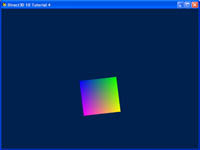
# Direct3D 11教程4：3D空间

原文地址：<http://msdn.microsoft.com/en-us/library/ff729721.aspx>。

## 概览

在前面的教程中，我们在窗口中央绘制了一个三角形，我们着重讲解如何从顶点缓存中的获取顶点位置。而在本教程中我们将讨论3D位置和变换的细节。

本教程会在屏幕上绘制一个3D物体，上一个教程我们关注如何在屏幕上绘制一个2D对象，现在我们要显示一个3D物体。



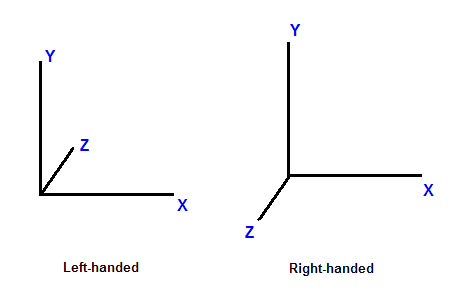
## 源代码

(SDK root)\Samples\C++\Direct3D11\Tutorials\Tutorial04

## 3D空间

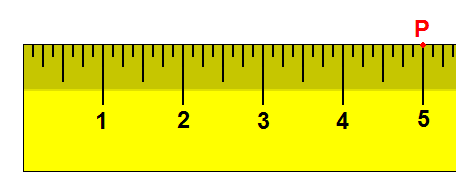
在前面的教程中，三角形的顶点坐标数据就是它的屏幕坐标，但是，这种情况并不常见。我们需要一个系统用来表示在3D空间中的物体并显示这些物体。

在真实世界中，物体是处在3D空间中的。这意味着要将一个物体放置在世界的指定位置，我们需要使用一个坐标系统并定义位置对应的坐标。在计算机图形学中，3D空间通常使用笛卡尔坐标系。在这个坐标系中，三根轴X，Y和Z互相垂直，定义了空间中的每个点。这个坐标系统分为左手系和右手系。在左手系中，X轴指向右方，Y轴指向上方，Z轴指向前方。在右手系中，X轴和Y轴的指向相同，但Z轴指向后方。



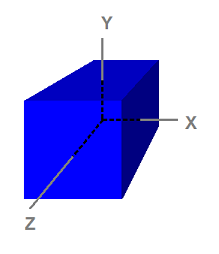
**图1. 左手坐标系和右手坐标系**

下面讨论一下坐标系统。在不同的空间一个点具有不同的坐标。以一维为例，假设我们有一把尺，P点在5英寸处。如果将尺向右移动1英寸，则P点的坐标变为4。通过移动尺，参照系发生了变化，结果是该点有了一个新的坐标。



**图2. 1D空间**

在3D中，一个空间是由原点和三个从原点出发的轴定义的。在计算机图形学中通常使用下列空间：对象空间（**object space**），世界空间（**world space**），视空间（**view space**），投影空间（**projection space**）和屏幕空间（**screen space**）。



**图3. 定义在对象空间中的立方体**

### 对象空间（Object Space）

注意立方体的中心在原点。对象空间又称为模型空间（**model space**），表示创建3D模型的艺术家使用的空间。通常，艺术家在创建模型时将它的中点放置在原点上，这样做可以很容易地进行诸如旋转之类的变换操作，在下一个教程中我们会讨论到变换操作。立方体的8个顶点坐标如下：

(-1, 1, -1)

( 1, 1, -1)

(-1, -1, -1)

( 1, -1, -1)

(-1, 1, 1)

( 1, 1, 1)

(-1, -1, 1)

( 1, -1, 1)

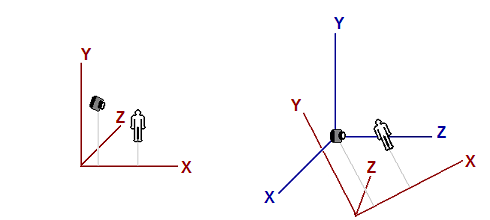
因为对象空间是艺术家在设计和创建模型时使用的，所以存储在磁盘上的模型也处于对象空间。应用程序可以创建一个顶点缓存代表这个模型并使用模型数据进行缓存的初始化。因此，顶点缓存中的顶点通常也是处于对象空间中的，这样意味着顶点着色器接收的是对象空间中的顶点数据。

