0,

 3,4,7,

 0,4,3,

 1,6,5,

 2,6,1,

 2,7,6,

 3,7,2,

 6,4,5,

 7,4,6,

 };

 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;

 // 绑定索引缓冲

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

 // 设置图元拓扑

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

 // 创建常量缓冲

 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;

 // 初始化世界矩阵

 g\_World1 = XMMatrixIdentity();

 g\_World2 = XMMatrixIdentity();

 // 初始化视矩阵

 XMVECTOR Eye = XMVectorSet( 0.0f, 1.0f, -5.0f, 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 );

 // 初始化投影矩阵

 g\_Projection = XMMatrixPerspectiveFovLH( XM\_PIDIV2, 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\_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()

{

 // 更新时间

 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;

 }

 // 1st Cube: Rotate around the origin

 g\_World1 = XMMatrixRotationY( t );

 // 第2个立方体：绕原点旋转

 XMMATRIX mSpin = XMMatrixRotationZ( -t );

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

 XMMATRIX mTranslate = XMMatrixTranslation( -4.0f, 0.0f, 0.0f );

 XMMATRIX mScale = XMMatrixScaling( 0.3f, 0.3f, 0.3f );

 g\_World2 = mScale \* mSpin \* mTranslate \* mOrbit;

 //

 // 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 );

 //

 // 将深度缓冲重置为1.0 (最大深度值)

 //

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

 //

 // 更新第一个立方体的变量

 //

 ConstantBuffer cb1;

 cb1.mWorld = XMMatrixTranspose( g\_World1 );

 cb1.mView = XMMatrixTranspose( g\_View );

 cb1.mProjection = XMMatrixTranspose( g\_Projection );

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

 //

 // 绘制第一个立方体

 //

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

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

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

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

 //

 // 更新第二个立方体的变量

 //

 ConstantBuffer cb2;

 cb2.mWorld = XMMatrixTranspose( g\_World2 );

 cb2.mView = XMMatrixTranspose( g\_View );

 cb2.mProjection = XMMatrixTranspose( g\_Projection );

 g\_pImmediateContext->UpdateSubresource( g\_pConstantBuffer, 0, NULL, &cb2, 0, 0 );

 //

 // 绘制第二个立方体

 //

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

 //

 // Present our back buffer to our front buffer

 //

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

}