### 世界空间（World Space）

世界空间被场景中的所有对象共享，它用来定义对象的空间联系。要理解世界空间，你可以想象自己站在一个长方体房间的西南角面朝北方。我们定义自己的脚站在原点(0, 0, 0)。X轴在我们右方，Y轴向上，Z轴向前，即我们面朝的方向。这样，房间中的每个位置都可以定义为一组XYZ坐标。例如，有一个椅子在我们前方5英尺，右方2英尺，椅子上方8英尺高处的天花板上有一盏灯，我们可以将椅子的位置定为(2, 0, 5)，灯的位置定为(2, 8, 5)。如你所见，世界空间如其名所示，它说明了物体在世界中的位置联系。

### 视空间（View Space）

视空间，有时也称为相机空间，但是，视空间中的原点是观察者或者相机。观察方向（观察者视线方向）定义了Z轴，由程序定义的“up”方向成为Y轴，如下图所示。



**图4. 同一个物体在世界空间(左)和时空间(右)**

左图表示一个由人形对象和观察者（相机）组成的场景。用于世界空间的原点和轴为红色。右图表示对应世界空间的视空间。视空间轴用蓝色表示。为了能说明得更清楚，视空间与世界空间的朝向不同。注意在视空间中，相机朝向Z轴方向。

### 投影空间（Projection Space）

投影空间表示在视空间上施加了投影变换的空间。在这个空间中，可视内容的X、Y轴坐标范围从-1到1，而Z轴坐标的范围从0到1。

### 屏幕空间（Screen Space）

屏幕空间常常用来表示在帧缓存中的位置。因为帧缓存通常是一张2D纹理，所以屏幕空间是一个2D空间。左上角为原点坐标为(0, 0)。X轴正方向指向右方，Y轴正方向指向下方。一个宽w像素高h像素的缓存右下角的坐标为(w - 1, h - 1)。

## 空间之间的变换

变换通常用来将顶点坐标从一个空间转换为另一个。在3D计算机图形学中，图形管道有三种变换：世界变换，视变换和投影变换。诸如平移、旋转和缩放之类的独立变换会在下一个教程中讨论。

### 世界变换

世界变换将一个顶点从对象空间转换到世界空间，它通常由一个或多个缩放，旋转和平移变换组合而成。场景中的每个对象都有自己的世界变换，这是因为每个对象都有自己的大小、朝向和位置。

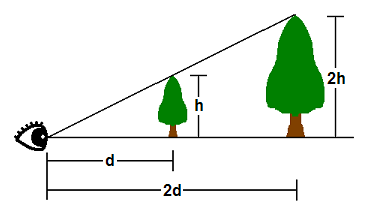
### 视变换

当顶点转换到世界空间之后，视变换就将这些顶点从世界空间转换为视空间。前面我们提到过，视空间就是从观察者（相机）角度观察到的世界。在视空间中，观察者位于原点沿Z轴方向观看。

以观察者为参照系的视空间意义不大，视转换矩阵是施加在顶点上的，而不是施加在观察者上。所以，视矩阵执行的变换与施加在观察者或相机上的变换恰相反。例如，如果你想将相机朝-Z方向移动5个单位，你需要计算出一个视矩阵将顶点沿+Z方法移动5个单位。虽然相机朝后移动，但是从相机的视点看来是顶点朝前移动。Direct3D有一个叫做**XMMatrixLookAtLH()**的方法被常常用来计算视矩阵。我们只需告知相机的位置，观察目标和观察者的向上方向（又称为up矢量），就可以获得一个对应的视矩阵。

### 投影变换

投影空间将顶点从诸如世界空间和视空间之类的3D空间转换到投影空间。在投影空间中，顶点的X和Y坐标是按X/Z和Y/Z的比例获取的。

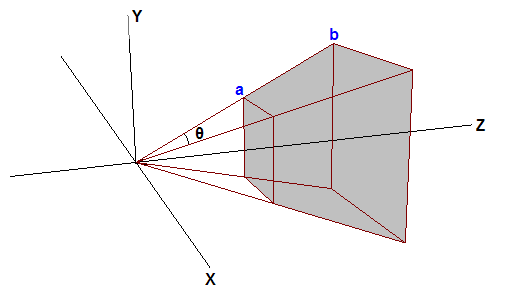


**图5. 投影**

在3D空间中，物体看起来有透视效果。即近处的物体看起来大一些。如上图所示，离开观察者d距离、高为h的树的顶端和离开观察者2d距离、高为2h的树的顶端的位置是相同的。因此，2D屏幕上的顶点的位置与X/Z和Y/Z直接相关。

有一个定义3D空间的参数称为**视场**（field-of-view，FOV）。视场表示对一个指定位置、指定观察方向来说，哪些物体是可见的。人类具有一个先前看的视场（我们不能看到自己背后的东西），我们无法看到离我们太近或太远的东西。在计算机图形学中，视场包含在一个视锥体中。视锥体由3D空间中的六个面定义，其中两个面平行于XY平面，它们叫做近裁平面（near-Z plane）和远裁平面（far-Z plane）。另外4个平面是根据观察者的水平和竖直视场定义的。视场越大，视锥体越大，观察者看到的物体就越多。

GPU会过滤掉位于视锥体之外的物体，这样就无需绘制不可见的物体了。这个过程称为剪裁（clipping）。视锥体是一个顶部被削去的金字塔。因为是针对视锥体平面进行剪裁，所以这个过程是复杂的，GPU必须根据每个顶点与平面的判断方程进行计算。因此，取而代之的是GPU通常首先进行投影变换，然后对视锥体进行剪裁操作。在视锥体上进行投影变换的结果是金字塔状的视锥体在投影空间中变为一个长方体。这是因为，如上所述，投影空间中的X和Y坐标是基于3D空间中的X/Z和Y/Z的。因此，点a和点b在投影空间中具有相同的X和Y坐标，这就是视锥体变为长方体的原因。



**图6. 视锥体**

设想有两棵树，它们的顶端都位于视锥体的边缘，d = 2h。顶部边缘的Y坐标在投影空间中就会变为0.5 (因为h/d = 0.5)。因此，任何经过投影变换后的Y值超过0.5的都会被GPU剪裁。这里的问题是0.5取决于由程序定义的垂直视场，不同的FOV值会导致GPU剪裁的值也不同。为了让处理过程更加简单，3D程序通常会缩放经过投影的X和Y值，使之的范围在-1和1之间。换句话说，任何在[-1 1] 区间范围之外的X和Y值会被剪裁。要让这种剪裁模式正确工作，投影矩阵必须对经过投影变换的顶点的X和Y坐标进行缩放，这是通过对h/d或d/h取倒数实现的。d/h也就是FOV半角的余切值。经过缩放，视锥体的顶部变为h/d \* d/h = 1。任何超过1的值会被GPU剪裁。这就是我们想要的结果。

同样的方法也用于处理投影空间中的Z值。我们想让近裁平面的Z值为0，远裁平面的Z值为1。当3D空间中的Z值 = 近裁平面的Z值，则投影空间中Z应为0；当3D空间中的Z值=远裁平面的Z值，则投影空间中的Z值应为1。经过以上处理，在[0 1]区间之外的Z值就会被GPU剪裁。

在Direct3D 11中，获取投影矩阵最简单的方法就是调用**XMMatrixPerspectiveFovLH()**方法。我们只需提供4个参数——FOVy，Aspect，Zn和Zf——就会返回一个拥有所需信息的一个矩阵。FOVy是Y方向的视场，Aspect是长宽比，它是视空间的长宽比。基于FOVy和Aspect可以计算FOVx。这个长宽比可以从渲染目标的长宽比获得。Zn和Zf是视空间中的近裁平面和远裁平面的Z值。

## 使用变换

在前面的教程中，我们写了一个程序在屏幕上绘制了一个三角形。当定义顶点缓存时，顶点位置是直接定义在投影空间中的，因此我们无需进行变换操作。现在我们理解了3D空间和变换，需要修改程序让顶点缓存定义在对象空间中。然后，我们会修改顶点着色器将顶点从对象空间变换到投影空间。

### 修改顶点缓存

因为我们是在三维空间中表示物体的，所以将上一个教程中的平面三角形变为一个立方体，这样能更好地理解其中的概念。

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 ) },

};

如你所见，我们定义了立方体的8个顶点，但我们并没有表示出三角形。如果直接将这些数据输入，那么输出的结果不会如我们所愿。我们还需要根据这8个点指定三角形构成立方体。

在立方体中，很多三角形会共享相同的顶点，重复定义相同的点会浪费空间。有一个方法可以只定义8个顶点，然后让Direct3D知道选取哪些点构成三角形。这个方法就是使用索引缓存。索引缓存包含一个列表，表示在缓存中的顶点的索引，用来指定每个三角形到底使用哪些点。如下面的代码所示：

// 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,

};

如你所见，第一个三角形由点3，1，0定义，即这个三角形顶点坐标为( -1.0f, 1.0f, 1.0f ),( 1.0f, 1.0f, -1.0f )和( -1.0f, 1.0f, -1.0f )。立方体有六个面，每个面有两个三角形，所以需要定义12个三角形。

因为每个顶点都是明确排列的，没有两个三角形共享边(至少以上定义的方式没有)，所以是按照三角形列表的方式定义的。要定义12个三角形列表方式的三角形，需要36个顶点。

索引缓存的定义方式与顶点缓存类似，我们需要指定结构体大小、类型等参数，然后调用CreateBuffer方法。类型为**D3D11\_BIND\_INDEX\_BUFFER**，因为我们使用DWORD定义数组，所以这里使用sizeof(DWORD)。

D3D11\_BUFFER\_DESC bd;

ZeroMemory( &bd, sizeof(bd) );

bd.Usage = D3D11\_USAGE\_DEFAULT;

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

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

bd.CPUAccessFlags = 0;

bd.MiscFlags = 0;

InitData.pSysMem = indices;

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

return FALSE;

创建了索引缓存之后，我们需要让Direct3D引用它们。我们需要指定一个指向缓存、格式和偏移量的指针。

// 绑定索引缓冲

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

## 修改顶点着色器

在上一个教程的顶点着色器中，我们对输入的顶点位置没有进行任何操作直接输出，这样做的原因是顶点位置已经定义在投影空间中了。但现在的顶点位置是定义在对象空间中的，所以必须在顶点着色器输出前对其进行变换。变换有3步：从对象空间变换到世界空间，从世界空间变换到视空间，从视空间变换到投影空间。要做的第一件事就是声明3个常量缓存变量。常量缓存用来存储应用程序传递到shader中的数据。在进行绘制前，应用程序通常需要将重要的数据写入到常量缓存中，在绘制过程中这些数据可以从shader中读取。在一个FX文件中，常量缓存变量的声明方式类似于C++中的全局变量。我们需要的三个变量为世界、视、投影变换矩阵，它们在HLSL中的数据类型为matrix。

声明了矩阵之后，我们就可以修改顶点着色器代码对输入位置进行变换。一个矢量乘以另一个矢量就可以进行变换，这是通过矩阵进行操作的。在HLSL中，使用的是**mul()**指令。变量声明和顶点着色器的代码如下：

cbuffer ConstantBuffer : register( b0 )

{

matrix World;

matrix View;

matrix Projection;

}

//

// Vertex Shader

//

VS\_OUTPUT VS( float4 Pos : POSITION, float4 Color : COLOR )

{

VS\_OUTPUT output = (VS\_OUTPUT)0;

output.Pos = mul( Pos, World );

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

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

output.Color = Color;

return output;

}

在顶点着色器中，每个**mul()**对输入位置进行一次变换操作。世界变换、视变换、投影变换是按顺序进行的，这是必须的，因为矢量和矩阵的乘法不符合交换律。

### 设置矩阵

更新了顶点着色器之后，我们还需要在程序中定义三个矩阵。这三个矩阵存储了绘制所需的变换。在绘制前，我们将这些矩阵数据复制到着色器常量缓存中。然后调用**Draw()**方法进行初始化，顶点着色器就会读取储存在常量缓存中的矩阵了。对于矩阵，我们需要定义3个**ID3D11Buffer**对象。**ID3D11Buffer**对象代表常量缓存中的矩阵。我们可以使用这3个对象将矩阵复制到常量缓存中。代码如下：

ID3D11Buffer\* g\_pConstantBuffer = NULL;

XMMATRIX g\_World;

XMMATRIX g\_View;

XMMATRIX g\_Projection;

要创建这3个对象，我们需要调用**ID3D11Device::CreateBuffer()**方法并指定D3D11\_BIND\_CONSTANT\_BUFFER。

D3D11\_BUFFER\_DESC bd;

ZeroMemory( &bd, sizeof(bd) );

bd.Usage = D3D11\_USAGE\_DEFAULT;

bd.ByteWidth = sizeof(ConstantBuffer);

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

bd.CPUAccessFlags = 0;

if( FAILED(g\_pd3dDevice->CreateBuffer( &bd, NULL, &g\_pConstantBuffer ) ) )

return hr;

下一步需要组合三个矩阵进行变换操作。我们想让三角形位于原点，平行于XY平面，这恰是这个三角形存储在顶点缓存中的形式，因此，无需进行世界变换，我们将世界矩阵设置为单位矩阵。我们将相机放置在[0 1 -5]，观察目标为[0 1 0]。我们可以调用**XMMatrixLookAtLH()**方法方便地计算一个视矩阵，因为我们需要让+Y轴指向上，所以使用 [0 1 0]作为up向量。最后是投影矩阵，调用**XMMatrixPerspectiveFovLH()**，视场垂直角度为90度(pi/2)，长宽比640/480(即后备缓存的大小)，近裁平面为0.1，远裁平面为100。这意味着任何比0.1还近、比100还远的物体在平面上是不显示的。这三个矩阵存储在全局变量g\_World, g\_View和g\_Projection中。

### 更新常量缓存

有了矩阵，我们必须将它们写入常量缓存中让GPU读取。常量缓存绘制以后的教程中详细讨论。现在，只需把它们想象成一个将常量传递到shader的容器。因为需要将矩阵写入常量缓存，所以需要调用**ID3D11DeviceContext::UpdateSubresource()**将它以相同的顺序存储到shader的常量缓冲中。为了简化步骤，我们创建了一个与shader中常量缓冲相同布局的结构。而且，因为在C++和HLSL中矩阵的排列顺序不同，所以首先需要对矩阵进行转置操作：

//

// 更新变量

//

ConstantBuffer cb;

cb.mWorld = XMMatrixTranspose( g\_World );

cb.mView = XMMatrixTranspose( g\_View );

cb.mProjection = XMMatrixTranspose( g\_Projection );

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

## 完整代码

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

// File: Tutorial04.cpp

//

// This application displays a 3D cube using Direct3D 11

//

// http://msdn.microsoft.com/en-us/library/windows/apps/ff729721.aspx

//

// THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF

// ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO

// THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A

// PARTICULAR PURPOSE.

//

// 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;

XMFLOAT4 Color;

};

struct ConstantBuffer

{

XMMATRIX mWorld;

XMMATRIX mView;

XMMATRIX mProjection;

};

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

// 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;

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 4: 3D Spaces", 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;

// 创建渲染模板视图

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 );

// 编译顶点着色器文件

ID3DBlob\* pVSBlob = NULL;

hr = CompileShaderFromFile( L"Tutorial04.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;

}

// 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 },

{ "COLOR", 0, DXGI\_FORMAT\_R32G32B32A32\_FLOAT, 0, 12, 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;

// 绑定输入布局

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

// 编译像素着色器文件

ID3DBlob\* pPSBlob = NULL;

hr = CompileShaderFromFile( L"Tutorial04.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( -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\_World = 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\_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;

}

//

// 设置矩阵使立方体旋转

//

g\_World = XMMatrixRotationY( t );

//

// 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 );

//

// 更新常量缓冲

//

ConstantBuffer cb;

cb.mWorld = XMMatrixTranspose( g\_World );

cb.mView = XMMatrixTranspose( g\_View );

cb.mProjection = XMMatrixTranspose( g\_Projection );

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

//

// Renders a triangle

//

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 ); // 36 vertices needed for 12 triangles in a triangle list

//

// Present our back buffer to our front buffer

//

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

}